WatTEST

Software Design Description Document

Manfred Cheung - 20577603

3-9-2018

Table of Contents

Table of Figures

Abstract

The document provides an overview of the design of WatTEST. WatTEST is a student response, or quizzing application built as an improvement on many of the existing applications currently available. Included are initial blueprints for the construction of the application, along with justifications for many of the design decisions which were made in the construction of the blueprint.

Introduction

Background

This report describes the design of a student response application, henceforth to be referred to using the working name, WatTEST. Within the education community, there is a growing desire to take advantage of the benefits of digital technology, in the drive to aid student learning. This can be seen in the vast array of applications allowing teachers to quickly gauge student comprehension of material, by giving short easy-to-take tests or quizzes. Examples of the most popular of such applications include solutions from iClicker, Top Hat, and Kahoot!. Note that this is not an exhaustive list, and in theory it should be possible to improvise quizzes to some extent with any of the polling software available such as Strawpoll, even if it is not specifically designed for educational use. However, when discussing existing solutions, only these three will be considered for brevity.

I personally feel there are issues with all these applications which make the existing choices less than ideal. Both Top Hat and iClicker require students to pay a fee to use their service. Top Hat asks for a $26 per term fee per student, while iClicker requires that the student either pay $42 for a new iClicker remote or acquire a used iClicker and pay a $7 used iClicker registration fee to the manufacturer in addition. [1] [2] [3]

Kahoot! on the other hand is free for classroom use which is commendable, although one should keep in mind though that this is only made possible by fact that educational users are subsidized by business Kahoot! users who must pay for the service. [4] Kahoot! also lacks features such as advanced question formats, verifiable student accounts, and score tracking over multiple rounds. Many of these deficiencies can be explained by the fact that these features would likely increase operational expenses beyond what Kahoot! will tolerate as result of increased database or server use.

Goals

In short, the existing applications are hampered by cost, either by charging students a premium for their use, or by restricting the features available. The goal of this application is to build a focused quizzing product which can be provided to students and teacher for little to no cost.

Scope

This application will be limited to functionality directly related to quizzing only. Any other functionality which do not fall within the scope will not be implemented regardless of usefulness to educators. At most, considerations will be made to allow modules possibly implementing these features to be easily

incorporated. This application will not attempt to replicate the multipurpose platforms which both iClicker and Top Hat provide.

Software Architecture: Overview

The exact system architecture or platform used by WatTEST is not finalized. However, what follows is a good approximation of the architecture based on the assumption that the resulting system will meet all requirements and implement the methods and algorithms listed later.

WatTEST will use client-server architecture, albeit an unusual implementation of it heavily inspired by the peer-to-peer (P2P) model. As result of this, the architecture itself is dynamic to a degree. The following will assume that the architecture is in steady state, with all clients connected to all necessary partners. When breaking this assumption to discuss connection brokering mechanisms for example, it will be stated explicitly.

System Architecture Overview

At the physical level, there are 4 key components of WatTEST; the static file server, the REST server, the teacher client, and the student client.

The static file server's job is to host the code used by the teacher client and the student client. This is in the form of HTML, JS, CSS, and image files transferred using HTTP/HTTPS. This server does not have to be separate from the REST server which is completely capable of serving static files but is designed to be as static files can be hosted alone for extremely low cost. Combining the static file server with the REST server can result in higher expenses due to increased network traffic.

The REST server provides two services. It must store classroom account information and act as a P2P connection broker. As such it will require a database for storage of account information. Data will be transmitted using JSON format.

Both the teacher and the student clients will require a connection to the REST server to broker a connection to the P2P network. Communication with the REST server will be done using HTTP/HTTPS to pass messages using JSON format. Once connected, teacher and student clients communicate using the WebSocket protocol. The REST server does not act as a proxy between clients.

The teacher client will act like a server in traditional client-server architecture. It is responsible for storing information relating to quizzes, quiz questions, past student scores and so on. It must also broadcast instructions to the student clients its connected to and process their responses.

The student client on the other hand, will act like a client in traditional client-server architecture. This client does not require persistent storage. Once connected to the P2P network, its only task is to act as an interface between the student and the teacher client.

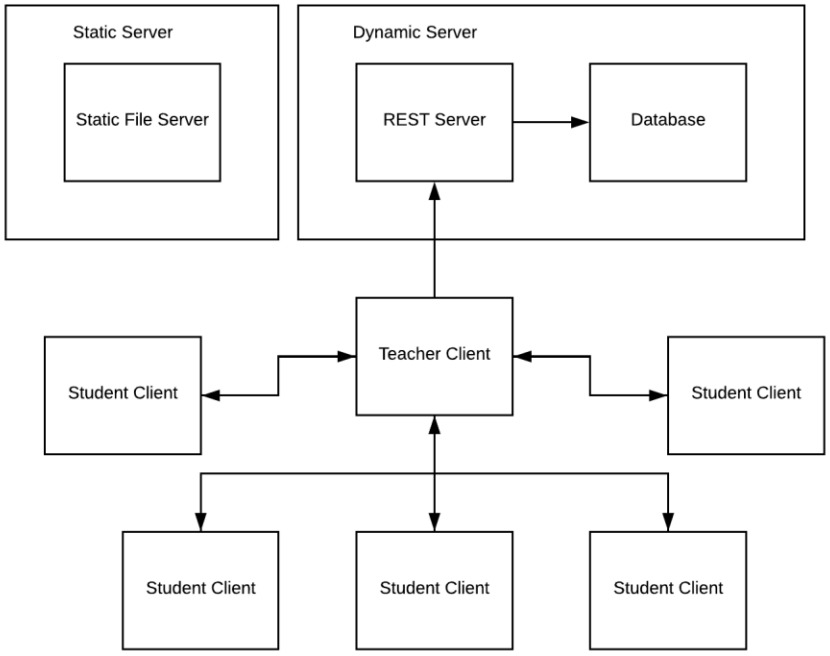

Figure 1:WatTEST Physical level architecture

Software Architecture Representation

The functions needed by WatTEST is now broken down further into major components. This decomposition was done with the goal of maximising cohesion within each module and minimising coupling between modules, which increases the maintainability of the system. The functions each of these components perform will be explained in greater detail below.

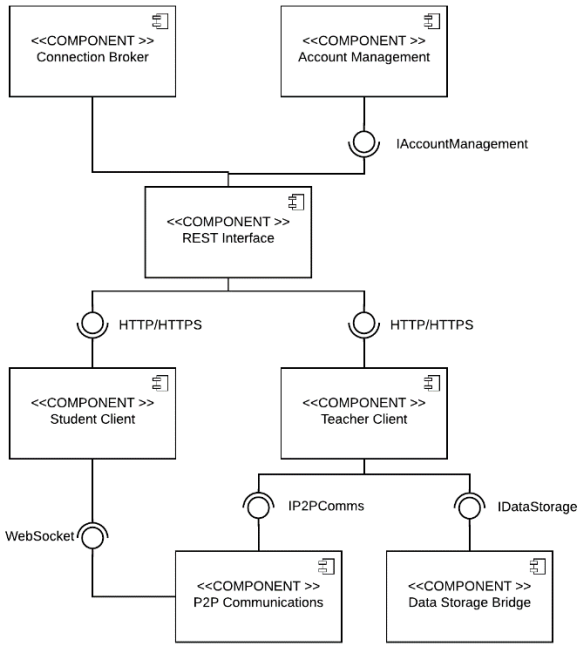

Figure 2: WatTEST Component diagram

Major Components

Connection Broker

The Connection Broker component is responsible for one task, which is routing student connection requests to the correct P2P network, therefore allowing it to join. Most of this work is performed by an external library imported into this component. This does not need an interface since it exposes no methods or attributes. However, it still needs to be initialized.

Account Management

The account management component handles all task related to classroom accounts. These include account creation, deletion, and updating. The type of information this component is expected to handle includes classroom ID, hashed and salted password, email address, and P2P network ID.

REST Interface

The REST interface as the name suggests provides an interface for all the server components. It does not perform any significant functions beyond this.

Student Client

The student client is only responsible for UI. It does not contain business logic only presentation logic and as such is essentially slaved to the teacher client. Therefore, the student client only contains the code needed to receive and send messages to the teacher client, and render the UI based on information received from the teacher client.

Teacher Client

The teacher client must by necessity support multiple pages, each with different functionality. However, multiple pages would perform poorly on a static server, thus the decision to build the teacher client as a single page application (SPA). The teacher client's only role is to serve as a router, switching between pages when appropriate. Each page contains both the presentation logic and business logic needed to build the UI and perform various tasks.

P2P Communications

The P2P communications component contains all the functions needed to interact with the student clients. These include a broadcast function and a receive event. In addition, validation of student logins falls to this component.

Data Storage Bridge

The data storage bridge is responsible for providing an interface for storage. The is necessary because it may be necessary to have several storage schemes, including but not limited to persistent browser storage, exported file storage, or even traditional database storage.

Interfaces

IAccountManagement

The IAccountManagement interface is provided by the account management component. It will be used by the REST interface component to perform create, read, update, and delete (CRUD) operations on the database.

IP2PComms

The IP2PComms interface is provided by the P2P Communications component. It will be used by the Teacher client to communicate with student clients.

IDataStorage

The IDataComms interface is provided by the Data Storage Bridge component. It will be used by the teacher client to perform CRUD operations on the storage medium of choice.

Architectural Goals and Constraints

What follows will be an explanation of the rationale behind the patterns used in this application. Discussion of the arrangement of components and interfaces will be included. As well, the effect of architectural decisions on quality aspects of the software will be covered.

Explanation of Architectural Decisions

The most notable architectural design choice made is the choice to implement an odd hybrid P2P / client-server architecture to handle communication between student and teacher clients. The largest reason for this is cost reduction. Performing work using centralized servers costs a lot of money. These servers must be capable of handling requests from all its clients, which involves sending and receiving network traffic, computation, and data storage. For a service properly supporting many clients, these costs can add up.

There were three architectures considered; traditional client-server, hybrid P2P / client-server, and full P2P. Traditional client-server means all clients regardless of type communicate via the server itself. As previously stated, this is expensive for various reasons. However, this architecture style is by far the easiest to implement.

The hybrid architecture involved offloading all the computation, data storage, and network load to the teacher client, which for all intents and purposes now acts as the central server for a single classroom. The central server now only handles connecting student clients to the teacher client. This architecture is more complex compared to traditional client-server, and thus suffers from the disadvantages of complexity; difficulty in coding and testing. In addition, it is unknown how many student clients a single teacher client can support while remaining performant, although this probably depends on the speed of the host computer and network connections.

Full P2P was also considered, which is where all student clients communicate with each other instead of communicating with the teacher client exclusively. This is even more complex compared to the hybrid architecture, however in theory it is possible. Under this system, all the student clients will still be slaved to the teacher client and the broadcasts from the teachers will still be used to disseminate information. However, responses from the student clients can be recorded using blockchain, eliminating the need for the teacher client to handle all the responses simultaneously. Theoretically such a system can be used to

support practically unlimited numbers of student clients under expected operating conditions. However, as most classrooms are limited in size anyway, the development costs of implementing such a system vastly outweighs the advantages.

Thus, the hybrid architecture was chosen to minimize the amount of work performed by the centralized server by shifting the work to the clients themselves, while at the same time keeping development costs reasonable. While unknown, it is reasonable to assume that this architecture can support all the students in a typical classroom, which is estimated to be between 1 – 100.

Architectural Constraints

Again, the exact architecture of the system is uncertain. The server for this application is intended to run on a free instance hosted by Heroku. However, if the application meets certain criteria, it could possibly be deployed on other servers, such as those belonging to the University of Waterloo as well. Thus, allowances must be made to ensure that any code is if not platform agnostic, then is platform tolerant. In this case, that involves ensuring all software frameworks are widely recognized and supported.

Due to operating cost concerns, it is assumed that the platform will allow the server minimal resources, which may include limited CPU time, database space, and inbound and outbound network traffic speed and bandwidth.

Also, it is assumed that the teacher client is run on average computer hardware. This implies that processing and network speed will not be outstanding.

Impact on Quality Aspects

Performance

The performance of the chosen hybrid architecture has already been discussed to some extent. However, there are a few things to add here. The first is a list of speed measurements which will either be important to user experience or are expected to be bottlenecks of the system. This list of measurements, in no particular order, is a good place to start when building benchmarks measuring performance.

- Connection brokering time
- DB response time
- Student response confirmation time

Scalability

The scalability of the chosen architecture has been discussed at length previously. In short, the goal was to support a maximum of 100 concurrent student clients per teacher client. It is reasonable to assume this goal will be met given what is known about the system. The capacity of the server itself is unknown as well but considering the architecture is designed to place as little load as possible on the server, it is more likely that the teacher client becomes a performance issue before the server does. However, we do expect that the load placed on the server to be bursty rather than uniform. This is due to a teacher's students all attempting to join the teacher client at the same time. This can be mitigated by intelligent caching to a large extent if it becomes an issue.

Security

The security of the system is much more difficult to manage due to its P2P inspired architecture. There are many potential attack vectors, some of which are listed below according to physical level component. This is in no way a complete list.

- Server
    - Denial of service (DOS)
    - Weak authentication
    - SQL injection
- Teacher Client
    - Denial of service (DOS)
    - Weak authentication
    - Cross Site Scripting
- Student Client
    - Cross Site Scripting

One should notice that the teacher client, as result of its role as both server and client, is now liable to attack vectors targeting both. This was deemed an acceptable trade-off. The risks can be mitigated by hardening the client against attack.

Modifiability

As discussed previously, various methods were used to ensure the code is modifiable. All boundaries on the physical and component level are loosely coupled to ensure changes in one section do not overtly affect another. Thus, modules can be added or removed with minimal effort.

Use Case View

REST Server View

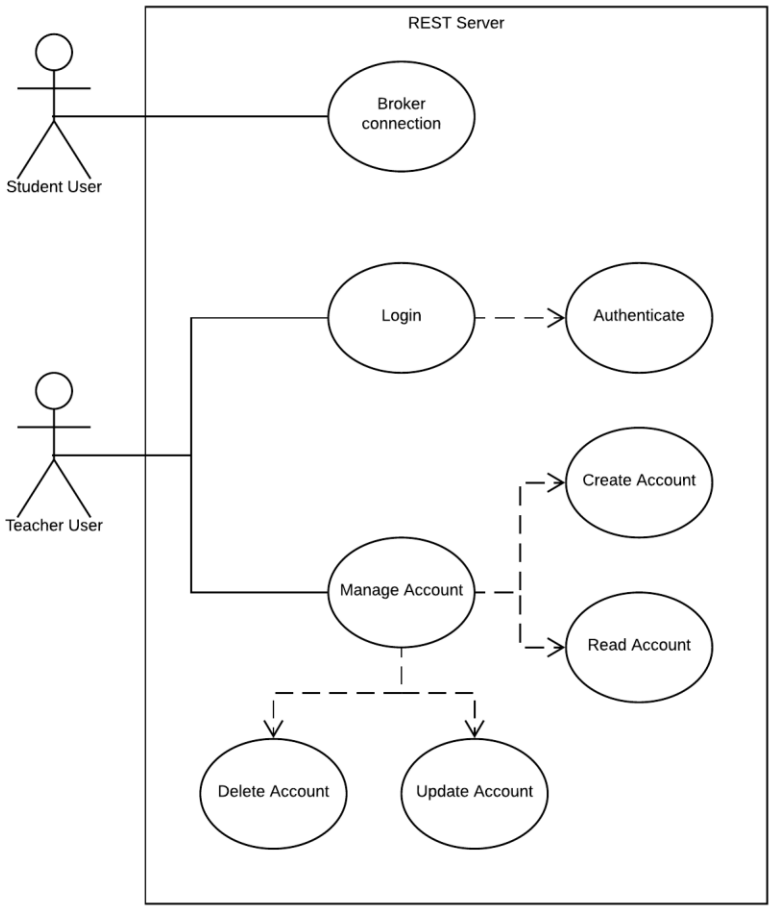

Figure 3: REST Server Use Case

Actor: Student User

Precondition: Student can access Internet, REST Server is online, teacher client hosting classroom is online

Use Case: ConnectToClassroom

- Navigate to student client address
- Enter classroom ID
- REST server retrieves P2P network ID identifying classroom
- Student client uses P2P network ID to join classroom

Actor: Teacher User

Precondition: Teacher can access Internet, REST Server is online

Use Case: TeacherCreateAccount

- Navigate to teacher client address
- Click on new classroom
- Fill out all required form fields
- Click enter
- Login using new account

Precondition: Teacher can access Internet, REST Server is online, teacher is already logged in

Use Case: TeacherReadAccount

- Click on account

Use Case: TeacherUpdateAccount

- Click on account
- Click on edit account
- Make necessary changes to account information
- Confirm changes
- Redirect to quiz manager page

Use Case: TeacherDeleteAccount

- Click on account
- Click on edit account
- Click on delete account
- Confirm changes
- Redirect to teacher login page

Teacher Client View

Actor: Teacher User

Precondition: Teacher can access Internet, REST Server is online

Use Case: LoginToClassroom

- Navigate to teacher client address
- Enter classroom ID and password
- REST server updates P2P network ID identifying classroom
- Redirect to quiz manager page

Precondition: Teacher can access Internet, REST Server is online, teacher is already logged in

Use Case: CreateQuiz

- Click on create quiz button in manage quiz page
- Creates a new quiz object and saves to storage
- Redirects to edit quiz page loading the newly created quiz

Use Case: EditQuiz

- Click on edit quiz button in manage quiz page having selected quiz to be edited
- Redirects to edit quiz page loading selected quiz
- Make desired changes to quiz
- Click save quiz button
- Quiz object is saved, replacing old quiz object
- Redirects to manage quiz page

Use Case: DeleteQuiz

- Click on delete quiz button in manage quiz page having selected quiz to be deleted
- Deletes quiz object in storage

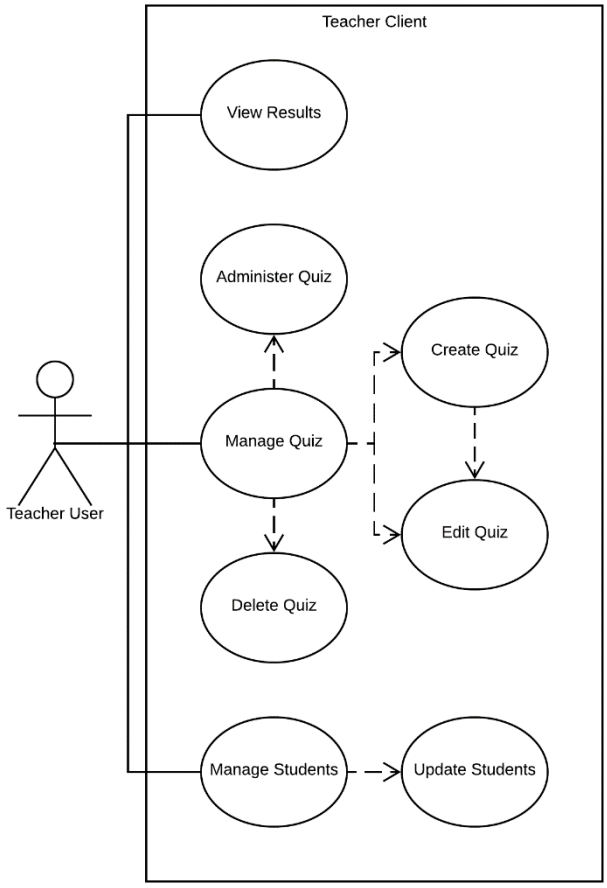

Figure 4: Teacher Client Use Case

Use Case: AdministerQuiz

- Click on start quiz button in manage quiz page
- Executes instructions specified in the quiz object
- Saves responses received from student clients in storage

Use Case: CreateQuiz

- Click on create quiz button in manage quiz page
- Creates a new quiz object and saves to storage
- Redirects to edit quiz page loading the newly created quiz

Use Case: ViewResults

- Click on view results link in manage quiz page
- Redirects to view results page loading student results

- Display student results

Use Case: UpdateStudents

- Click on students link in manage quiz page
- Redirects to manage students page loading student list
- Display student list
- Make desired changes to student list
- Click save student list button
- Student list object is saved, replacing old student list object
- Redirects to manage quiz page

Student Client View

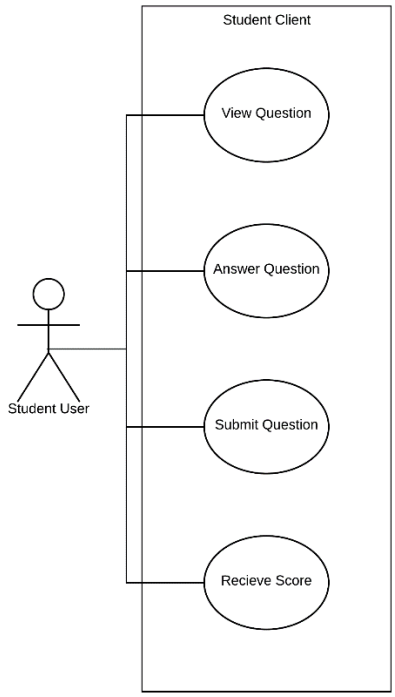

Figure 5: Student Use Case

Actor: Student User

Precondition: Student can access Internet, REST Server is online, teacher client is online, student client is connected to the teacher client, quiz has started

Use Case: AnswerQuizQuestion

- View the question
- Answer the question by manipulating UI elements
- Submit the response
- Wait for timer to finish
- Receive score on question

Deployment View

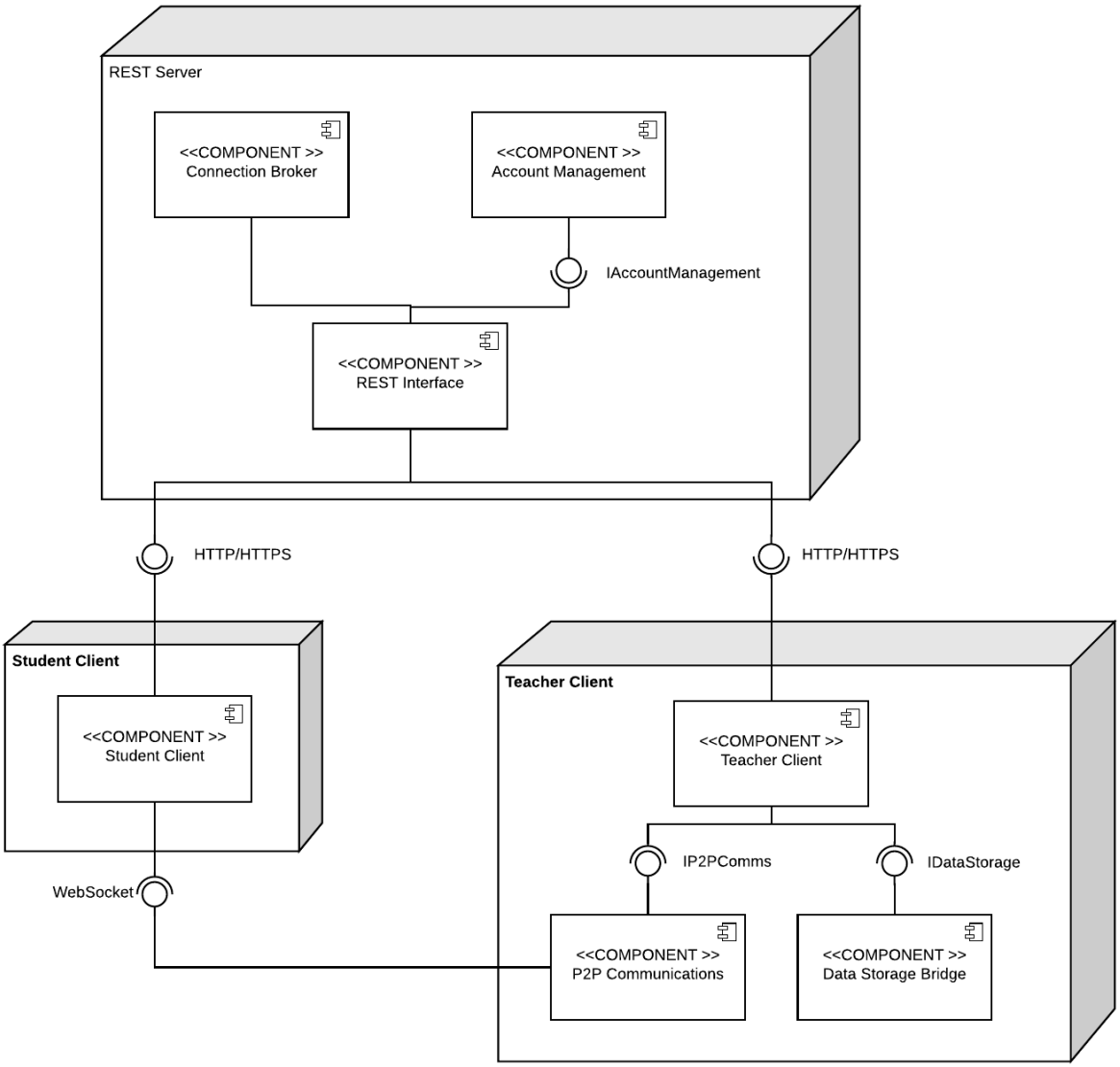

Figure 6: WatTEST Deployment Diagram

The diagram above maps components identified earlier to physical level components. All features in from the component diagram are preserved, including components and interfaces.

Identifying Modules and Control Flow

As discussed earlier, WatTEST has been broken down into components with the goal of increasing cohesion and decreasing coupling. The components used are as follows:

- Connection Broker
- Account Management
- REST Interface
- Student Client
- Teacher Client
- P2P Connections
- Data Storage Bridge

Interfaces are used for interactions between components. The non-standard interfaces are listed below:

- IAccountManagement
- IP2PComms
- IDataStorage

UML Class diagrams

UML class diagrams have been created for each physical level component. All subsystems inside each component are included.

Student Client Diagram

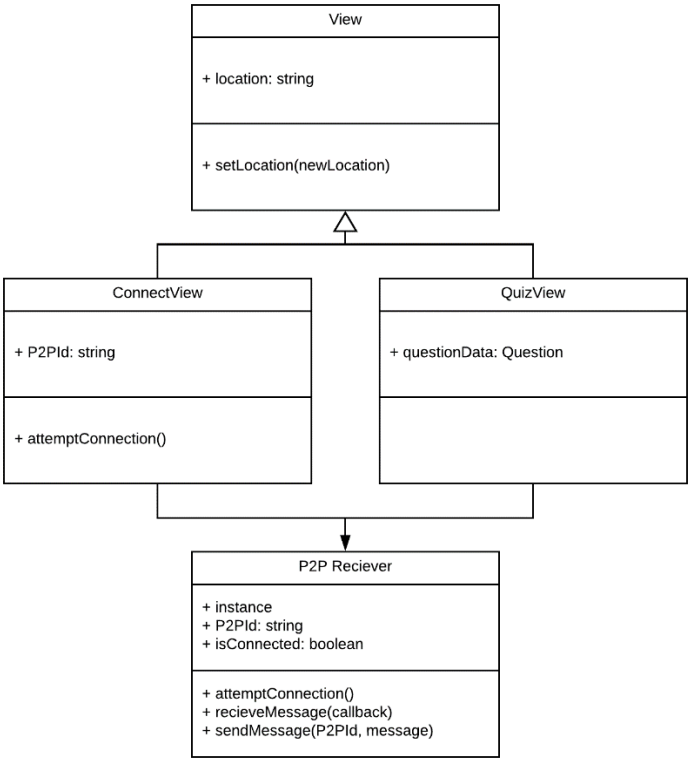

Figure 7: Student Client Class Diagram

REST Server Diagram

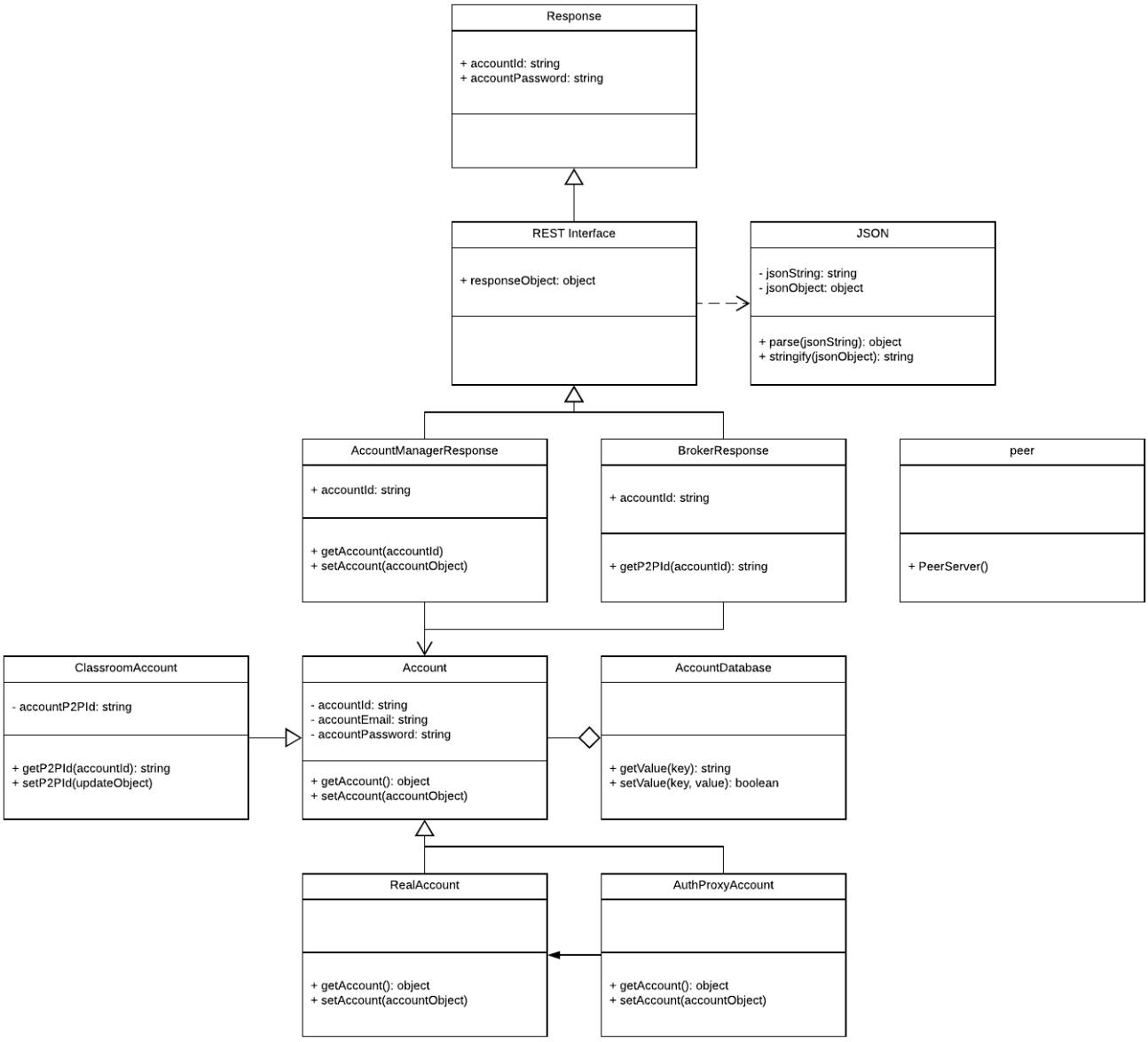

Figure 8: REST Server Class Diagram

Teacher Client Diagram

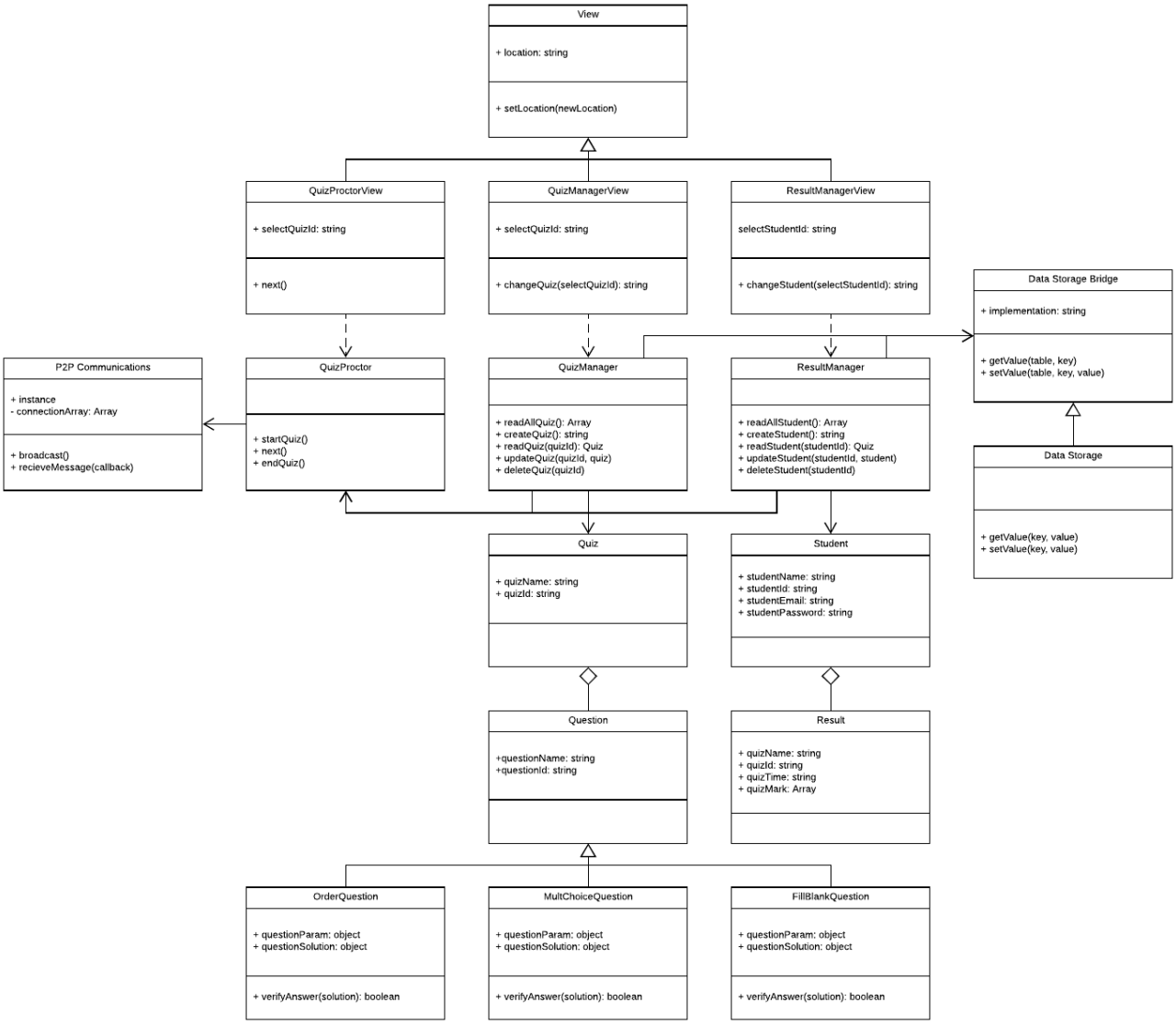

Figure 9: Teacher Client Class Diagram

Module identification rationale

The reason and the method behind module identification and encapsulation are very similar to what was done earlier for components. Again, the end goal is to increase cohesion within each module and decrease coupling between modules. This should increase the maintainability and modifiability of the system. In this case, the goal was for each module to be functionally cohesive. Also, we aimed for a maximum coupling level of stamp coupling, which is the most flexible level of coupling in terms of modifiability while at the same time keeping coupling acceptably low.

Control Flow Design

In all non-trivial modules, the application has a mostly decentralized control structure. Although controllers are heavily used, each module or even component maintains control over matters concerning its own function. Operations outside its own responsibility are passed to other modules, who maintain independence from the calling function. It is possible to build this completely decentralized without controllers of any kind. However, the resulting loss of readability is not worth it.

Interfaces

Listed are the interfaces designed for communicating within each node, along with the methods exposed.

IAccountManagement

IAccountManagement provides an interface consisting of only four methods; getAccount, setAccount, getP2PId, and setP2PId. No attributes are exposed. The reason for this is to keep the Account Management subsystem as independent as possible from the data which might be contained within the account. The loss of efficiency which might be caused by returning the entire account object is minimized by the small size of the account object and the unlikelihood of this function being called. An exception was made for getP2PId and setP2PId because these functions are likely to be called very often.

IP2PComms

IP2PComms consists of two methods; broadcast which takes an object as a parameter, and recieveMessage, which takes a function as a parameter.

IDataStorage

IDataStorage consists of two methods; getValue which takes a table and key string as parameters, and setValue, which takes a table and key string, and a value object as parameters.

WebSocket

The WebSocket interface essentially aims to permit the transfer of information through the use of callback functions. Essentially, if one end sends a message my passing an argument into a function, the other end receives the message through a callback function passed to a listener. Both the P2P Communications and the P2P Reciever modules in the teacher and student clients respectively use this interface. Only objects are to be sent through this interface. One example of use is on the student client side, where all received messages are parsed and used to update the state object in the view, and all sent messages contain an object specifying the question answer.

HTTP/HTTPS

Both protocols, which are equivalent as far as the modules are concerned, are used to communicate with the REST server through standard HTTP CRUD request methods. (PUT, GET, POST, DELETE) The REST Server always returns a valid JSON object containing the information requested or a status message.

Structural Design Specification

Implemented Design Patterns

There are multiple design patterns used throughout the application. However, we will concentrate on three of the most important.

Bridge Pattern

The Data Storage Bridge component implements the bridge pattern. This was done to decouple the abstraction of data storage from its implementation. The reason why this was done is because there are many possible data storage strategies which could be used on the front end with many reasons to use one over another. Data can be stored in the browser, which itself has inconsistent support for various technologies such as indexedDB, webSQL, and localStorage. Data can be stored in a file which can be downloaded, moved around, backed up, and loaded again. Or, although it was mentioned earlier that the REST server is designed to perform as few tasks as possible, it is possible to create a REST based storage system on it. This option can be considered as part of a monetization scheme for example. We can even consider implementing the strategy pattern to actively change data storage methods depending on need or configuration. The point is that it pays off to decouple data storage from its implementation, so various forms of storage can be used without changing code everywhere data storage is required.

Singleton Pattern

The P2P communications and P2P receiver component both implement the singleton pattern. These components hold an instance of an object managing either WebSocket connections from all student clients, or the WebSocket connection to the teacher client. If this instance is lost, then the brokering protocol must be completed all over again. Performing this operation more than once is obviously undesirable for a variety of reasons. Thus, the singleton pattern is used to ensure that one instance of the P2P communications component persists and is accessible from all modules which need it.

State Pattern

The QuizProctor component implements the state pattern. This component is responsible for advancing to the next stage of the quiz whenever the teacher pleases. This could mean placing the next question on the screen, or finishing the quiz, or revealing answers etc. The modules interacting with QuizProctor do not know what the next stage of the quiz is. However, we want to give other modules the ability to advance to the next stage of the quiz, regardless of what it might be. Hence, the state pattern is used to allow the interface to stay simple while permitting whatever complex behaviours next stage might hide.

Façade Pattern

The Façade pattern is used in multiple places throughout the application. Its purpose is to abstract a complex subsystem by providing a set of interfaces for it. This allows tasks to be performed by a subsystem without requiring knowledge of the specific operations or implementation of that subsystem.

This reduces coupling between the subsystem behind the façade and its users. For example, both QuizManager and ResultManager are examples of façades. Both facades hide the complexity behind performing CRUD operations on quizzes and student results respectively.

Important Modules for Control Flow

Module control flow shall be examined on a component-by-component level. Reference the class diagrams provided above.

REST Server

The controller in this component is the REST Interface class. This module receives HTTP requests, parses the requests into a usable format, initiates data processing, and finally returns the result. While this sounds like it could be centralized control, it isn't because the responsibility for data processing is passed to a dedicated module.

A proxy has been implemented to control access to the account methods.

Teacher Client

The client is set up as a series of interchangeable pages, where each page is composed of a view logic module and a business module. A router changes which module pair is active at any given time based on parameters it is given, which is usually the URL. Note that this pattern is widely used in SPAs.

The most important module in this component is the view module, which acts as the router in this case. Not only do all the page specific modules inherit from this module, the method it contains, setLocation, can change which view module is active, thus changing all behaviours.

The Data Storage and Data Storage Bridge Modules are also significant in terms of control flow. As previously discussed, the bridge provides a wrapper around the actual Data Storage implementation, decoupling the two. This was done so that multiple implementations could be used with minimal changes in code outside of these two modules. Another benefit is that it allows the implementation interface and the bridge interface to differ. In this case, I am aware that the Data Storage implementation I plan on using takes only key and value; it is a non-relational database. However, this is not ideal for the data I want to store. Therefore, with the bridge I can pass in both key, value, and table, and use logic in the bridge to allow the key-value store to handle it.

Student Client

The student client is quite simple, seeing as it is slaved to the teacher client and thus need not handle anything more complex than UI. As we can see, the router pattern appears again, with one view class and two specific view classes inheriting from it. This time, there are no business logic module dependencies attached to the view logic modules. That is because any business logic required is simple enough that hooking the P2P Receiver module to the view modules directly is easier.

Data Structure and Algorithms

There are several important data structures of note here. They are the quiz object and by extension the question object and its variations, and the student object and by extension the result object.

These data structures will be illustrated in the form of annotated JSON objects. Note that although the JSON specification does not support comments, the standard double slash comment will be used here to indicate the following is an explanation and not a feature of the data structure.

Quiz Object

```
{
	quizId: "de933fd", // Random string
	quizName: "My Quiz",
	quizQuestion: [
		{
			questionType: "multiple_choice",
			questionName: "First Question",
			questionContentFormat: "string",
			questionContent: "What has 4 legs?",
			questionChoices: [
				"Horse",
				"Bug",
				"Dolphin",
				"Shark"
			],
			questionAnswer: ["0"]
		},
		{
			questionType: "order",
			questionName: "Second Question",
			questionContentFormat: "markdown",
			questionContent: "Order from largest to smallest
				![alt text][logo]

				[logo]: https://github.com/adam-p/markdown-
				here/raw/master/src/common/images/icon48.png 'Image
				Text'",
			questionChoices: [
				"Dog",
				"Bug",
				"Dolphin",
				"Elephant"
			],
			questionAnswer: ["3", "2", "0", "1"]
		}
	]
}
```

Student Object

```
{
	studentId: "c93hswi", // Random string
	studentEmail: "email@place.com",
	studentName: "Bob Jones",
	// student password hashed and salted
	studentPassword: "c5e635ec235a51e89f6ed7d4857afe58663d54f5"
	studentResult: [
		{
			quizName: "My Quiz",
			quizId: "de933fd",
			quizTime: "1520643004",
			answerMark: [true, true, false, false, true]
		}
	]
}
```

Behavioural Design Specification

There are a few boundary control use cases which all parties must deal with as result of the nature of P2P networks. These use cases will be listed, illustrated, and explained.

Actor: Teacher User

Precondition: Teacher can access Internet, REST Server is online, teacher client is online

Use Case: GracefulShutDownTeacher

- Click log out in homepage
- Disconnect all connected student clients
- Delete P2PId on REST Server
- Show teacher client page

Precondition: Teacher can access Internet, REST Server is online, teacher client is online

Use Case: GracefulShutDownStudent

- Click log out in homepage
- Disconnect from teacher client
- Show student client page

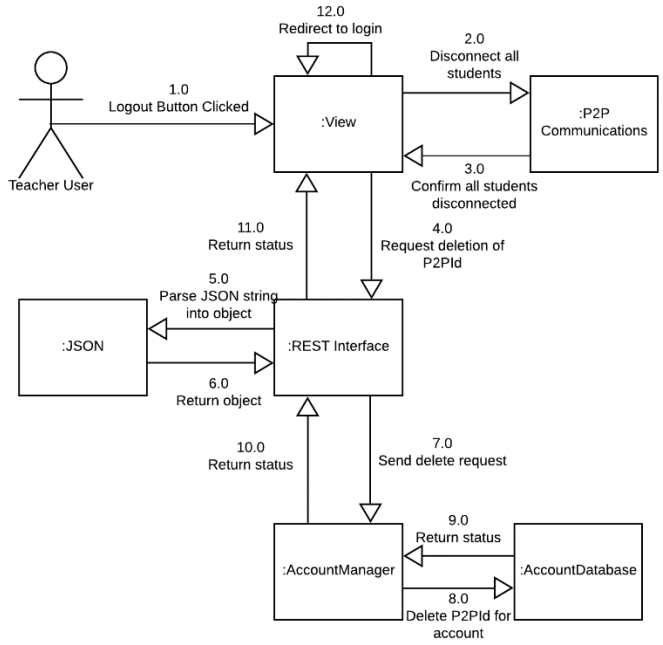

Figure 10: GracefulShutDownTeacher Collaboration Diagram

Bibliography

[1] MacMillan Learning, "Student Store," [Online]. Available: https://store.macmillanlearning.com/us/product/iClicker2-student-remote/p/1498603041.

[2] Tophatmonocle Corp., "Pricing," [Online]. Available: https://tophat.com/pricing/.

[3] UC San Diego, "Registration Fee for Used iClickers Starting Winter 2015," [Online]. Available: https://acmsblog.wordpress.com/2014/12/11/used-iclicker-fee-coming-for-winter-2015/.

[4] Kahoot!, "Pricing," [Online]. Available: https://kahoot.com/businesses/pricing/.