Hindawi
Mobile Information Systems
Volume 2018, Article ID 6071580, 12 pages
https://doi.org/10.1155/2018/6071580

Research Article

HealthNode: Software Framework for Efficiently Designing and Developing Cloud-Based Healthcare Applications

Ho-Kyeong Ra,[1] **Hee Jung Yoon**,[1] **Sang Hyuk Son**,[1] **John A. Stankovic**,[2] **and JeongGil Ko**[3]

[1]*Information and Communication Engineering, Daegu Gyeongbuk Institute of Science and Technology (DGIST), Dalseong-Gun, Daegu, Republic of Korea*
[2]*Computer Science, University of Virginia, Charlottesville, VA, USA*
[3]*Software and Computer Engineering, Ajou University, Yeongtong-gu, Suwon, Republic of Korea*

Correspondence should be addressed to JeongGil Ko; jgko@ajou.ac.kr

Received 19 January 2018; Accepted 1 March 2018; Published 19 April 2018

Academic Editor: Andrea Gaglione

With the exponential improvement of software technology during the past decade, many efforts have been made to design remote and personalized healthcare applications. Many of these applications are built on mobile devices connected to the cloud. Although appealing, however, prototyping and validating the feasibility of an application-level idea is yet challenging without a solid understanding of the cloud, mobile, and the interconnectivity infrastructure. In this paper, we provide a solution to this by proposing a framework called HealthNode, which is a general-purpose framework for developing healthcare applications on cloud platforms using Node.js. To fully exploit the potential of Node.js when developing cloud applications, we focus on the fact that the implementation process should be eased. HealthNode presents an explicit guideline while supporting necessary features to achieve quick and expandable cloud-based healthcare applications. A case study applying HealthNode to various real-world health applications suggests that HealthNode can express architectural structure effectively within an implementation and that the proposed platform can support system understanding and software evolution.

1. Introduction

The advancement of Internet of Things (IoT) applications has allowed various data points collected from a large number of heterogeneous devices to be gathered in a single repository for application development. Naturally, the distributed nature of these systems concentrates on the cloud infrastructure for achieving novel application designs with the support of integrated data processing and effective resource management.

In particular, healthcare-related IoT applications have developed to the point where it not only preserves the safety and health of individuals but also improves how physicians deliver care. Using smart and mobile devices, healthcare IoT allows the delivery of valuable data to users and lessens the need for direct patient-physician interaction.

Despite their attractiveness, however, implementing such cloud-based applications is not in any way trivial. This is especially true for a large number of nontechnical researchers in the healthcare domain. While health and IoT domains are being revolutionized in convergence, designing an effective cloud application asks the researchers to know various technical domains from the server operation, local and web-application languages, to data communication protocols. For example, for efficient web development, developers typically require knowledge of up to five different programming languages, such as JavaScript, HTML, CSS, a server-side language such as PHP, and SQL.

For overcoming these challenges, a recently introduced software platform called Node.js [1] has gained a significant amount of traction in the developer communities. Specifically, Node.js is a scalable single-threaded server-side JavaScript environment implemented in C and C++ [2]. Developers can build scalable servers without using threading but rather by using a simplified model of event-driven programming that uses callbacks to signal the completion of a task [3]. Owing to its simplicity, industry leaders such as Microsoft [4], IBM [5], Netflix [6], PayPal [7], and Walmart [8] have integrated the

support of Node.js into their cloud platforms. Notwithstanding the recent developments, Node.js's approach in developing web applications has made it an attractive alternative to more traditional platforms such as Apache + PHP and Nginx servers.

One of the many benefits of using Node.js is its architecture that makes it easy to use as an expressive, functional language for server-side programming [9]. Although it may be trivial to perform development in Node.js once a developer fully comprehends the language, there are many obstacles prior to being able to implement real cloud-based applications for beginners. For example, Node.js requires multiple initialization steps, such as the configuration of the HTTP package and route.js, which guides where the request is routed. Due to the framework being relatively young and the software yet reaching maturity, developers still face the lack of documentation support and can face troubles in receiving support from the development community.

To the best of our knowledge, there has not yet been a Node.js software design or implementations guideline documentation, where developers are influenced to create well-structured healthcare applications. Nevertheless, for software developers, the fact that organizing the program design with solid boundaries is crucial in a successful real-world deployment. Designing an ad hoc model will stimulate logical complexity and cause difficulties in maintaining and updating the application over time. Moreover, without a sturdy guideline documentation, it takes a significant amount of time and needless effort even to configure the primary application development environment.

In this paper, we present a practical solution for implementing cloud-based healthcare applications by providing a framework called HealthNode, which consists of (1) a software design and (2) essential APIs and implementation guidelines for prototyping. We specifically emphasize on healthcare applications rather than other types of applications because Node.js is capable of supporting the complex requirements that healthcare applications ask, which are the needs to support multiple patient-doctor connections, exchange medical data with a unified language and data format, and allow reusability of the developed medical components. Nevertheless, we put as one of our future work to expand HealthNode to be applicable for other applications.

To simplify the use of Node.js, our software design helps developers to easily observe data flow, create modules, and add in functions even without detailed descriptions, which Node.js lacks. We use a top-down approach that structures our software design with a hierarchy of modules and a divide-and-concur approach that organizes the tasks for each module. The top-down approach helps developers to observe application flow from the main module, which acts as the main class in Java, to other submodules in a top-down manner. It is essentially the breaking down of the application development to gain insight into its compositional sub-modules. The divide-and-concur approach breaks down each module into submodules that specify target tasks for each module. This tactic also enables multiple developers to work on different portions of the application simultaneously.

To guide the development of cloud-based healthcare applications using Node.js, we provide APIs and a guideline that constructs the skeleton of essential components within the implementation. We focus primarily on the back end of the server, which contains the core operational functions. Our envision is that HealthNode would overall influence in developing cloud-based healthcare applications.

The contributions of this work can be summarized in three-fold:

(i) A software design that tackles the challenges of maintaining and updating cloud applications developed using Node.js. This design layout will support the resulting application to comprise well-defined, independent components which lead to better maintainability. In addition, new capabilities can be added to the application without major changes to the underlying architecture.

(ii) APIs and implementation guideline that provides explicit, but straightforward instructions for developing general-purpose Node.js prototype. This guideline instructs how to (1) create prototypes by using a limited set of APIs, (2) divide modules and organize function, (3) allow a client communicate to a server, (4) utilize cloud application database, and (5) handle uploading and downloading files. Through this guideline, developers can focus on implementing application logic.

(iii) Examples of healthcare application that open the possibilities for a new structure of healthcare to be developed. Our software design and guideline can be the basis for developing a variety of cloud-based healthcare applications. We provide examples of systems that can be enabled by our work.

In Section 2, we discuss some of the technical challenges of implementing cloud applications using Node.js. We introduce the overall structure of Node.js architecture, libraries, modules, and functions that make up Node.js applications, as well as HealthNode's software design in Section 3. In Section 4, we present our guideline that instructs the implementation procedures for developing cloud-based healthcare applications and introduce couple potential applications where our work can be applied in Section 5. Finally, we position our work among others in Section 6 and conclude the paper in Section 7.

2. Challenges

Technically, for developing a cloud-based healthcare application using Node.js, there are many materials both online and offline for users to easily get a start on building a software environment. However, based on our experiences, the level of these tutorials mostly remains in the beginner level, and due to its relatively new life cycle, it is fairly difficult to identify the information required to perform more advanced tasks. Furthermore, practical troubles that typically arise within this development phase include the lack of (1) a formal software structure, (2) fundamental

guidelines for advanced functionality implementation, and (3) real-application examples. We use the remainder of this section to discuss the importance of such support and the challenges that application developers can face due to such limitations.

2.1. Usability. There is an active community that supports Node.js. More developers watch the repository of Node.js at GitHub than other recently trending software environments. Nevertheless, Node.js is relatively new compared to traditional web-application frameworks such as ASP.NET. Therefore, naturally, in contrast to these older frameworks, Node.js lacks documentation and examples on how to structure the overall implementation. Due to a small number of easy-to-follow guidelines, Node.js is not commonly used to its full capability where developers can create data exchange applications for connected infrastructure such as IoT applications. An ideal application should allow clients to submit data to a server and the server to respond back to the clients to fully utilize the cloud computation power.

2.2. Feasibility. For beginners to use Node.js for implementing a fully functioning prototype, it takes a significant amount of time and effort to feasibly set up the basic application environment, properly route incoming and outgoing information, and format the overall application structure. Ideally, to catalyze the development of various cloud applications, this process should be simple to both understand and implement.

2.3. Maintainability and Extensibility. In designing the back end of a cloud application using Node.js, existing tutorials often promote examples using only a single module containing all the possible functions, regardless of system design. Using a single module limits the tasks to be distributed to other modules and therefore diminishes advantages of designing an organized implementation. Although Node.js supports building a hierarchy of multiple modules, ad hoc plans of software structuring at the hierarchy level can cause increased logical complexity without solid boundaries between heterogeneous modules. Moreover, to maintain and update applications efficiently, proper documentation is essential. However, documenting within a single module or multiple ad hocly planned modules can add additional burden to the code review process. To support these issues, a software design with an organized structure of modules will not only help developers in designing their software but also benefit them in maintaining their applications. Furthermore, such a repository of modules can ultimately benefit the application to be more conveniently extensible over time.

3. Architecture

This section first describes the overall structure of the Node.js architecture, libraries, modules, and functions that make up a typical Node.js application. We then present HealthNode design that uses a top-down approach and the divide-and-conquer strategy, which are both the essence of any software development.

3.1. Background

3.1.1. Architectural Description of Node.js. Well known for its event-based execution model, the Node.js platform architecture uses a single thread for executing application code which simplifies application development. However, heavy calculations and blocking I/O calls that are executed in the event thread prevent the applications from handling other requests. Node.js tackles this issue by using event loop and asynchronous I/O to remain lightweight in the face of data-intensive, real-time applications [10]. The Node.js execution model is different to the thread-based execution model where each client connection is processed by a separate thread. Overall, the platform must coordinate data input and output adaptably and reliably over a different range of distributed systems.

3.1.2. Libraries and Modules of Node.js. Node.js comes with an API covering low-level networking, basic HTTP server functionality, file system operations, compression, and many other common tasks. Moreover, the available external libraries of Node.js can add more capability in a module form. The modules are delivered by public or private package registries. The packages are structured according to the CommonJS package format and can be installed with the Node Package Manager (NPM) [11].

Our software design works off the Express library, which is one of the Node.js packages that support the rapid development of Node.js cloud application [12]. It helps set up middlewares to respond to HTTP requests, specifies a routing table which is used to achieve various action based on the HTTP method and URL and allows to present HTML pages based on passing arguments to templates dynamically. Other than public libraries, local modules also can be referenced either by file path or by name. Unless the module is the main module, a module that is referenced by a name will map into a file path.

3.1.3. Functions of Node.js. In programming, a function is defined as the portion of code that performs a specific task with series of statements. It has a capability of accepting data through parameters for a certain task and returning a result. In Node.js, a function requires extra implementation for routing requests. To support the ease of routing, the Express package enables the capability to create middleware functions. Middleware functions allow setting up a routing path in one line. In addition, middleware functions are only accessible by clients and are not accessible by back-end computation functions. For example, when a client submits and requests data to the server, one of the middleware functions is triggered, and output is returned to the client. For accessing data, middleware functions only access data through shared objects or a library such as MongoDB.

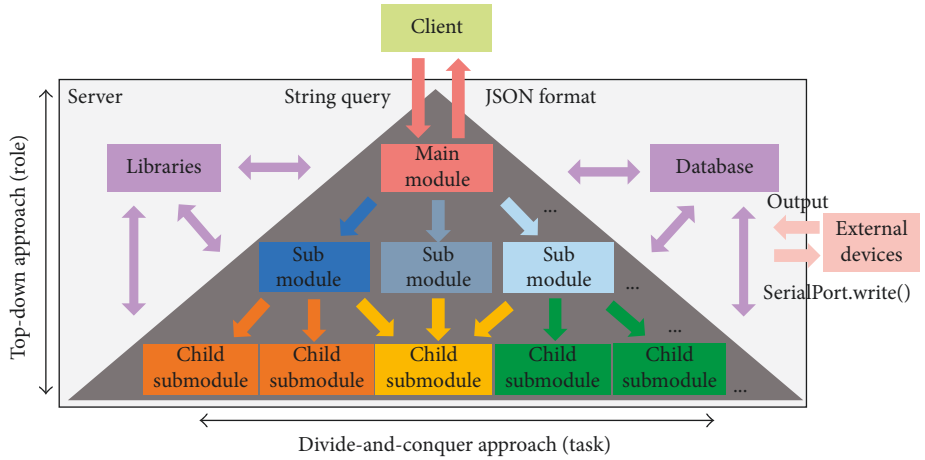

Figure 1: HealthNode software design.

3.2. HealthNode. We now detail two software design processes that make up HealthNode. The top-down approach helps developers to couple and decouple modules while the divide-and-conquer approach guides developers to divide the task into simpler modules while enabling multiple module developments concurrently. We use the top-down approach first and then the divide-and-conquer approach in the latter step so that the roles between the modules are first defined before tasks are assigned to each of the roles. Although both of these techniques are commonly used in other types of web and application developments, to the best of our knowledge, there has not been a foundational software design that supports Node.js implementation structure.

3.2.1. Top-Down Approach. The top-down and bottom-up approaches play a key role in software development. The top-down approach is a standard inheritance pattern which decomposes a system to gain insight into subsystems. Each subsystem is then refined in yet greater detail in many additional subsystem levels until the entire specification is reduced to base elements. This process continues until the lowest level of the system in the top-down hierarchy is achieved. In a bottom-up approach, the base elements of the system are first specified. These elements are then linked together to form larger subsystems until a complete top-level system is constructed. Using the bottom-up approach may be beneficial for implementing first-level systems for early testing. However, the bottom-up approach is not suited particularly for our software design due to its requirement of permitting space to grow. Since our study focuses on prototyping, which requires a continuous update, it must be easy to add and couple modules. However, the bottom-up strategy does not allow this, and over time, organization and maintenance issues may exist.

In the software design of HealthNode, a hierarchy structure creates a connection between modules that supports data flow. Figure 1 visualizes how HealthNode maps the top-down perspective by starting from the main module and initializing submodules. Each submodule can also have multiple child submodules with external packages. In the main module, a shared component such as a database is initialized, and necessary submodules are coupled.

The top-down approach is effective when the application idea is clear, and the system design is ready prior to implementation. Looking at Figure 1, a top-down design concentrates on designing vertical hierarchy levels and uses couples to connect data flow. The coupling process happens during the period when modules are added. This process finishes when the submodule is ready to be used after testing and connecting to a higher module. In addition, the coupling process can be used for sharing child submodules. In cases when already implemented child submodule is required by other submodules, each of the submodules can couple to preimplemented child submodule to avoid redundancy. During implementation, coupling is used to create a weak connection between modules. In other words, a submodule is allowed to use a child submodule only once during implementation. The decoupling process can be easily done due to the weak connection between all the modules.

The decoupling process supports maintaining and extending the application over time. When a particular submodule expands, in such case as having two different tasks, the submodule needs to be decomposed. To decompose a module, the decoupling process closes the connection between the higher submodules to the current module. This will lead the decoupling process to stop allowing higher modules to use the current submodule. Overall, the coupling and the decoupling process is completed by initializing and removing the connection between modules and submodules.

3.2.2. Divide and Conquer. Divide and conquer is a concept of recursively breaking down a system into two or more subsystems until these become simple enough to be labeled directly. The outputs to the subsystems are then combined to provide an output to the highest system.

The divide-and-conquer approach is different than the top-down approach in a sense that the top-down approach defines hierarchy levels while the divide-and-conquer method focuses on horizontally dividing a task in each

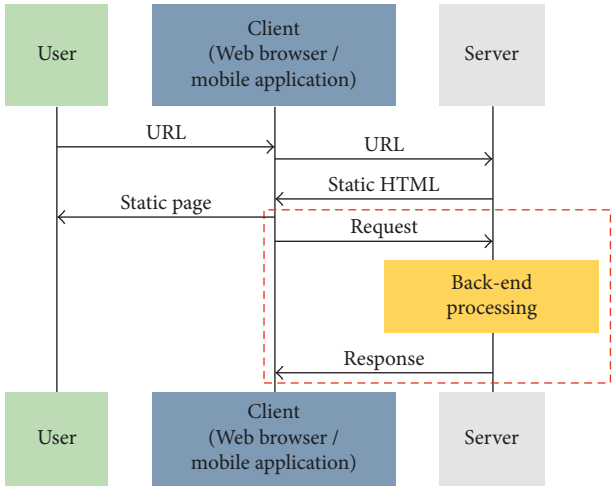

Figure 2: Location of HealthNode in a general cloud application.

level to specify a task. The higher the module level is, the application design uses task characteristics to divide modules. The lower the module level is, the design concentrates on the functionality of higher modules' basic requirements. Moreover, in HealthNode's software design, the divide-and-conquer process also reduces redundancy by dividing a module into groups of reusable and unique task-oriented functions. Since one of the functions or middleware functions within the higher module uses submodules, the divide-and-conquer technique can be easily applied using one or two lines of code.

4. Implementation

In this section, we present the implementation components of HealthNode and a step-by-step guideline that instructs the implementation procedures for developing Node.js cloud applications using the proposed framework. Figure 2 shows a general application execution flow from users to clients and then from clients to a server. We focus on the core components located at the back end of the server and the communication components that are used between the clients and the server. Note that we do not look into the details of the server's front end, since there are many existing examples of front-end frameworks available [13–15].

In HealthNode, the proposed implementation guideline follows five essential steps:

(1) A hierarchy structure is set up between modules.

(2) A function is placed on each of the modules to be used for computation.

(3) A middleware function is added into the function, which directly communicates to a client.

(4) During the initialization process of the main module, database information is configured to be used throughout the implementation process.

(5) The client prepares the communication procedures for testing.

Specifically, we describe the implementation details of structuring the hierarchy of the back end, operations that take place in the modules, and the communication process between the client and the server in detail using the following subsections. We will use code snippets to present usage implementations for the concepts in HealthNode.

4.1. Hierarchy. In the main module, all of the necessary libraries required for application development are imported during the initialization stage. As the sample implementation on lines 1 to 10 in Listing 1 shows, the Express Library, along with supporting libraries, is first imported to construct a hierarchical structure of the application along with enabling most of the HTTP utilities. The MongoDB library is then imported to be used for creating a connection between the back-end server and the database.

The main module and the submodules are connected through a coupling process as exemplified on lines 20 and 21 in Listing 1. Commonly required libraries are shared using parameters during coupling process. The coupling process allows a client to access the submodule method. Following the top-down approach, although conventional Node.js application allows methods in the main module, HealthNode recommends methods to be exclusively in the submodules to avoid design complexity. Having functions in the main module can cause documentation and maintenance issues due to its limited role of initializing the core tasks. In contrast, submodules can couple to and call functions in the child submodule as shown on lines 25 to 30 in Listing 1. When calling functions in the child submodule, parameters with information can be passed to acquire necessary information. All modules are coupled with only one or two lines of code, which allows for a simple decoupling process for extending the application when needed. This procedure is the first principle in the divide-and-conquer process.

4.2. Module. As Figure 3 shows, each of the submodules contains initialization blocks and functions that include libraries to be used within the current submodule. Note that each function consists of multiple middleware functions. For the function to be part of a module, it requires each function to be exported as illustrated through Listing 2 (lines 1 to 5). A middleware function is required when a client needs to communicate to a server. The module containing a function without middleware functions usually sits as the lowest child submodule unless it requires assistance from lower level child submodules. In other words, if the function is only used for the back-end computation and called by a higher submodule, the function is accessible by calling the function name such as "ChildSubModuleFunction" as Listing 2 shows on lines 17 to 22. The function can further process computation tasks with the data received through a function parameter. Each submodule requires at least one function that may only have a computation task or have both computation task and middleware functions.

4.3. Middleware Functions. As Listing 3 shows, a middleware function can be added to a function. Each middleware function contains `REQUEST` and `RESPONSE` parameters.

```
(1) //Necessary libraries and variables
(2) var express=require( express );
      ...
(9) var mongoClient=require( mongodb ).MongoClient;
(10) var mongoDBuri =  mongodb://localhost:27017/database
      ...
(20) //Coupling sub modules
(21) require( ./SubModuleONe.js )(app, mongoClient,mongoDBuri);
      ...
(25) //Coupling to child sub-module and its function
(26) var ChildSubModule = require( ./ChildSubModuleOne ).FunctionName;
      ...
(29) //Inside of fuction or method call child sub-module function
(30) ChildSubModule(data);
```

Listing 1: Initializing libraries, submodules, and child.

REQUEST and RESPONSE is a basic operation and key feature for enabling the communication between the clients and the server. When clients need information, a mobile application or a web browser sends a REQUEST message to the server. A request message is sent in the form of a query string. During the REQUEST message processing, the body of the REQUEST is primarily used. The message is then unpacked by using the "stringfy" API function from the "querystring" library. After unpacking the query, the message is translated into values of an object. Values of an object are used for executing tasks or computation. To execute tasks, middleware functions can also call functions offered through other modules to process back-end computation tasks. In addition to the back-end computation, external executables, such as compiled machine learning algorithms, can also be executed using the "child_process" library as shown on lines 1 to 8 in Listing 4. Additionally, lines 10 to 20 in Listing 4 show an example of using the "SerialPort" library from the NPM, so that the development boards enabled with serial communication can be controlled by the WRITE command and output data. Furthermore, REQUEST can be used to receive file-level data. By using "fs" library, an incoming file can be processed and saved to a particular location (cf. lines 26 to 35 in Listing 4).

Each REQUEST from the clients is answered at the server using a RESPONSE. Specifically, RESPONSE has the role of returning messages and is in the form of JavaScript Object Notation (JSON) message. In JSON messages, heterogeneous information can be stored as an object. Within the object, there is a collection of <field name,data> pairs and there can be a single object or an array of objects. These objects can be placed on the JSON message for exchange. Once the JSON message prepares necessary information for the client, information is sent back to the client by using the WRITE function as we present in Listing 5. The uploaded file can be accessed from the client by using the "fs" library shown on lines 41 to 49 in Listing 4.

4.4. Database. To provide or store information, having a database is also essential. We specifically chose to support MongoDB in HealthNode. In contrast to MySQL, MongoDB

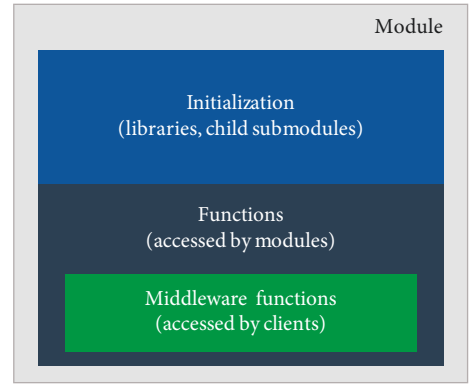

Figure 3: General structure of a module.

uses dynamic schemas, which means that records can be created without first defining the structure [16]. In MongoDB, three basic (yet important) operations are INSERT, FIND, and DELETE. To access the database, the "MongoDB" library is initialized in the main module and shared with the submodules. Prior to any database operation, a connection is established by using the CONNECT function along with the URL and port number of the database server. With the established connection, query string information can be inserted, deleted, or used for finding data as we show an example in Listing 6. For updating documents in the database, the documents are replaced by using DELETE and INSERT commands.

4.5. Client. Communicating with the server from a client is also an essential operation. For mobile applications, typically, the communication between the client and the server is accomplished by using a web browser or HTTP library. For a web browser, the client retrieves or sends information by using POST operation with string query. Specifically, the web browser uses JavaScript to embed information in string query and requests POST to the server. After sending REQUEST, a RESPONSE from the server is returned to the client in JSON message format. As presented in Listing 7, the JSON

```
(1) module.exports = function(app, mongoClient,mongoDBuri){
(2)   app.post ( /method , function(req, res){
(3)     //Method contents
(4)   });
(5) };
      ...
(17) function ChildSubModuleFunction(data){
(18)   //Use data
(19)   info =  Information ;
(20)   return info
(21) }
(22) exports.ChildSubModuleFunction = ChildSubModuleFunction;
```

Listing 2: Structure of a middleware function and child submodule function.

```
(1) var querystring = require( querystring );
(2)   app.post( /method , function(req, res){
(3)
(4)     var postObjectToString = querystring.stringify(req.body);
(5)     var postObject = querystring.parse(postObjectToString);
(6)     //Data on  Info  is accessed from post object(query string)
(7)     var Info = postObject[ Info ];
(8) ...
```

Listing 3: Example of middleware function.

message is parsed into readable objects and accessed by a client web browser. In a mobile application, specifically for androids, the Apache HTTP library is used to simulate the web browser. The POST operation in mobile applications works similarly as that of the web browser.

4.6. HealthNode API. For designing the API set, we gathered general requirements from a number of previous works on health and home monitoring projects [17, 18] and implemented necessary methods for the HealthNode API. The API uses and extends the Express library which allows developers to add necessary methods by following the conventional Express implementation rules [12]. We include a library for Android mobile and web browser applications to communicate with the HealthNode server applications. Both mobile and web applications API enables sending JSON messages and files by accessing middleware functions on the server. The HealthNode APIs can be installed using the NPM install commands. Starting the server and importing basic libraries such as "MongoDB" and "fs" are already managed by simply importing the API. Other than the previously mentioned libraries, APIs also use the existing "SerialPort" library to communicate with external development boards such as an externally connected Arduino or Raspberry Pi.

4.7. Security. The security of data exchange between a server and a client can be protected using Transport Layer Security (TLS), which is a well-known protocol that provides privacy and data integrity. By following the Express API on TLS, HealthNode can enhance the security during data exchange. Moreover, the privacy of patient health information is password protected. When a developer implements the data exchange process, login function needs to be used prior to requesting data from the server. For example, an Android application or a web browser client can request data from the server after obtaining login approval.

5. Case Study-Based Evaluation

There are various types of cloud-based healthcare applications that can take advantage of HealthNode's software design and implementation guidelines. Such examples include applications that simply log information to the cloud through the web, exchange medical or healthcare information between mobile devices, or execute physical component actuation through a local network. In general, HealthNode supports the fundamental requirements for developing of cloud applications. These requirements include sending/receiving data, constructing a database to store information, calling external executables for machine learning algorithms, and ensuring space for the application to be expanded.

In this section, we evaluate HealthNode by providing possible application scenarios of how our software design and implementation guideline can be used for different mobile-cloud application development. Note that the case studies benefit from HealthNode due to the framework (1) following intuitive design strategies which help external field members to understand the system design and enhance the medical features of the system, (2) containing a practical

```
(1) //Execute external program or classifier
(2) var querystring = require( child_process ).executable;
        ...
(4) app.post( /method , function(req, res){
(5)   executable( ./execProgram  + testFile , function(err, stdout, stderr){
(6)     //output is on stdout
(7) }
(8) });
        ...
(10) //For serialport
(11) var SerialPort = require( serialport );
(12) var Readline = SerialPort.parsers.Readline;
(13) var port = new SerialPort( /dev/ttyUSB0 ,{baudrate: 9600 });
(14) var parser = port.pipe(Readline({delimiter:  \r\n }));
(15) //Getting data from external board
(16) parser.on( data , function(data){
(17)       console.log(data);
(18) });
(19) //Sending data to external board
(20) port.write( some data );
        ...
(26) //File upload/download
(27) var fs = require( fs );
        ...
(30) //File upload:
(31) app.post( /uploads , function(req, res) {
(32)   fs.readFile(req.files.fileU.path, function(err,data){
(33)     var dirname =  /file/dir/location ;
(34)     var newPath = dirname +  /uploads/  + req.files.fileU.originalname;
(35)     fs.writeFile(newPath, data, function (err){
        ...
(41) //Access file on server:
(42) app.get( /uploads/:file , function(req, res){
(43)     file = req.params.file;
(44)     console.log( File requested:  + file);
(45)     var dirname =  /file/dir/location ;
(46)     var img = fs.readFileSync(dirname +  /uploads/ + file);
(47)     res.writeHead(200, { Content-Type :  image/jpg });
(48)     res.end(img,  binary );
(49) });
```

Listing 4: Calling child submodule function, controlling external development board using serial communication, and managing file exchange.

set of medical application-related methods which allows the developer to utilize or alter the given functions to complete cloud-based health application implementation, and (3) allowing these applications to communicate with external sensing systems.

5.1. Case Study: AsthmaGuide. To evaluate HealthNode's design and framework, we use one of our previous works, AsthmaGuide [17], as a case study. AsthmaGuide is a monitoring system for asthma patients in which a smartphone is used as a hub for collecting indoor and outdoor environmental information and physiological data. Specifically for indoor environments, we use Sensordrone [19] to measure information of the patient's surroundings such as the temperature, humidity, and air quality. For outdoor environmental data, we use a national database to gather information of air quality, pollen count, and asthma index. Furthermore, we collect physiological data from the patients by collecting their lung sounds using an electronic stethoscope, and present questionnaires that patients fill out manually on an Android application. The data collected over time is then displayed through a cloud web application for both patients and healthcare providers to view.

By utilizing HealthNode, AsthmaGuide first gathers requirements, and consequently, each of the requirements is placed into a designated role level with the top-down approach and assigned a job with the divide-and-conquer approach as shown in Figure 4. The requirements are directly linked with middleware functions in which each requirement is responsible for exchanging data between the client and the server.

Figure 4 also shows the list of 26 middleware functions that are required to implement AsthmaGuide. These middleware

```
(1) module.exports = function(app, mongoClient,mongoDBuri){
(2)   app.post( /method , function(req, res){
(3)     res.write( Response back );
(4)     res.end();
(5)   });
(6) };
```

Listing 5: Example of responding to REQUEST.

```
 (1) module.exports = function(app, mongoClient,mongoDBuri){
 (2)   app.post( /method , function(req, res){
 (3)     mongoClient.connect(mongoDBuri, function(err, database){
             ...
(11)       collection = database.collection( testingCollection );
(12)       //Instead of insert other functions(delete, find) works also
(13)   collection.insert(postObject, function(err, records){});
```

Listing 6: Example of MongoDB INSERT operation.

```
 (1) <script>
 (2) var data_server = http://localhost/methodName ;
 (3) var requestInfo = {requestInfo: data };
 (4) var receivedData = {receivedInfo:  };
 (5) $(document).ready(function(){
 (6)   $.post(data_server, requestInfo).done(function(data){
 (7)     objArr = JSON.parse(data);
 (8)     if(objArr !=  ){
 (9)       $.each(objArr, function(key1, obj){
(10)         receivedData.receivedInfo = obj[ receivedInfo ];
(11) });
```

Listing 7: Example of requesting to the server from a web browser.

functions provide necessary results back to a web browser or a mobile device while accepting incoming data from clients. Each of the middleware functions is mapped to the HealthNode design pattern accordingly to the categorized alphabetic letter. All middleware functions are prebuilt into HealthNode and are accessible by importing the HealthNode library. The library implementation follows the HealthNode design pattern, and a developer can reference the implementation as well as add the middleware functions to enhance their application.

For example, when a patient needs to upload his or her collected indoor and outdoor environmental data as well as physiological data to the server, middleware functions such as "Patientlogin," "PatientRetrieveZipCode," "PatientRetrieveCountryCode," "PatientInsertData," "FileUploadImageFile," "FileUploadWaveFile," "ClassifyLungSound," and "GeneratePatientAdvice" are used. Note that "ClassifyLungSound" and "GeneratePatientAdvice" are AsthmaGuide application-specific functions. During mobile or web-application implementation, the AsthmaGuide developer calls the needed middleware functions to request and insert data to the server. Besides these simple data upload operations, AsthmaGuide requires far more methods when the system starts interchanging information between a patient and a healthcare provider. Therefore, HealthNode can help reduce the burden of developers by easing the development complexity in the process of implementing and maintaining cloud applications.

5.2. Case Study: Smart Home Automation Framework. Another case study that we apply HealthNode on is a system called the Smart Home Automation Framework (SHAF) [18]. IoT covers a various network of physical objects with actuation and sensing embedded units. Under the hood, the communication between devices is connected through multiple network protocols. One of the domains that take advantage of IoT is home automation. Home automation uses different types of network protocols such as Wi-Fi, Bluetooth, and ZigBee. However, existing home equipment often requires

A. StartServer
B. DoctorRegister
C. DoctorRegisterCheck
D. DoctorLogin
E. DoctorCommentPatient
F. DoctorRetriveComments
G. DoctorRetriveCommentsOfAPatient
H. PatientRegister
I. PatientRegisterCheck
J. PatientLogin
K. PatientRetriveDataByDate
L. PatientRetriveAllData
M. PatientRetrieveMostRecentData
N. PatientRetrieveAdvice
O. PatientRetrieveZipCode
P. PatientRetrieveCountryCode
Q. PatientInsertData
R. PatientRetriveCommentsFromDoctor
S. EnviromentRetriveInfo
T. FileUploadImageFile
U. FileUploadWaveFile
V. FileUploadMp4File
W. FileUploadRetrieveFile
X. ClassifyLungSound
Y. GeneratePatientAdvice
Z. SendAlert

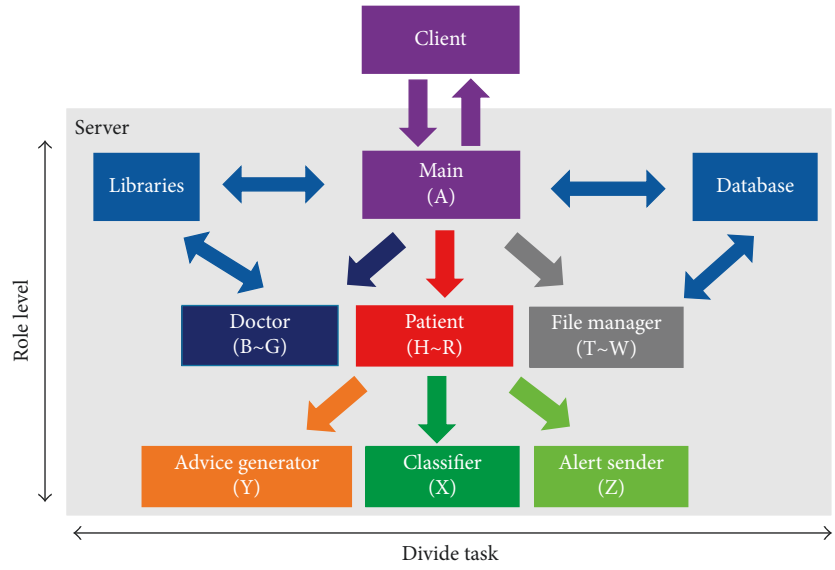

Figure 4: Designing AsthmaGuide according to HealthNode design pattern.

network communication-enabled power plugs or devices that hold a unique communication protocol specified by a manufacturer. While these types of equipment typically follow a standard communication capability, each device is limited to communicate only within the same network protocol. The goal of SHAF is to resolve issues that can be raised due to such limitations.

Specifically, SHAF targets to provide and maintain a comfortable and healthy living environment for patients. For example, the surrounding temperature is a critical metric for those who are sensitive to cold or hot temperatures such as people with chronic health conditions, given that extreme (than normal) temperatures can aggravate various symptoms. Figure 5 illustrates an example of SHAF monitoring and actuating the smart home by using the HealthNode framework. We use Raspberry Pi with ZigBee as a smart central server and an Arduino with ZigBee as sensor nodes. For SHAF ZigBee communication, multihop communication is enabled for larger homes. The server accepts incoming JSON queries where a client can request sensor readings or operate an actuation unit. While our current client application is implemented on Android and Windows smartphones, any programming languages that can support JSON message requests can communicate with the smart home's central server.

SHAF's server should have the capability of handling requirements or multiple middleware functions such as "AddSensor," "RemoveSensor," "RefersehSensorReadings," "ActuateSensor," "LearnHomeUsage," "AutonomusMode," and "HealthyLivingEnvironmentMode." Although the quantity of the middleware functions is not large, the use of HealthNode allows for easier software maintenance since it instructs task-specific submodules rather than one large

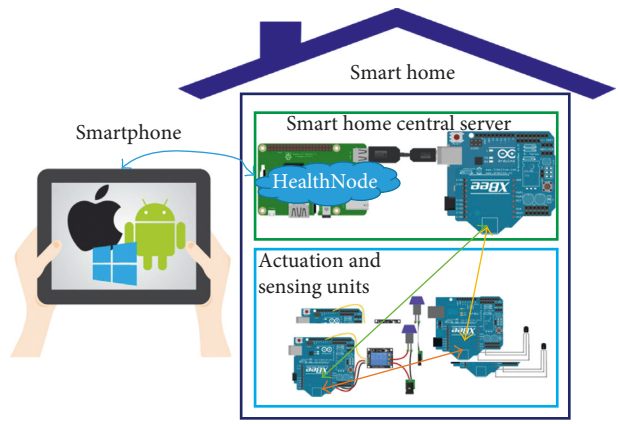

Figure 5: Communication map of SHAF.

module. During maintenance, the developer couples and decouples the submodule by changing one or two lines of code. Specifically, using HealthNode's software design, we structure the architecture of SHAF so that it contains three submodules (e.g., patient, caregiver, and sensor manager). Followed by the architecture, multiple middleware functions are implemented. For example, the patient and the caregiver submodules are accessed from the mobile or web browsers by using JSON message requests. When there is a request, one of the middleware function returns the result to the client followed by the database operation. The server also logs up-to-date sensing and actuation values by using the "SerialPort" library and middleware function in the sensor manager submodule. Furthermore, stored data can be used by a child submodule with a machine-learning algorithm for automatic environment configuration based on a user's

preference. Note that all of the middleware functions are prebuilt into HealthNode and are accessible by importing the HealthNode library.

6. Related Work

Existing Node.js applications are spread across various fields of study including IoT or web [20–25], medical [26, 27], transportation [28], and environmental [29] domains. To develop a well-structured application using Node.js, Frees [30] proposes a way to overcome many challenges in teaching web development by placing Node.js in the computer science curriculum. He presents a semester-long, 14-week course outline to allow students to fully understand the use of Node.js and be able to apply it for web development. Although Node.js is easily utilized compared to standard techniques, Node.js still requires a strong background before it can be used to its full capacity. Understanding Node.js development merely at a surface level will result the application to stay simple and be more prone to mistakes and difficulties in maintaining the application over time. A study by Ojamaa and Düüna [31] stated that mistakes are more common with Node.js applications because programmers lack the extensive experience of writing JavaScript application.

Many attempts have been made to solve this issue. There are existing books [9, 32–38] and online tutorials [39, 40] that go into the depths of using Node.js. Although these sources provide extensive guidelines, there has not been a straightforward software design that allows developers to get a complete picture of the Node.js programming structure for general-purpose application prototype.

7. Conclusion

Healthcare applications are emerging at an exponential rate, and application systems in the remote healthcare application domain are becoming critical sectors in IoT research. As Node.js becomes a well-used essential tool for developing cloud-based applications, we propose HealthNode, which is a general-purpose framework for developing healthcare applications on cloud platforms using Node.js. The principal goals of HealthNode are to provide explicit software design, API, and guidelines to achieve quick and expandable cloud-based healthcare application. We specifically tailor HealthNode for healthcare applications due to a significant potential for addressing many of the challenges of providing accessible, cost-effective, and convenient healthcare. With development support systems such as HealthNode, we envision that the development of mobile-connected application systems will inevitably increase in the healthcare domain.

Data Availability

The data used to support the findings of this study are available from the corresponding author upon request.

Conflicts of Interest

The authors declare that they have no conflicts of interest.

Acknowledgments

This work was supported in part by the DGIST R&D Program of the Ministry of Science, ICT and Future Planning (18-EE-01), the Global Research Laboratory Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Science, ICT and Future Planning (NRF-2013K1A1A2A02078326), the DGIST Research and Development Program (CPS Global Center) for the project "Identifying Unmet Requirements for Future Wearable Devices in Designing Autonomous Clinical Event Detection Applications", and the Ministry of Trade, Industry and Energy and the KIAT through the International Cooperative R&D Program (no. N0002099; Eurostars-2 Project SecureIoT).

Supplementary Materials

We have included an NPM installable supplementary in which a developer can install the library and execute the developer's application using Node.js. The supplementary folder also consists of Android and Web client source codes for uploading data and getting results back from the server. (*Supplementary Materials*)

References

[1] Joyent Inc., Node.js, http://www.nodejs.org/, 2016.
[2] R. R. McCune, *Node.js paradigms and benchmarks*, Striegel, Grad OS, 2011.
[3] S. Tilkov and S. Vinoski, "Node.js: using javascript to build high-performance network programs," *IEEE Internet Computing*, vol. 14, no. 6, pp. 80–83, 2010.
[4] Microsoft, Microsoft azure, https://www.azure.microsoft.com/en-us/develop/nodejs/, 2016.
[5] IBM, Node.js @ ibm, https://www.developer.ibm.com/node/, 2016.
[6] Netflix, Node.js in flames, http://www.techblog.netflix.com/2014/11/nodejs-in-flames.html, 2014.
[7] PayPal, Paypal node sdk, http://www.paypal.github.io/paypal-node-sdk/, 2016.
[8] Joyent Inc., Node.js at walmart: introduction, https://www.joyent.com/developers/videos/node-js-at-walmart-introduction, 2016.
[9] P. Teixeira, *Professional Node.js: Building Javascript Based Scalable Software*, John Wiley & Sons, Hoboken, NJ, USA, 2012.
[10] T. Capan, Why the hell would i use node. js? a case-by-case tutorial, 2015.
[11] Node Package Manager, Npm, https://www.npmjs.com/, 2016.
[12] Express, Express api, http://www.expressjs.com/en/4x/api.html, 2017.
[13] M. A. Jadhav, B. R. Sawant, and A. Deshmukh, "Single page application using angularjs," *International Journal of Computer Science and Information Technologies*, vol. 6, no. 3, pp. 2876–2879, 2015.
[14] N. Jain, P. Mangal, and D. Mehta, "Angularjs: a modern mvc framework in javascript," *Journal of Global Research in Computer Science*, vol. 5, no. 12, pp. 17–23, 2015.
[15] V. Balasubramanee, C. Wimalasena, R. Singh, and M. Pierce, "Twitter bootstrap and angularjs: frontend frameworks to expedite science gateway development," in *Proceedings of the*

2013 IEEE International Conference on Cluster Computing (CLUSTER), p. 1, Indianapolis, IN, USA, September 2013.

[16] MongoDB, Mongo db and mysql compared, https://www.mongodb.com/compare/mongodb-mysql, 2016.

[17] H.-K. Ra, A. Salekin, H.-J. Yoon et al., "AsthmaGuide: an asthma monitoring and advice ecosystem," in *Proceedings of the 2016 IEEE Wireless Health*, pp. 128–135, Charlottesville, VA, USA, October 2016.

[18] H.-K. Ra, S. Jeong, H. J. Yoon, and S. H. Son, "SHAF: framework for smart home sensing and actuation," in *Proceedings of the 2016 IEEE 22nd International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, p. 258, Daegu, Republic of Korea, August 2016.

[19] Sensorcon, Your smartphone can do much more with sensordrone, http://www.sensorcon.com/sensordrone, 2014.

[20] Y. Jiang, X. Liu, and S. Lian, "Design and implementation of smart-home monitoring system with the internet of things technology," in *Lecture Notes in Electrical Engineering*, pp. 473–484, Springer, Berlin, Germany, 2016.

[21] H. Lee, H. Ahn, S. Choi, and W. Choi, "The sams: smartphone addiction management system and verification," *Journal of Medical Systems*, vol. 38, no. 1, pp. 1–10, 2014.

[22] T. Steiner, S. Van Hooland, and E. Summers, "Mj no more: using concurrent wikipedia edit spikes with social network plausibility checks for breaking news detection," in *Proceedings of the 22nd International Conference on World Wide Web companion*, pp. 791–794, Rio de Janeiro, Brazil, May 2013.

[23] I. K. Chaniotis, K.-I. D. Kyriakou, and N. D. Tselikas, "Proximity: a real-time, location aware social web application built with node.js and angularjs," in *Proceedings of the Mobile Web Information Systems: 10th International Conference, MobiWIS 2013*, Paphos, Cyprus, August 2013.

[24] S. K. Badam and N. Elmqvist, "Polychrome: a cross-device framework for collaborative web visualization," in *Proceedings of the Ninth ACM International Conference on Interactive Tabletops and Surfaces, ITS'14*, pp. 109–118, New York, NY, USA, September 2014.

[25] T.-M. Grønli, G. Ghinea, and M. Younas, "A lightweight architecture for the web-of-things," in *Mobile Web Information Systems*, pp. 248–259, Springer, Berlin, Germany, 2013.

[26] J. Kim, E. Levy, A. Ferbrache et al., "MAGI: a Node. js web service for fast microRNA-Seq analysis in a GPU infrastructure," *Bioinformatics*, vol. 30, no. 19, pp. 2826-2827, 2014.

[27] T. Di Domenico, E. Potenza, I. Walsh et al., "Repeatsdb: a database of tandem repeat protein structures," *Nucleic Acids Research*, vol. 42, no. D1, pp. D352–D357, 2013.

[28] A. Nurminen, J. Järvi, and M. Lehtonen, *A Mixed Reality Interface for Real Time Tracked Public Transportation*, Helsinki Institute for Information Technology (HIIT), of Aalto University and University of Helsinki, Helsinki, Finland, 2014.

[29] K.-L. Wang, Y.-M. Hsieh, C.-N. Liu et al., "Using motion sensor for landslide monitoring and hazard mitigation," in *Intelligent Environmental Sensing*, pp. 111–127, Springer, Berlin, Germany, 2015.

[30] S. Frees, "A place for Node.js in the computer science curriculum," *Journal of Computing Sciences in Colleges*, vol. 30, no. 3, pp. 84–91, 2015.

[31] A. Ojamaa and K. Düüna, "Security assessment of Node.js platform," in *Proceedings of the Information Systems Security: 8th International Conference, ICISS 2012*, Guwahati, India, December 2012.

[32] A. Mardan, "Publishing Node.js modules and contributing to open source," in *Practical Node.js*, pp. 261–267, Springer, Berlin, Germany, 2014.

[33] G. Rauch, *Smashing Node.js: JavaScript Everywhere*, John Wiley & Sons, Hoboken, NJ, USA, 2012.

[34] J. R. Wilson, *Node.js the Right Way*, Pragmatic Programmers, Dallas, TX, USA, 2014.

[35] C. Gackenheimer, *Node. js Recipes: A Problem-Solution Approach*, Apress, New York, NY, USA, 2013.

[36] C. J. Ihrig, *Pro Node.js for Developers*, Apress, New York, NY, USA, 2013.

[37] M. Thompson, *Getting Started with GEO, CouchDB, and Node.js*, O'Reilly Media Inc., Newton, MA, USA, 2011.

[38] S. Pasquali, *Mastering Node.js*, Packt Publishing Ltd., Birmingham, UK, 2013.

[39] TutorialsPoint, Node.js-express framework, http://www.tutorialspoint.com/nodejs/nodejs_express_framework.htm, 2016.

[40] M. Kiessling, Node beinner book, http://www.nodebeginner.org/, 2016.