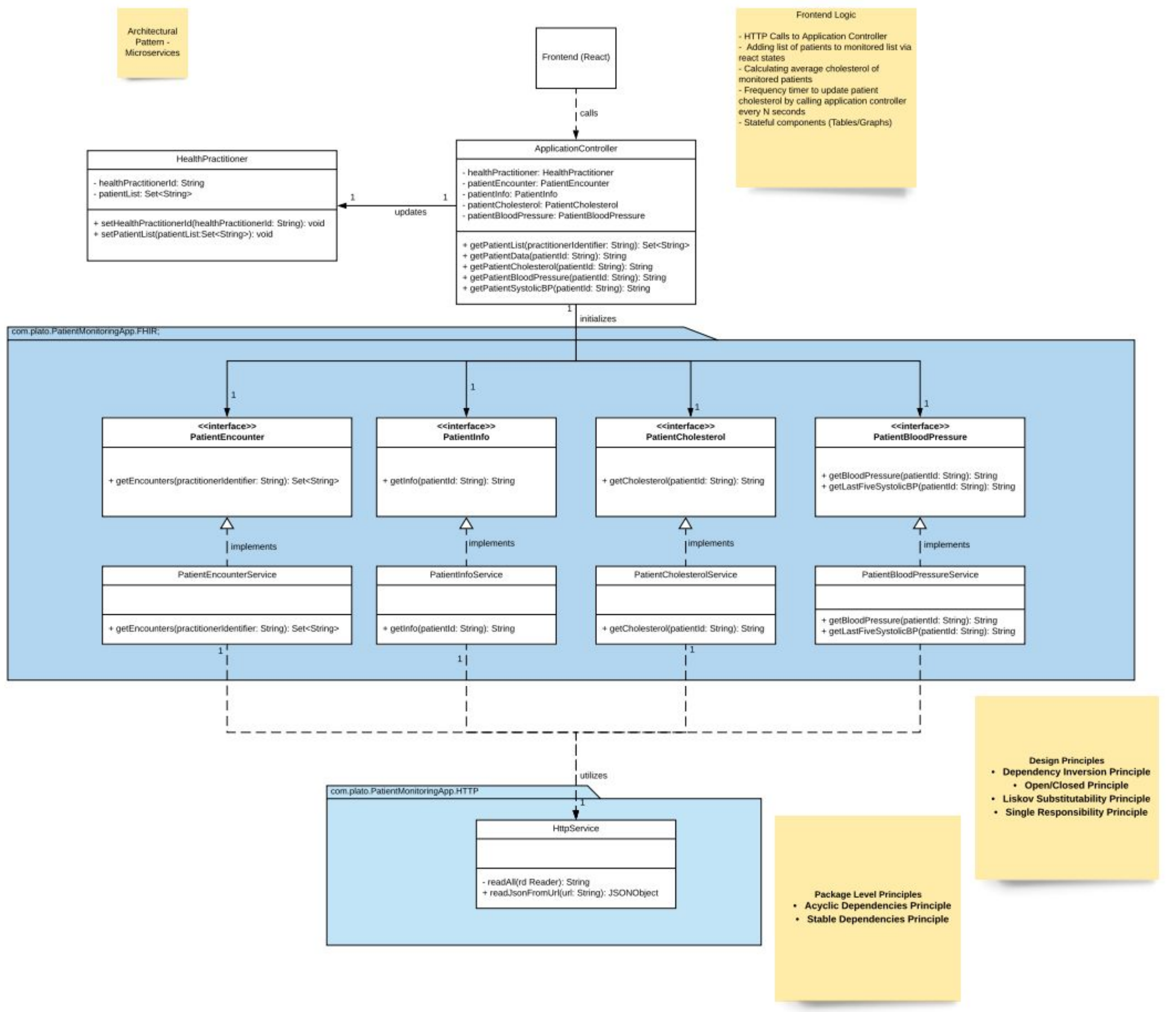
Architectural Pattern - Microservices
Frontend (React)
calls
Frontend Logic
- HTTP Calls to Application Controller
- Adding list of patients to monitored list via react states
- Calculating average cholesterol of monitored patients
- Frequency timer to update patient cholesterol by calling application controller every N seconds
- Stateful components (Tables/Graphs)
HealthPractitioner
- healthPractitionerId: String
- patientList: Set<String>
+ setHealthPractitionerId(healthPractitionerId: String): void
+ setPatientList(patientList:Set<String>): void
1
updates
1
ApplicationController
- healthPractitioner: HealthPractitioner
- patientEncounter: PatientEncounter
- patientInfo: PatientInfo
- patientCholesterol: PatientCholesterol
- patientBloodPressure: PatientBloodPressure
+ getPatientList(practitionerIdentifier: String): Set<String>
+ getPatientData(patientId: String): String
+ getPatientCholesterol(patientId: String): String
+ getPatientBloodPressure(patientId: String): String
+ getPatientSystolicBP(patientId: String): String
1
initializes
com.plato.PatientMonitoringApp.FHIR;
1
<<interface>> PatientEncounter
+ getEncounters(practitionerIdentifier: String): Set<String>
1
<<interface>> PatientInfo
+ getInfo(patientId: String): String
1
<<interface>> PatientCholesterol
+ getCholesterol(patientId: String): String
1
<<interface>> PatientBloodPressure
+ getBloodPressure(patientId: String): String
+ getLastFiveSystolicBP(patientId: String): String
implements
implements
implements
implements
PatientEncounterService
+ getEncounters(practitionerIdentifier: String): Set<String>
PatientInfoService
+ getInfo(patientId: String): String
PatientCholesterolService
+ getCholesterol(patientId: String): String
PatientBloodPressureService
+ getBloodPressure(patientId: String): String
+ getLastFiveSystolicBP(patientId: String): String
1
1
1
utilizes
com.plato.PatientMonitoringApp.HTTP
1
HttpService
- readAll(rd Reader): String
+ readJsonFromUrl(url: String): JSONObject
Design Principles
• Dependency Inversion Principle
• Open/Closed Principle
• Liskov Substitutability Principle
• Single Responsibility Principle
Package Level Principles
• Acyclic Dependencies Principle
• Stable Dependencies Principle

Please refer to Class Diagram File for clearer diagram

Introduction

Following the second assignment, the client wanted to add in new features to the application. The two main features that were to be added are new graph views to visualize the data and to extract blood pressure values from a patient.

Extending the system

Using our current architecture design, extending and adding new features to the application is very simple. Our original design utilizes hinge points which were created using the **Dependency Inversion Principle**[1]**.** When extending our system, we can clearly see the benefits of this. To add in a new feature, all we had to do was design a new interface(PatientBloodPressure) and implement it. All the other interfaces and services do not need to be modified, which demonstrates the usefulness of **Open/Closed Principle**[2] while adhering to **Liskov Substitutability Principle**[3]**.**

Design Principles

As stated before in the previous design rationale, some of the design principles that were adopted for this application were:

- **Dependency Inversion Principle**
- **Open/Closed Principle**
- **Liskov Substitutability Principle**

- **Single Responsibility Principle**[4]

We decided to use these three principles with the expectation that the system was going to be extended in the future. It provided hinge points which were easily extensible to add in new features. As for the single responsibility principle, we made sure all the new classes are serving a single concrete purpose so it is easier to find bugs in the code if needed.

Moving on, here are some of the package level design principles that we have applied in our original design:

- **Acyclic Dependencies Principle**[5]
- **Stable Dependencies Principle**[6]

While extending our system, we paid close attention to these principles so they are properly implemented. We made sure that any new classes/interfaces that are added to our packages do not form any cyclic dependencies. This is to prevent unnecessary dependencies which might cause future bugs. Following our previous design, the new classes implemented also depend on the direction of stability which is our HTTP package. As expected, the HTTP package is stable and did not change when adding new functionality.

Patterns

The application that we are developing is a web application and the architectural pattern that we have decided to adopt was the microservice approach. Following the **microservice architecture**[7], we separated our code into a collection of services which are independent and loosely coupled. Our frontend(React) makes it's client request to our backend(Spring Boot)'s application controller, which acts as an api layer. Here it calls the services that the frontend has requested.

The advantage of using this approach is that our components are distinct and separated. Development can be done for different services at the same time and will not disrupt the system. If this application were to be deployed in the future, we can deploy the frontend and backend separately. If the frontend needed a refactor, we don't need to deploy our backend.

React

With the decision of using react at the beginning, we are able to make use of stateful components. We could add new views such as graphs easily by using the same state as the tables. This is also resource friendly as calls to the server will only be made when the app is open and one call can be used to update several components. If another frontend framework were to be used, other design patterns such as **Observer**[8] could be useful in this scenario.

Refactoring [9]

During our development process, we constantly refactored to make sure that the design of the software remains healthy. When writing code, it is very easy to lose track and write a lot of code that might be redundant. We made sure to remove all unnecessary code and extract repeating code into a reusable function. Some of the classes had methods that did not belong to them so we moved them to another more appropriate class.

As for the frontend, originally we had a single page containing all the code which was very messy. We decided to refactor this to put each component into a small class to make the code more maintainable. We also renamed some of the methods to make it easier to understand.

References

[1]
FIT3077 Week 4 Lecture Principles of Object-Oriented Design 1
https://lms.monash.edu/pluginfile.php/10536249/mod_resource/content/2/OOP1.pdf
[2]
FIT3077 Week 4 Lecture Principles of Object-Oriented Design 1
https://lms.monash.edu/pluginfile.php/10536249/mod_resource/content/2/OOP1.pdf
[3]
FIT3077 Week 4 Lecture Principles of Object-Oriented Design 1
https://lms.monash.edu/pluginfile.php/10536249/mod_resource/content/2/OOP1.pdf
[4]
Dhurim Kelmendi - SOLID Principles made easy
https://medium.com/@dhkelmendi/solid-principles-made-easy-67b1246bcdf
[5]
FIT3077 Week 6 Lecture Design Patterns 2
https://lms.monash.edu/pluginfile.php/10625707/mod_resource/content/1/Design_Patterns_2.pdf
[6]
FIT3077 Week 7 Lecture Principles of Object-Oriented Design 2
https://lms.monash.edu/pluginfile.php/10668187/mod_resource/content/1/OOP2.pdf

[7]
FIT3077 Week 10 Software Architectural Patterns 1
https://d3cgwrxphz0fqu.cloudfront.net/aa/d2/aad2c5336a1fb647118ec64aae9eab8d9ba01ce2?response-content-disposition=inline%3Bfilename%3D%22FIT3077_ArchitecturalPatterns.pdf%22&response-content-type=application%2Fpdf
[8]
FIT3077 Week 5 Lecture Design Patterns 1
https://lms.monash.edu/pluginfile.php/10580885/mod_resource/content/1/Design_Patterns_1.pdf
[9]
FIT3077 Week 9 Refactoring
https://d3cgwrxphz0fqu.cloudfront.net/0a/9b/0a9b02a90c56f1f92ddd9bf583b4af70a8769699?response-content-disposition=inline%3Bfilename%3D%22Week%209%20-%20Refactoring.pdf%22&response-content-type=application%2Fpdf