Homework 3

Mohammed Algadhib
algadhim@oregonstate.edu

Riley Kraft
kraftme@oregonstate.edu

Diego Saer
saerd@oregonstate.edu

Johnny Chan
chanjoh@oregonstate.edu

1. HIGH-LEVEL ARCHITECTURES

1.1 CLIENT-SERVER

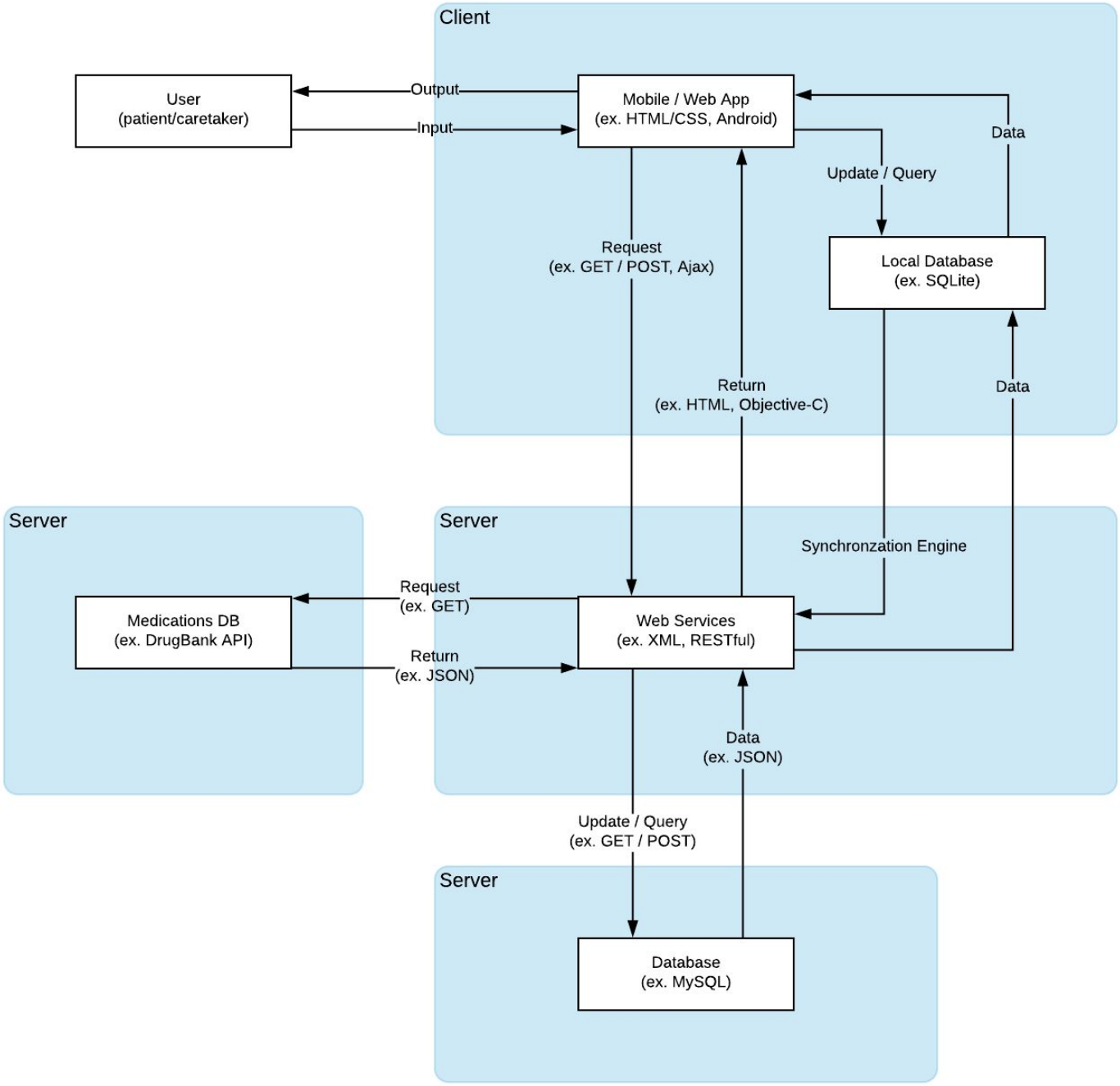
The following diagram depicts a client-server architecture for the software application, which outlines the tasks and workloads between the user and the client, the client and the web server, and the web server and supporting servers. In this architecture, it is important to note that the client does not share any of its resources with the web server. Instead, the client requests data and/or functions from the web server, which in turn makes requests to the supporting servers, and returns data to the client.

This architecture was also designed with a hybrid application in mind. A hybrid application is one in which one architecture supports both a web application interface as well as a native application interface, facilitating cross-platform support. In order to implement a hybrid, provide seamless operation during sporadic connectivity and maintain the integrity of the data, it was identified that this architecture would have to implement a local datastore and synchronization web service. The client should NOT be able to change data on the database server. Rather, the client requests synchronization from the web server at regular intervals, usually about every 1 minute. The web server initiates its synchronization engine to get data from the client's local datastore, synchronize it to the database server, which may result in initiating other web services and creating more data, then finally synchronizing the user's database on the server to the local database on the user's client.

The client talks to the web server through requests to various web services. The client essentially handles the presentation of the application to the user. This is where the engineering team would create the browser pages and mobile app interfaces using HTML, CSS, Javascript, Java, Objective-C, etc. However, since we are implementing a hybrid application, the mobile app would also feature the local datastore in something like SQLite. To use the web browser interface the user would have to have steady connectivity and could request data through the web server to the database server, thus there is no need for a local datastore for the web browser interface.

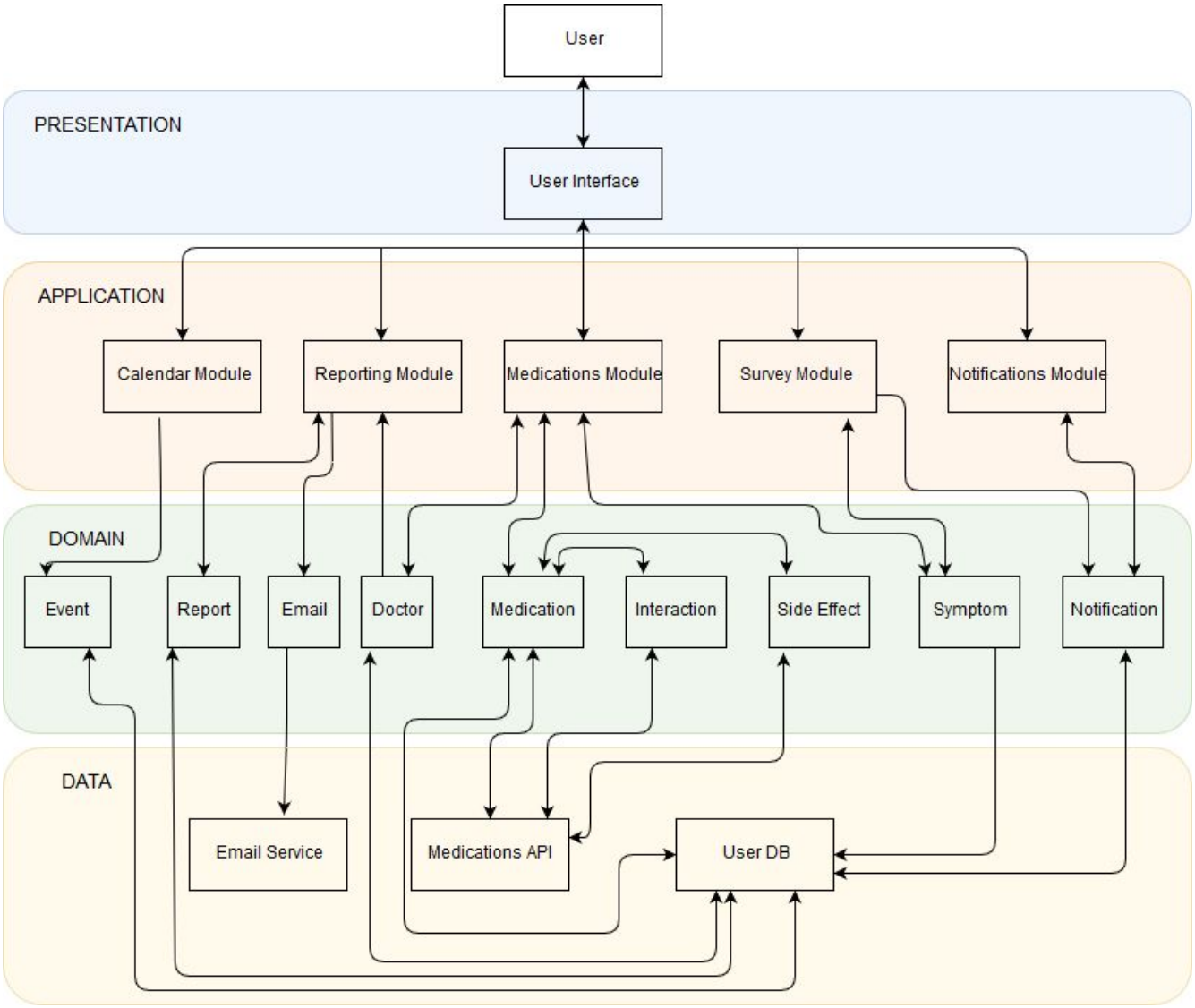
The web server processes all the client's requests and, in turn, creates requests of its own to other servers, then syncs data to the client. Requests such as GET and POST from the client are processed through the different web services. For instance, if the requests from the client requires data from the Medications DB then the web server would be responsible for creating the request to the Medication DB API. The web server would also process the returned data to be stored in the database server, then the database server would be synced to the client's local datastore. For the web browser interface, requests from the client for another web page would be sent to the web server, processed, and returned to the client interface as the requested HTML web page.

The application database is held on its own server, or series of servers, to maintain the integrity of the users' data. Only requests from the software's web services can access and manipulate the data stored in the database servers. This way, clients cannot compromise the software application system, their own data, or other user's data.


Client
User
(patient/caretaker)
Output
Input
Mobile / Web App
(ex. HTML/CSS, Android)
Data
Update / Query
Request
(ex. GET / POST, Ajax)
Local Database
(ex. SQLite)
Return
(ex. HTML, Objective-C)
Data
Server
Server
Synchronzation Engine
Request
(ex. GET)
Medications DB
(ex. DrugBank API)
Return
(ex. JSON)
Web Services
(ex. XML, RESTful)
Data
(ex. JSON)
Update / Query
(ex. GET / POST)
Server
Database
(ex. MySQL)

1.2 LAYERED

The following diagram depicts the system with a layered architecture. The presentation layer takes care of making the software look appealing and easy to use. The application layer contains the code for organizing and carrying out the different functionalities of the software, like adding medications or viewing all the calendar events. The domain layer contains all the actual objects that hold important information, like an actual report object or a medication object. These are handled by the modules in the application layer above. The data layer at the bottom holds all external information stores that interact with the system through the internet.



2. KEY QUALITY ATTRIBUTES

The system proposed in this paper has the following quality attributes that must be satisfied to deliver the final software application. These quality attributes are reliability, efficiency, usability, maintainability, testability, flexibility, portability, reusability, interoperability, and integrity. Nonetheless, for this systems the key quality attributes that are significant for the success of the final deliverable are integrity, reliability, interoperability, and usability. These attributes are essential requirements for the system's architecture based on the following assessments.

2.1 Integrity

The system needs to be secure. This application contains private data related to the patient's health records, as a result the flag of securing the access to the applications and its databases rises. Health records are considered to be private data that could be valuable for some hackers. In the server-client architecture, the client doesn't have permission to change the data on the server's database, and is only permitted to request synchronization from the server to the local database. As a result, the data on the server maintains its integrity. In the layered system architecture, the data transferred to and from the data layer are encrypted to prevent any unauthorized access.

2.2 Reliability

Software contains all the required attributes to perform in assumed conditions. In the client-server architecture diagram, depictions of the clients, which would be our app users can send, receive, and store data from a database reliably through the client-server interface as long as the client and server have a connection with each other. The server is sending and getting requests to databases as expected. In the the layered architecture diagram, client-server is divided into layers that molds closely with the client-server interface and it contains bidirectional arrows that data are flowing to the user and servers.

2.3 Interoperability

Software is capable of communicating with other systems. Both architectural designs allows the user to communicate with the server when users make submissions or requests. As a result both architectures are designed to be able to communicate through the standardized requests to databases and API, consequently, the architectures can be easily integrate most databases and api to part of the system. Moreover, both systems could generate requests to SQL databases and APIs using GET requests and parsing JSON returns.

2.4 Usability

In both architectural designs, the requirements of the users are satisfied. Users can complete tasks such as filling out surveys, medications, and generating health reports as intended by the software. Each of the task are divided into its own modules as in the layered architecture diagram.

2.5 Efficiency

Software may seem efficient with both architectural designs, but it cannot be measured without the software being implemented. The designs have the required attributes for efficiency such as using AJAX, asynchronous calls to servers, and RESTful APIs. Nonetheless, in the server-client architecture the suggested frameworks are verified to be efficient.

2.6 Maintainability

Over time all systems require some modifications to match any security threats or close any bugs. As a result the system components should be able to be updated independent with minimal change to other system components. Both architectures proposed systems that are built by multiple components, as a result, they both support maintainability. For further comparison between both architectures, we find the server-client architecture has fewer total modules, as a result, the number of modules to modify are less, but, the change might be significant. On the other side, in the layered architecture, more components may get modified with smaller change.

2.7 Testability

The system is built by multiple software modules and their modules should be tested separately to verify that every module is producing the expected output. Moreover, these modules need to be tested after integration to verify the stability of the system and the interoperability of each component. To verify the testability in the most efficient manner automated testing techniques such as unit testing are required every time before building the system executable. In both architectures, the sub-modules could be tested as black box system and their output could be verified using automated unit tests.

2.8 Flexibility

The system proposed should be robust against unusual events. In the layered architecture diagram we find that the system is constructed from four

different layers, these layers could be treated as separate subsystems that could be evaluated for robustness and flexibility. On the other side, the server-client architecture is also constructed with 4 components, 3 servers and 1 client. Similar to the layered architecture each component is treated as a subsystem, however, in this architecture there are less interfaces between the modules, hence, the subsystems are less vulnerable to unusual events.

2.9 Portability

Both architectural designs are not specific to any one device. Both designs give the user the freedom to use practically any popular devices such as computers and smartphones. Users can access their data on any device anywhere in the world once an account has been created and data is connected. The server-client architecture proposes a hybrid implementation of the system, thus, the system will be cross-platform. In comparison, the layered architecture could also utilize a hybrid design of its layers. In addition, if a system doesn't support a specific hybrid layer efficiently, this layer could be changed with no major redesign of the architecture.

2.10 Reusability

Reusability goes hand in hand with portability. Both architectures are built upon smaller subsystems, layers in the layered architecture and server or client in the server-client architecture, which makes each subsystem usable for different applications. However, in the layered architecture there are more modules that could be utilized in other systems.

3. FAILURE MODES

3.1 FAILURE TO SYNC DATABASE

The fault tree below shows various ways in which data fails to sync between the client's local datastore and the database server. For example, say the user just recorded a new prescription medication into their mobile application. After the new prescription is added, the software triggers a request to a Medications DB API. The API returns all the known side effects of this new medication and all known interactions with the patient's other medications. As defined earlier in the architectures, the mobile app would: 1) save the new prescription data to its local datastore, 2) request the web server to sync the local datastore with the database server, 3) the web server requests medication symptoms and interactions from the Medications DB API, 4) the web server records any data returned from the API, 5) then syncs the database server with the client's local datastore.

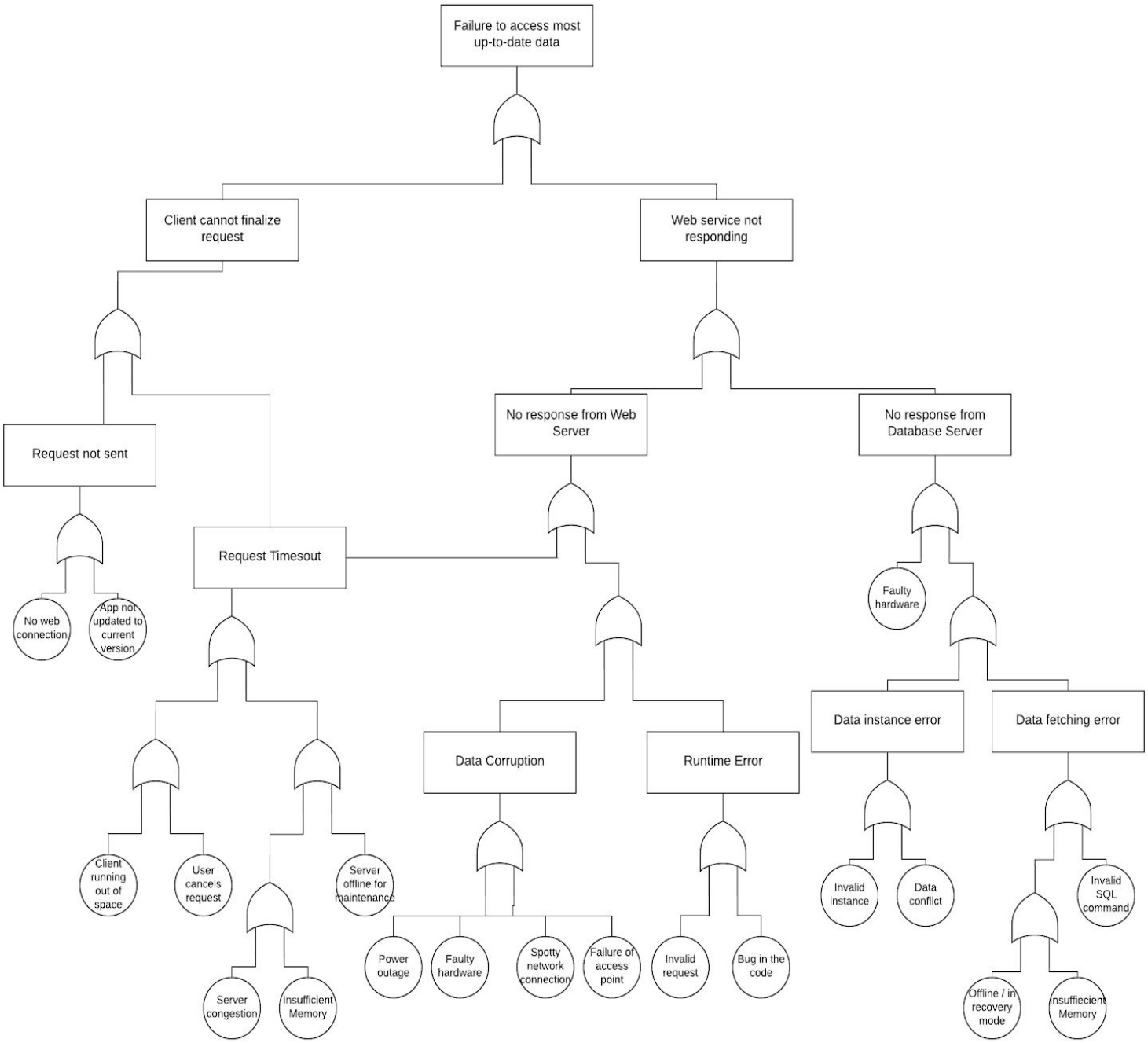
Among these various requests between the client and web server, web server and Medications API and web server and database server, several things could go wrong resulting in the failure to sync the database server with the client's local datastore. This could leave the user with a record of a new prescription but no information on potentially dangerous side effects or interactions with their other medications. The central objective of this software application is to reduce the statistics on elderly patients having dangerous reactions to their various medications. A failure like this would defeat the purpose of engineering this software.

In regards to the two architectures, let's address the fundamental differences in their designs. The layered architecture is a logical structure organizing the various modules of the application and the relationships among those modules. The intent of this architecture is to show the control of data and functionality move from one layer of processing to another in a stack-like fashion. The client-server architecture is a physical structure organizing the various components which request services and return data between each other. The intent of this architecture is to show the bidirectional relationships between the various software and hardware components that make up the network of this software application.

Based on the identified basic events that could trigger this failure, the layered architecture appears to be lacking in the detail to plan for the mitigation of many of these events. In addition, several connections between the layers would each have to account for several of the same failure mode events. For example, each of the 6 connections to/from the User DB in the Data layer would have to account for: invalid SQL commands, DB offline, insufficient memory, invalid instances, server congestion, power outage, and faulty hardware. While the layers do provide a way to organize the various logical components of the software system into manageable modules, the layers do not appropriately organize the requests and returns between the various hardware components.

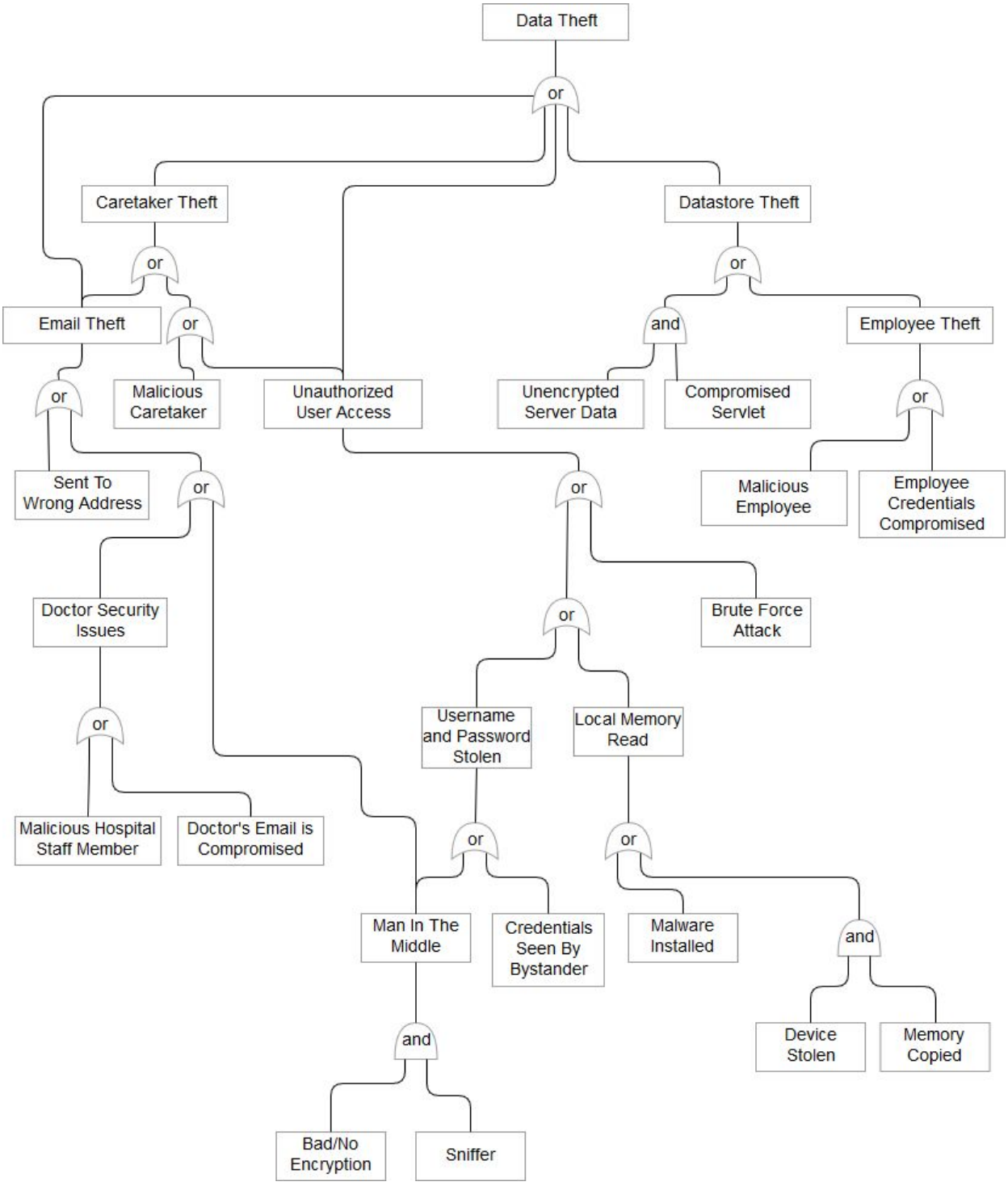
The client-server architecture divides the components of the system to potentially suggest both the software and hardware needs of this application system, making it easier to devise where failures can happen and why. Whereas, the layered architecture defines the logic of the transfer of information among the various application

processes. Considering several of the potential failure events for this case deal with faulty hardware, power outages or unreliable connectivity, we need an architecture that addresses the organization of physical components as well as logical ones. In this case, the client-server architecture would be the better choice.



3.2 DATA THEFT

The figure Below is a fault tree that shows the possible ways user data theft could occur.



There are 15 leaves to the fault tree above corresponding to the events that could potentially cause a data theft incident. 6 Of these events will only cause a fault if they are paired with another event. This leaves only 12 situations which could cause a failure. The Table below lists these situations, whether software can prevent their occurrence, and the software solutions needed to prevent them.

Events That Lead to Failure	Can Software Prevent Failure?	Client Server Architecture Solutions	Layered Architecture Solutions
Report Sent To Wrong Address	No	n/a	n/a
Malicious Hospital Staff Member	No	n/a	n/a
Doctor's Email Is Compromised	No	n/a	n/a
Malicious Caretaker	No	n/a	n/a
Bad/No Encryption When Communicating With Datastore + Sniffer	Yes	Encryption Needed For The Following: 1) App <-> Web Services 2) Local DB <-> Web Services 3) DB <-> Web Services	Encryption Needed For The Following: 1) Notifications Obj. <-> DB 2) Symptoms Obj. <-> DB 3) Medication Obj. <-> DB 4) Doctor Obj. <-> DB 5) Report Obj. <-> DB 6) Event Obj. <-> DB
Login Credentials Seen By Bystander	No	n/a	n/a
Device Is Infected By Malware	Yes	1) Local DB Needs To Be Encrypted 2) Malware Detection Needs To be implemented On Device To Prevent New Data Being Read	1) All 9 Domain Objects Need To Be Encrypted 2) Malware Detection Needs To be implemented On Device To Prevent New Data Being Read
Device Is Stolen + Memory Copied	Yes	Local DB Needs To Be Encrypted	All 9 Domain Objects Need To Be Encrypted
Log In Through Brute Force Attack	Yes	Need To Limit Login Attempts	Neet To Limit Login Attempts
Malicious Employee With Access To Datastore	No	n/a	n/a
Employee With Access To Datastore Has Credentials Compromised	No	n/a	n/a
Server Data Badly/Not Encrypted + Servlet Compromised	Yes	Need To Encrypt Server Data With Robust Encryption	Need To Encrypt Server Data With Robust Encryption

The table above shows that for all the data theft causing events that software can fix, both architectures can deal with them. The difference is that the implementation would be more complicated in the layered architecture. If there were bad or no encryption used when communicating with the datastore, the client-server architecture needs to make sure 3 components send encrypted information while the layered architecture needs to ensure 6 components send encrypted information. A similar case appears when the device is infected by malware, which could potentially send sensitive information to malicious groups or individuals, or when the device is stolen and the memory is read as the client-server architecture only needs to secure the local database while the layered architecture needs to secure each of the 9 types of objects that house sensitive information.

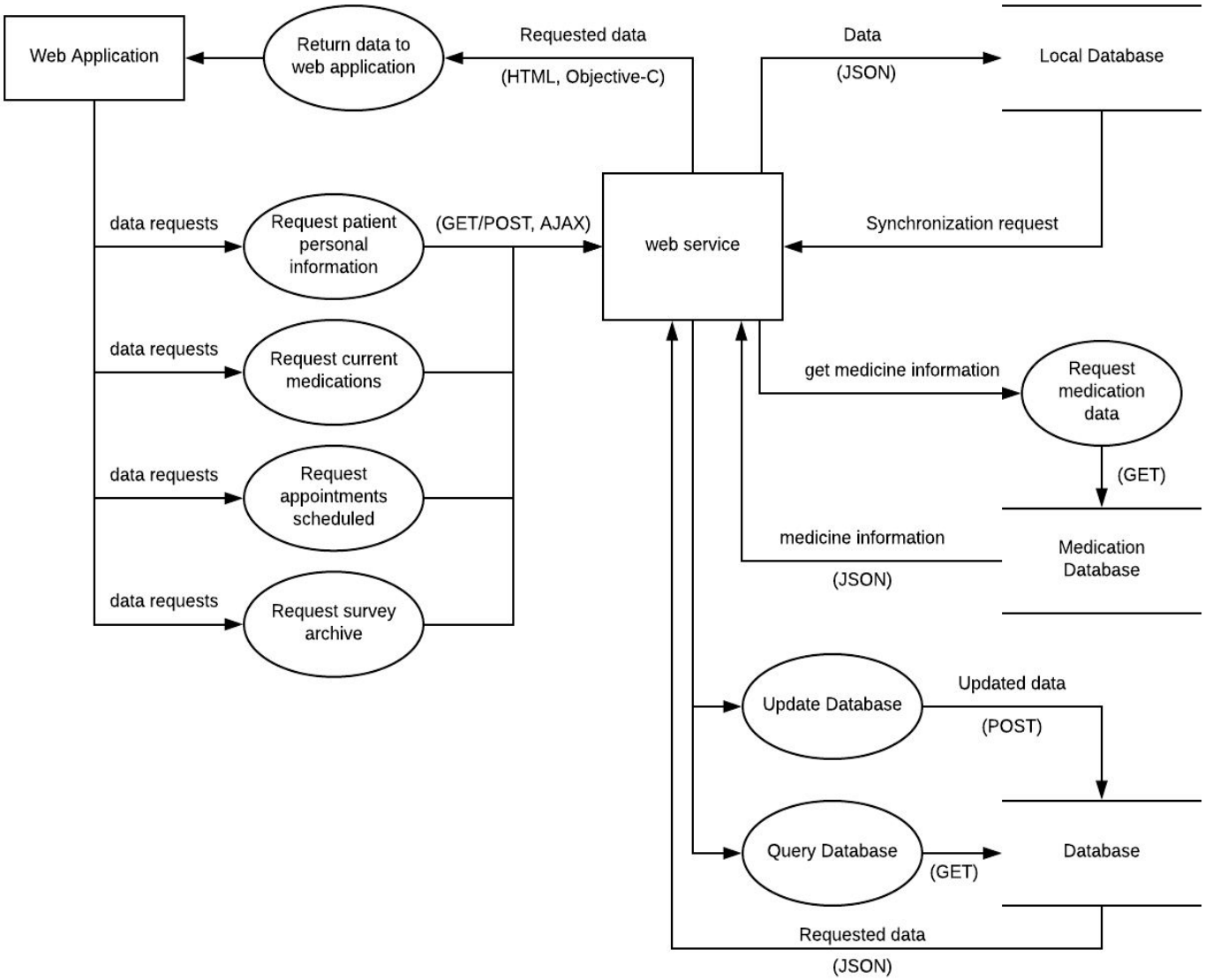
4. ARCHITECTURE DECOMPOSITION

Decomposition is achievable in both architectures, however the implementation of it may change in complexity. In the client-server, the security approach to prevent the user from modifying the data on the remote server is an effective solution for data theft and keeping the client synchronized with the server. Moreover, what makes the client-server architecture more secure is that in its modules there are less interfaces between its components, thus there are less components to secure, and it is easier to integrate with different systems. This results in the reduction in the number of failure points in the system.

4.1 WEB SERVER

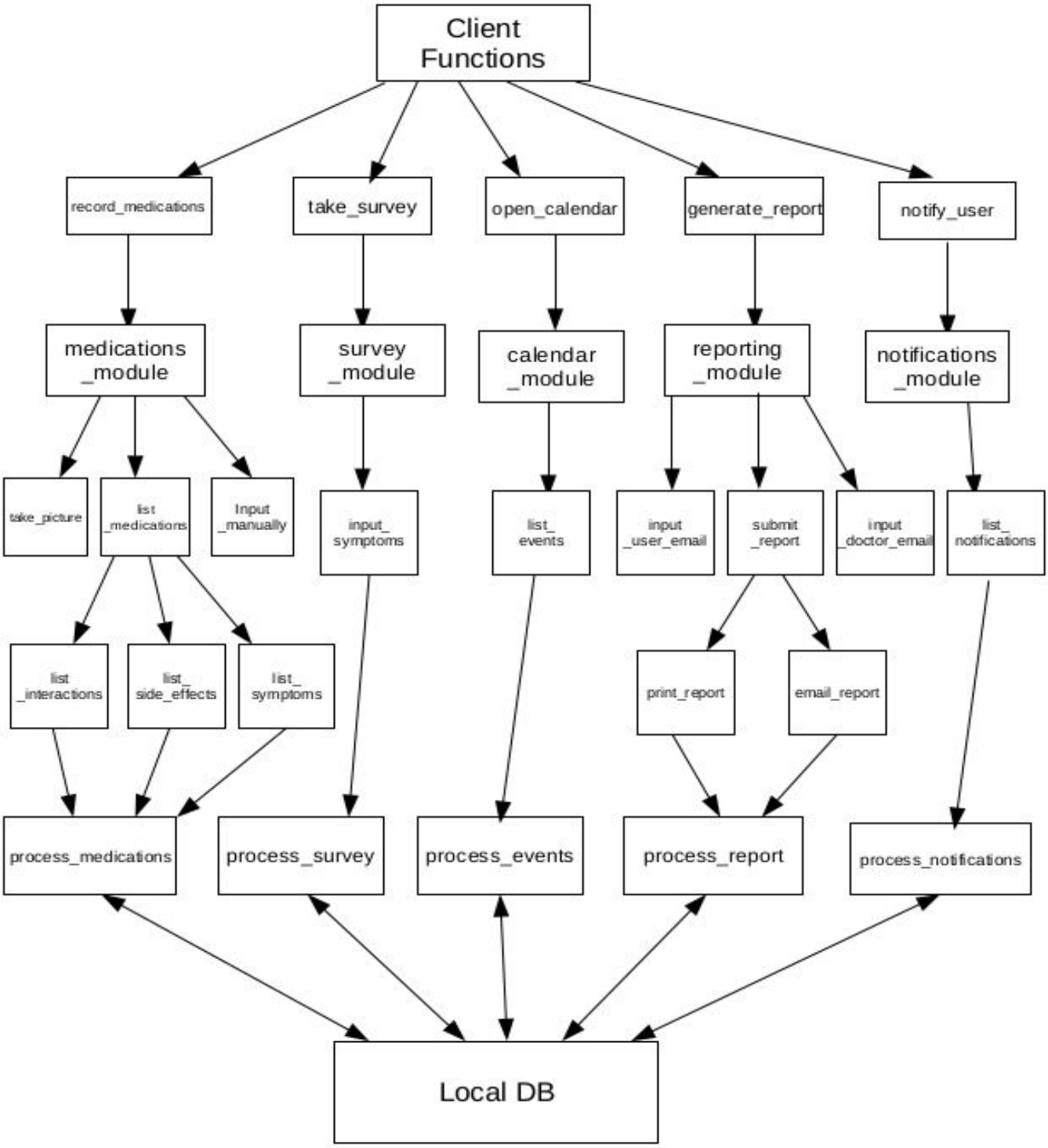
The web service module of the system is decomposed by processes. The main functionality of this component is to perform specific processes of w requests and returns between other components. The client application sends requests to the web server, which facilitates the various web services. Such requests include asking for: medication side effects and interactions from the Medications DB API; returning records from the Database

Server; syncing data with a mobile client's local datastore; etc. Web service tasks are requesting from the remote database and API to present data to the user when using the client application. The processes required for a mobile client to store and retrieve data from its Local Database were omitted from this DFD as these process do not reside on the Web Server. See the Client function decomposition DFD above for this information.



4.2 CLIENT

Functional decomposition was to used to dissect the client component to gain a more in depth insight into the identity of the constituent components of our software. This type of decomposition diagram is very simple. Functions are divided by specific tasks and divided into sub-functions which includes modules of processing medications, surveys, generating reports, calendar, and notifications. Each of these modules are then divided into small steps that users interact with, and their data is stored safely at every step of the process on the local database. The local database is also accessed by the processes within the client device to retrieve said stored data.



5. ARCHITECTURE VALIDATION

These use cases call for a series of functions within the user interface. Since the architecture is meant to show pieces of the system and their relationships, with strong emphasis on the internals of the system, this evaluation will focus on the background stages required to enable these use cases.

5.1 RECORDING A PRESCRIPTION

From the HW2 document, events 1 - 43 describe the available functions the user has access to via the native mobile app interface on the client device. Here, the user navigates the screen, utilizing buttons, text entry and a camera, all situated on the client device. All data entered in the prescription form is stored in temporary memory in the local datastore on the client.

After the user selects the SUBMIT button at event 43 is when the servers come into play. This action triggers a request to the web server's synchronization engine to pull data from the local datastore's temporary memory and record the changes to the database server. After the database server has synched with the most-up-to date information, the web server requests the medication records from the database server and creates a GET request to send to the Medications DB API. The API then processes the request and returns any found records of side effects and/or interactions to the web server. The web server parses the response using JSON and records the new records in the database server. If any new records are interactions, then the web server also creates requests to add new notifications records for each interaction record returned from the API. Finally, the web server calls on the synchronization engine again, but this time to pull data from the database server and sync it to the client's local datastore.

The interface of the client is then prompted to switch to the Alerts repository, where the user can view any new notifications that were created on the web server, recorded to the database server, and synced to the client's local datastore.

One major aspect of this use case has not been addressed in the current version of our client-server architecture, and that is how to implement the translation of the prescription label into text for the prescription form. Optical Character Recognition (OCR) is a large undertaking and the decision to be made here is whether the engineering team should create the functionality within the native part of the app, create an API, or simply utilize an existing open source API, like Google's Tesseract OCR. The first option would not prompt a change in the current version of the architecture. However, using an API for this feature would call for an additional server and more component connections.

5.2 HEALTH SURVEY

From the HW2 document, events 1 - 20 describe the available functions the user has access to via the mobile / web app interface on the client. Like the previous use case, here is where the user navigates the screen, utilizing buttons and text entry, all situated on the client device. If the client is a mobile device, all data entered in the daily survey and symptoms are stored in temporary memory in the local datastore.If the client is a web browser, all the data entered sits in its respective fields until submitted as a GET or POST request to the web server.

After the user selects the SUBMIT buttons at events 9 and 20, the client either calls a request to the web server to record the data in the database if the client is a web browser, or calls a request to sync the local datastore to the database server via the web server synchronization engine if the client is a mobile device.

The client then requests the web server to query the severity of their symptoms, by which the web server queries the database server and parses the resulting records. Based on the severity of the return records, the web server returns a message to the client to be displayed on the survey confirmation interface at events 12 and 22. In addition, any records returned to the web server from the database with cautionary severity prompts the web server to create notification records which are sent to the database server to be recorded. Finally, the web server calls on the synchronization engine again, but this time to pull data from the database server and sync it to the client's local datastore.

The interface of the client is then prompted to switch to the Alerts repository, where the user can view any new notifications that were created on the web server, recorded to the database server, and synced to the client's local datastore. If the client is a web browser, there is no local datastore to sync with, thus the client merely requests the Alerts repository interface from the web server, which requests the records from the database server to be presented on the client.

5.3 REPORTING TO DOCTORS

Everyday, the client device queries the calendar data, in the local datastore if on a mobile device or through the web server to the database if on a web browser. If the query returns any records of a doctor's appointment within the next 24 - 48 hours, then the client sends a request to the web server to create a report.

When the web server receives a request for a report, the web server in turn sends a query request to the database server for records from the past 30 days. The web server parses and analyzes the records returned from the

database to create a PDF report. The web server then sends the PDF back the database server to be stored.

The web server then calls on the synchronization engine to pull data, including the newly recorded report, from the database server and sync it to the client's local datastore if on a mobile device. If using a web browser, the client merely requests the Report repository web page, which queries the report records through the web server to the database server and displays the returned records to the user via the client device interface.

From the client interface, the user selects the reports to be submitted through printing or email. If the user chooses email, the client device requests the email address of the doctor for which the patient has an appointment from the web server, which in turn requests the record from the database server, and is returned to the client.

What the current version of our client-server architecture does not cover are the components and connections to output a report through an email server or a printer client device.

6. ARCHITECTURE CONFORMANCE

To evaluate that the chosen architecture of the system is effective for the purpose of the project and satisfies the design principles, the quality attributes of the system must be satisfied. In the client-server architecture the system is structured by dividing it into multiple modules. These modules are treated as separate subsystems. From the high level architecture of the system we find that a couple of the modules need to be further broken down in the decomposition design. For instance, the Mobile/Web App module might need to be decomposed based on functionality into different tasks, like communicating with the local database, communicating with the web service, and communicating with the user.
Below we have a list of the 4 most important quality attributes along with the 2 most important minor quality attributes, along with a short description on how well the client-server architecture design serves them.

6.1 KEY QUALITY ATTRIBUTES

Does the architecture keep the user's data secure (Integrity)?

The division of the system into client and server components creates a series of "gates" by which data must be verified before it can affect the user, the system, or other users. From a software point of view, the user's data could be reasonably well protected if the following protocols are implemented:

- Encrypt all internet communications, i.e. encrypt all communications between the client and the servers, and between the servers themselves
- Encrypt all databases. This includes the local database on the user's device and the external databases.
- Recommend the user install anti malware software.
- Limit the login attempts to prevent against a brute force attack.

Outside a software perspective, a few recommendations should be made known to the user.

- To make sure the caretaker is trustworthy or to give them limited functionality.
- To print the report for the doctor instead of emailing it.
- To take care around bystanders who may try to take a peek at the user's login information.

Finally, the databases themselves must be managed by a reputable source so that the information contained in them are well protected.

Does the architecture address safeguards to make sure the system always operates as needed (Reliability)?

In the synchronization failure-mode, it was discussed that several connection, hardware and software features have the potential to affect the reliability of this system. Apart from heavy testing there is no way to be sure the system is reliable upon release. However, the division of components within the client-server architecture narrows down the count of failure sources. Implementing redundancies and emergency protocols, e.g. how the system should behave if it cannot sync with the external databases, should be implemented for each of the architecture's components.

Does the architecture offer ways to ensure components can communicate with other components effectively (Interoperability)?

Unless the user has a reliable internet connection on their device, not much can be done on the user's end to ensure communication with the web server. This is why the architecture features a local datastore that syncs through the web server to the database server are frequent intervals. This way, the user can access the most up-to-date information before their connection was lost.

For the servers themselves, which must be reliable to ensure proper interoperability, the databases and network equipment should have backup power and be located in secure areas. Also, the databases should be backed up regularly on redundant systems. If a third party is used to house them, then they should be reputable. The APIs should also be from reputable sources that have robust net infrastructures.

Each component in the system communicates through GET/POST or JSON, utilizing AJAX and SQL, which are all common and trusted web and component communication methods.

Is the system easy to use (Usability)?

The decomposition of the client component addresses the flow functionality for the client device. The grade of usability would depend on the user interface. The user interface paper prototypes would need to be implemented and tested on a sample of senior citizens.

6.2 MINOR QUALITY ATTRIBUTES

Can the system run on different platforms (Portability)?

The architecture contains few internal interfaces between its components. These interfaces use common standard protocol to initiate requests, which improves the portability of the system as some subsystems, like the external databases, could be changed in the future for scalability.

The system is intended to be used mainly as a smartphone or tablet app. In order to extend its portability, web browser versions are also implemented. The chosen architecture implements a hybrid implementation of the user interface which means that the system is cross-platform, which supports the portability attribute.

Can the system easily be tested for errors (Testability)?

The system is constructed from a couple of submodules that can be treated as black-boxes. Each sub module does not have a large variety of inputs and outputs, thus automated unit tests could be developed to test the system during and after the development phase to insure that the system is ready to be open to the public.

7. ARCHITECTURE MODIFICATIONS

As was mentioned in the validation of use case 1, the current client-server architecture does not address the need for an OCR. Implementing an OCR within the native app would be a large undertaking, perhaps just as large as creating the rest of the software. Not to mention the memory and processing cost of storing it on a client device. Thus, the most efficient choice may be to utilize an API. Again, creating this API from scratch may be just as complex as creating the rest of the software. Hence, using an existing open source API is perhaps the best solution in this case.

To implement the OCR API into our architecture, we would need to have another supporting Server in the diagram, much like the one for the Medications DB API. The Client would send a request to the Web Server to translate a prescription label. The Web Server in turn would send the GET/POST request to the OCR API. The API processes the image and returns the results to the Web Server to be parsed in JSON. The Web Server then responds to the Client to display the resulting data in the appropriate form fields.

As was mentioned in the validation of use case 3, the current client-server architecture does not address the need for an email service or a printer client. Neither of these components require the Client to send a request to our system Web Server since the data being sent is already on the Client device. In this case, the Client would send a request to the email or printer services already available on the Client device. The email service would then send a request to its outside email server. The printer service would then send the printing job to a connected printer client.

To implement this into our architecture, we would need to add an additional Server and an additional Client to the diagram. The new Server would represent the outside email server and the new Client would represent the printer client device. In addition, our user's Client device would need two subcomponents added, much like the Local Database. The new subcomponents would represent the Email Service and Printer Service available on the user's Client device. The mobile or web app would send a request to these services, which would in turn talk to the components outside our software system. The services on the user's Client device would only return confirmations that the data was sent to their respective outside components successfully (e.g. "your email was sent successfully").

In regards to the failure modes that were addressed, these have given us much to think about regarding the potential risks of our architecture and how to mitigate them. However, since our architecture is a "high-level" diagram, we haven't found that adding or altering anymore high-level components or connections would address these failures. In order to do this, we would need to create lower-level data flow diagrams for every component within our system, not to include the outsourced web services since we could not control these.

As was mentioned in our conformances above, to mitigate the failures from outside sources the team would have to do thorough research into the APIs which would support this project. Not just into the software and connectivity, but also into the hardware and supporting infrastructure. Covering these concerns would give us a good place to start in reducing the risk of losing connectivity due to outside sources, thus reducing the risk of poor interoperability and reduced reliability for our system.

Also as was mentioned in our failure modes and conformances above, to reduce the risk of failures we need to address each component and each connection between

our architecture components. Each component can run the risk of failure either by software or hardware. As well, each connection between components, including between the user and the client, runs the risk of a security failure. This team has found that the client-server architecture, after implementing the suggested modifications, is an appropriate method for dividing the components of this system. Each component comes with its own set of integrity, reliability and interoperability requirements, as well as its own set of failure risks. This way, the team can more easily isolate the needs of the system and the potential sources of failure. Furthermore, decomposing each component as they are currently divided provides a good way to divide the labor of the system and address the core requirements of each component. For example, the client must provide functionally the user can interact with easily, the web service must be the process go-between for the client and all other supporting servers, the database must organize data to be stored and retrieved with minimal duplicate information, etc. The client-server architecture serves the needs of our software design well and, with further decomposition of the components, it would also serve as a good foundation to plan the execution of implementing the system.

8. REFERENCES

[1] Coderwall. Jun 2017. *Simple Data Synchronization for Web & Mobile apps(working offline)*. [Online] https://coderwall.com/p/gt_rfa/simple-data-synchronisation-for-web-mobile-apps-working-offline

[2] DrugBank API. [Online] https://docs.drugbankplus.com/v1/#introduction

[3] Free OCR API. [Online] https://ocr.space/ocrapi

[4] Jha, Ajay Kumar. Feb 2007. *A Risk Catalog for Mobile Applications*. [Online] http://testingeducation.org/articles/AjayJha_Thesis.pdf

[5] Lucid Charts. *What is a Data Flow Diagram*. [Online] https://www.lucidchart.com/pages/data-flow-diagram