TECHNISCHE UNIVERSITÄT DRESDEN

Department of Computer Science Institute for Systems Architecture, Chair of Computer Networks

Master Thesis

GRAPHICAL DISCUSSION SYSTEM

Kaijun Chen

Born on: 18th September 1990 in China
Matriculation number: 3942792
Matriculation year: 2013

to achieve the academic degree

Master of Science (M.Sc.)

Supervisor

Tenshi Hara

Iris Braun

Supervising professor

Prof. Dr. rer. nat. habil. Dr. h. c. Alexander Schill

Submitted on: 22nd July 2016

TECHNISCHE UNIVERSITÄT DRESDEN

Faculty of Computer Science Institute of Systems Architecture Chair of Computer Networks

MASTER THESIS ASSIGNMENT

TOPIC: Graphical Discussion System

Name (sur, given): Chen, Kaijun		Degree Programme:	Master Informatik (PO 2010)
Matriculation No: 3[illegible]942792		Project/Focus:	Tech-enhanced Learning
Responsible Professor:	Prof. Dr. rer. nat. habil. Dr. h. c. Alexander Schill		
Involved Staff:	Dipl.-Inf. Tenshi Hara, Dr.-Ing. Iris Braun		
Start:	8 January 2016	Due:	17 June 2016

GOAL

At the Chair of Computer Networks a teaching and learning platform has been developed. Currently, its features include a text-based discussion system as well as a virtual interac-tive whiteboard system.

The goal of this assignment paper is to combine both systems into one graphical discussion system that allows for textual as well as graphical discussions. Contributions need to be quotable as is custom in forum systems, however allowing to not only annotate graphical contribution like images, but actually enabling modification of quoted graphical contents. Additionally, any contribution (textual as well as graphical) should be manageable by means of graphic interaction, especially drag & drop gestures with mouse as well as finger tips.

In order to achieve the goal, all aspects of modern web technologies need to be considered, especially HTML 5. Additionally, a feasible concept for storing of contributions and their relations is required on the server. The data model must respect possible future additions, especially storing of client-side private keys for encryption within the local browser storage.

Existing solutions and concepts must be investigated and assessed before conceiving a concept for the desired graphical discussion system. Afterwards, a proof-of-concept implementation is mandatory.

An evaluation of the graphical discussion system should be executed, focussing on usability aspects as well as the capabilities of the conceived data model.

i. A. Braun

Prof. Dr. rer. nat. habil. Dr. h. c. Alexander Schill
(responsible professor)

FOCUSES

- Investigation of related work and current state of research,
- definition of requirements and criteria for quantitative design,
- conception of an evaluation method,
- implementation of proof-of-concept components, and
- evaluation and assessment of the results.

30. Mai 2016

TECHNISCHE UNIVERSITÄT DRESDEN

Fakultät Informatik Prüfungsamt

Antrag auf Verlängerung der Bearbeitungszeit

(•) **Master-Arbeit** (maximal 13 Wochen)

( ) **Bachelor-Arbeit** (maximal 13 Wochen)

Zutreffendes ankreuzen

Name Chen	Vorname Kaijun
Geburtsdatum 18.09.1990	Fachsemester
Matrikelnummer 3[illegible]942792	Studiengang Informatik Master (PO 2010)

Betreuender HSL: Prof. Dr. Dr. h.c. Alexander Schill

Beginn: 8. Januar 2016

Abgabe: 17. Juni 2016

Dauer der Verlängerung: 5 Wochen

Neuer Abgabetermin: 22. Juli 2016

Begründung der Verlängerung:

Auf Grund der Evaluationskomplexität muss die Fixierung der schriftlichen Ausarbeitung verzögert werden. Damit die Qualität der Ergebnisse nicht gefährdet wird, ist eine Verlängerung um ca. 1 Monat (5 Wochen) als Sicherheitspolster erforderlich.

inval. Mitarbeiter
(Hans, Dipl.-Inf.)

Zustimmung betreuender HSL : Schill

Dieser Antrag ist mit einer Kopie der Aufgabenstellung für die ~~Diplom~~Masterarbeit termingerecht im Prüfungsamt vorzulegen.

Technische Universität Dresden
Fakultät Informatik
Prüfungsamt
01062 DRESDEN

24. Mai 2016
Datum

Unterschrift Prüfungsamt

Entscheidung des Prüfungsausschusses:

Dem Antrag auf Verlängerung wird stattgegeben / ~~nicht stattgegeben~~

15.6.2016
Datum/Unterschrift Prüfungsausschuss

TECHNISCHE UNIVERSITÄT DRESDEN

Department of Computer Science Institute for Systems Architecture, Chair of Computer Networks

Statement of authorship

I hereby certify that I have authored this Master Thesis entitled *Graphical Discussion System* independently and without undue assistance from third parties. No other than the resources and references indicated in this thesis have been used. I have marked both literal and accordingly adopted quotations as such. They were no additional persons involved in the spiritual preparation of the present thesis. I am aware that violations of this declaration may lead to subsequent withdrawal of the degree.

Dresden, 22nd July 2016

Kaijun Chen

CONTENTS

1. Introduction 11
   - 1.1. Motivation . . . 11
   - 1.2. Goals and Research Questions . . . 11
   - 1.3. Thesis Outline . . . 12

2. State of the Art 13
   - 2.1. Modern Web Development . . . 13
     - 2.1.1. Evolution . . . 13
     - 2.1.2. RESTful Web Service . . . 17
   - 2.2. Graphics on the Web . . . 18
     - 2.2.1. Canvas . . . 18
     - 2.2.2. SVG . . . 19
     - 2.2.3. Comparision . . . 19
   - 2.3. Real-Time Communication . . . 20
     - 2.3.1. Long polling . . . 20
     - 2.3.2. WebSockets . . . 21
     - 2.3.3. WebRTC . . . 21
     - 2.3.4. Advantages . . . 21

3. Conception 23
   - 3.1. Aims and Objectives . . . 23
     - 3.1.1. Basic Functionalities . . . 23
     - 3.1.2. High Interactivity . . . 26
   - 3.2. General Concept . . . 27
     - 3.2.1. Architecture . . . 29
     - 3.2.2. Communication . . . 30
   - 3.3. Data Definitions . . . 30
     - 3.3.1. General Data Model . . . 31
     - 3.3.2. RESTful API Definitions . . . 32
     - 3.3.3. WebSocket Definitions . . . 35
   - 3.4. Graphical Data Serialization . . . 35
     - 3.4.1. Canvas over SVG . . . 36
     - 3.4.2. Storable and Reversible Canvas Data . . . 36

3.4.3. Drawing Tool . . . 38
3.5. Real-Time Demand . . . 40

4. Implementation 43
4.1. General . . . 43
4.1.1. Platform and Framework . . . 43
4.1.2. Architecture . . . 44
4.1.3. Automatization . . . 45
4.1.4. Storage Structure . . . 47
4.2. Server of Graphicuss . . . 47
4.2.1. Architecture . . . 47
4.2.2. Model Layer Implementation . . . 49
4.2.3. Authentication . . . 50
4.2.4. WebSocket Implementation . . . 52
4.3. Client of Graphicuss . . . 53
4.3.1. Architecture . . . 53
4.3.2. Composition of Components . . . 55
4.3.3. Data Flow . . . 56
4.4. Drawing Tool for Graphicuss . . . 57
4.4.1. Objectified Canvas . . . 57
4.4.2. Drawing Tool . . . 58

5. Evaluation 63
5.1. Usability . . . 63
5.1.1. System Usability Scale . . . 63
5.1.2. Interview based Usability Test . . . 64
5.2. Data Model Evaluation . . . 67
5.2.1. Evaluation Approach . . . 67
5.2.2. Analysis of Result . . . 68
5.3. Graphical Rendering Performance . . . 69

6. Conclusion and Future Work 73
6.1. Conclusion . . . 73
6.1.1. Modern Web Application . . . 73
6.1.2. Objectified Canvas . . . 73
6.1.3. Real-time communication . . . 74
6.2. Future Work . . . 74

List of Figures 75

List of Tables 77

List of Codes 78

A. System Usability Scale Table 81

B. System Usability Interview 82

References 82

1. INTRODUCTION

1.1. MOTIVATION

With the rapid development and popularization of the internet and technology, the traditional educational activities are moving to the online platforms.

A discussion of the teaching content or the educational material is always essential for both tutors and students in the teaching activities. However, the traditional approach of discussing shows its limitations. Not all the students have the same question and the tutor is able to offer explanation for only one question at the same time. Moreover, a discussion can only be performed normally after courses also requires the absence of the students as well as the tutor.

Therefore, a discuss system with intense interactivity as well as in crowdsourcing way is highly needed. To achieve high interactivity, a discuss system with graphical tool and real-time data communication is proposed. Students are able to contribute their questions and answers to get to the bottom of his deficiencies of teaching content and the educational material. In addition, students who have the same questions are able to acquire the best solution instantly which is recommended and approved by the community.

1.2. GOALS AND RESEARCH QUESTIONS

This work focuses on the development of a Web-based graphical discussion system, which features storable and quotable graphical discussion contribution. In addition, in order to improve the interactivity of the system which plays a significant role for the educational purpose,real-time communication is also designed.

Within this thesis the following research questions will be addressed in order to allow the conception, implementation and evaluation of the both client and server sides of graphical discussion system.

- How to construct and implement the system with modern Web technologies.

- How to design the data model of graphical contribution which is able to be persisted and restored back to sketch board.
- How to apply the real-time communication technology to improve the interactivity of the system.

1.3. THESIS OUTLINE

This master thesis has the following structure.

Chapter 2 gives general introductions of modern Web technologies. The new development process and architecture of Web application are discussed. Afterwards various graphics on the Web are presented and compared. Finally, alternatives of real-time communication technologies are listed.

Chapter 3 considers the general concept of the system. The first section deals with the requirements and mockups with expected functionalities. Thereafter the conception of general architecture is described. In addition, the concept of data model within the system is defined. Finally a feasible concept of serialized graphical data for storing is designed.

Chapter 4 covers the implementation of both client and server application of the graphical discuss system. Firstly, general overview about the application structure is given. The storage structure and relation of the data model are shown. At last, the implementation of drawing tool which provides user interfaces for drawing is presented.

Chapter 5 obtains the evaluation of system usability and graphical data model. The measurement methodology is introduced and test results are analyzed.

Chapter 6 is the epilogue with a summary of the thesis. The final section discusses future work or researches.

2. STATE OF THE ART

The following chapter gives an overview of state of the art. To achieve the high interactivity and responsiveness in the graphical discussion system, a lot of modern Web technologies should be applied.

However, there are always plenty of alternatives for each technology which could differ from system to system. So it is important to investigate and analyze the existing solutions and capture an overview about different alternatives of technologies. It is also vital to understand the benefit and drawback of the technologies used.

First of all, the general modern Web technology and development workflow will be introduced, which goes through the whole development and has a great impact on the development efficiency. The next part describes the different graphical technologies on the Web. Then an overview of the real-time communication technologies is illustrated and evaluated.

2.1. MODERN WEB DEVELOPMENT

2.1.1. EVOLUTION

With the increasing complexity of a web app and high demand of agile development, people are always thinking about how to improve patterns of the workflow in web development as well as the architecture of a web app. To achieve better scalability, maintainability and ubiquity of a web app, the architecture of an entire web app was involved in the past several years.

EARLY AGE

At the early age of web development, the back-end did all jobs for both client side(browser) and server side, such as rendering, calling system service, composing data, etc. Figure 2.1 shows the overview of web architecture: The advantage of this pattern is clear: The development and deployment are simple and straightforward, if the business logic doesn't

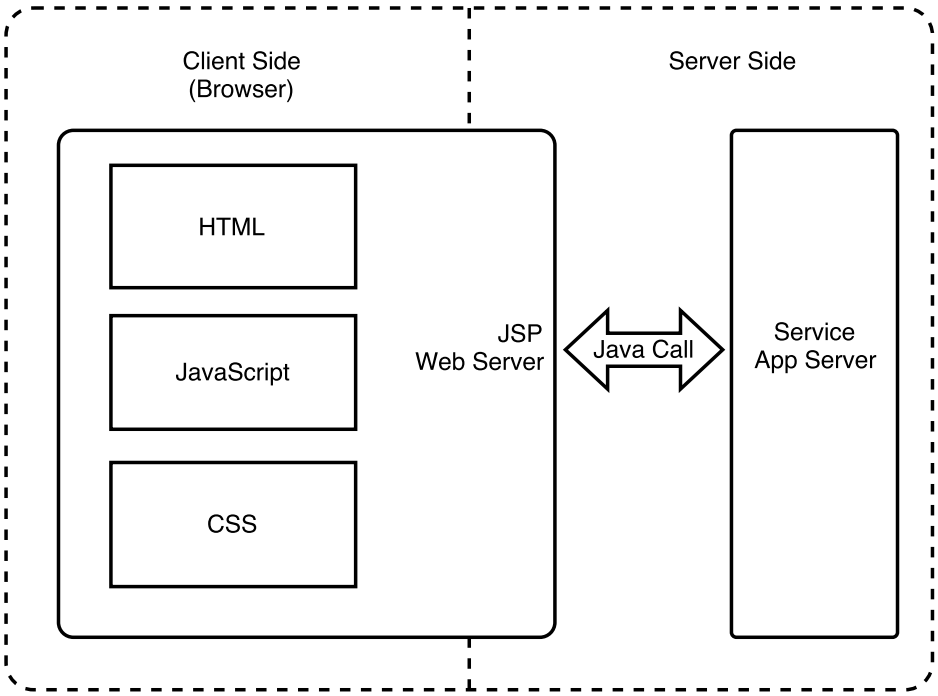

Figure 2.1.: Web architecture in early age

become complex. But with the increasing complexity of the product, the problems show up:

1. Development of front-end heavily depends on the whore development environment. Developers have to start up all the tools and services for testing and debugging only some small changes on view. In most cases, the front-end developer who isn't familiar with the back-end needs help while integrating the new views into the system. Not only the efficiency of development, but also the cost of communication between front-end developer and back-end developer are huge problems.
2. The own responsibility of front-end and back-end mess up, which could be expressed by the commingled codes from different layers, for example, there is no clear boundary from data processing to data representing. The maintainability of the project becomes worse and worse with the increasing complexity.

It's really significant to improve the maintainability of code, as well as the efficiency and responsibility of the division of work from both front-end and back-end in the whole web development phase. In the section below, an evolution of the technical architecture will reveal how these problems are solved.

WEB 2.0

Along with the birth of Gmail[1] in 2004, which is noted for its pioneering use of Ajax, the web application started to behave more interactively. Browser began to take over the job of data fetching, processing, rendering, such a sequence of workflow which could only be done by the server side formerly.

[1] https://mail.google.com/ - accessed 25 May 2016

The architecture in the Web 2.0 generation is presented in figure 2.2.

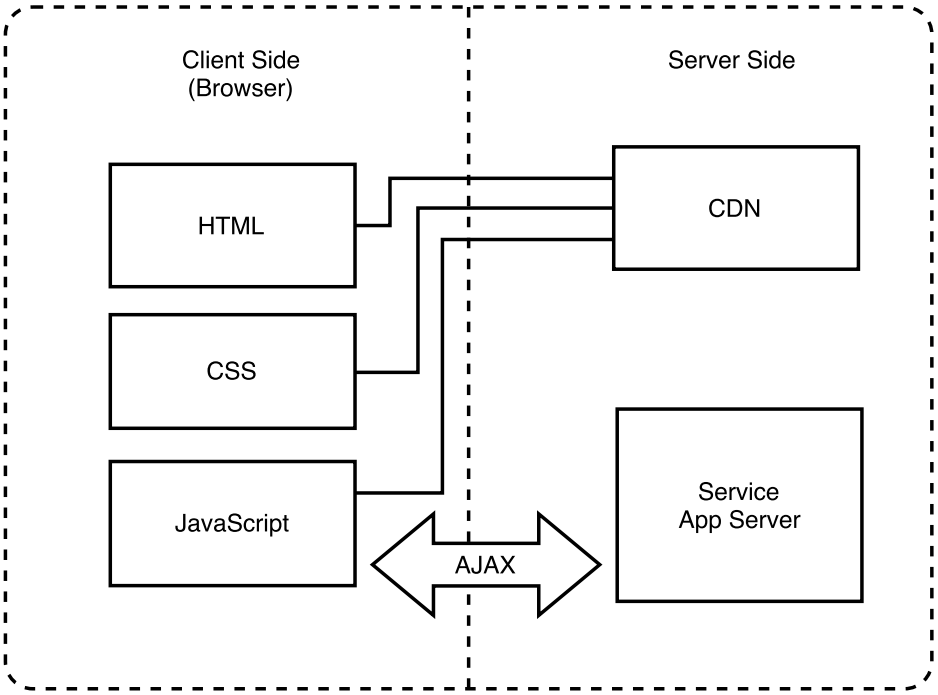

Figure 2.2.: Web 2.0 architecture

By using Ajax, the client has the ability to fetch data stream asynchronously, after which the client will consume the data and render it into the specific section of view. Usability was dramatically improved, because the entire view represented to users will not be refreshed and the front-end is able to process and render the data in its own intension, which means more flexible control of the consumption of data.

SINGLE PAGE APP

With the evolution of web technologies and promotion of these technologies in modern browsers by browser vendors, a new web development model called SPA was proposed and caught the developers' eye. The back-end is no more responsible for rendering and view controling, it only takes charge of providing services for the front-end.

The structure in figure 2.3 shows that the client side has the full control of view rendering and data consumption after data acquisition through web services which is released by back-end with promised protocol. All rendering tasks were stripped off from the server side, which means that the server side achieves more efficiency and concentrate more on the core business logics.

But more responsibility in front-end means more complexity. How to reduce the complexity and increase the maintainability of a front-end project becomes a significant problem. Developers come up with a new evolved variant of SPA as demonstrated in figure 2.4.

In general, the architecture is componentized and layered into the template, controller and model. Each component is isolated and has its own view as well as correlated logics. Front-end frameworks like EmberJS, AngularJS, ReactJS are providing such an approach

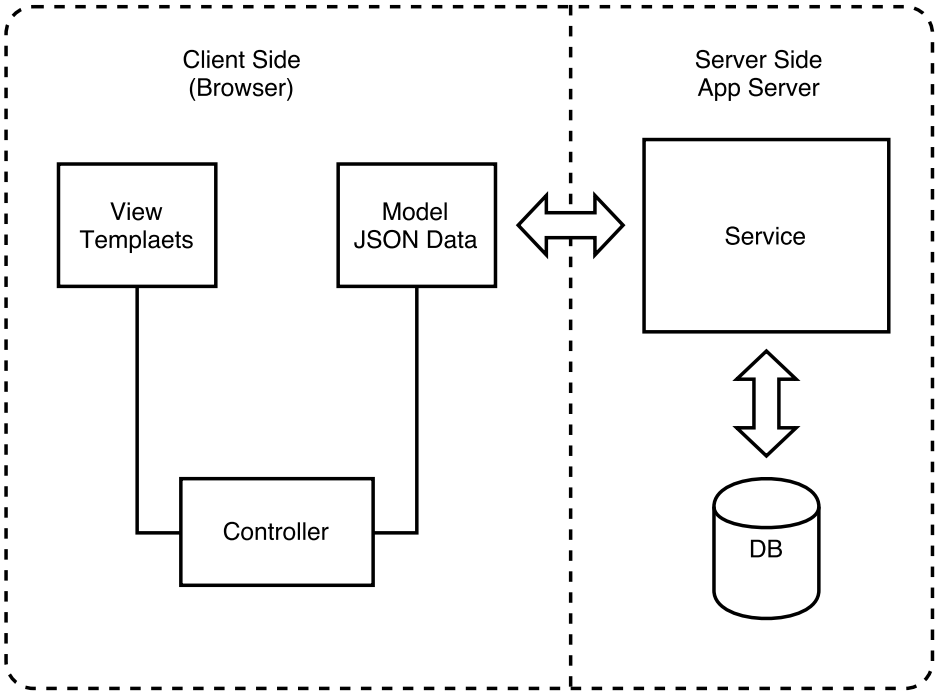

Figure 2.3.: SPA architecture

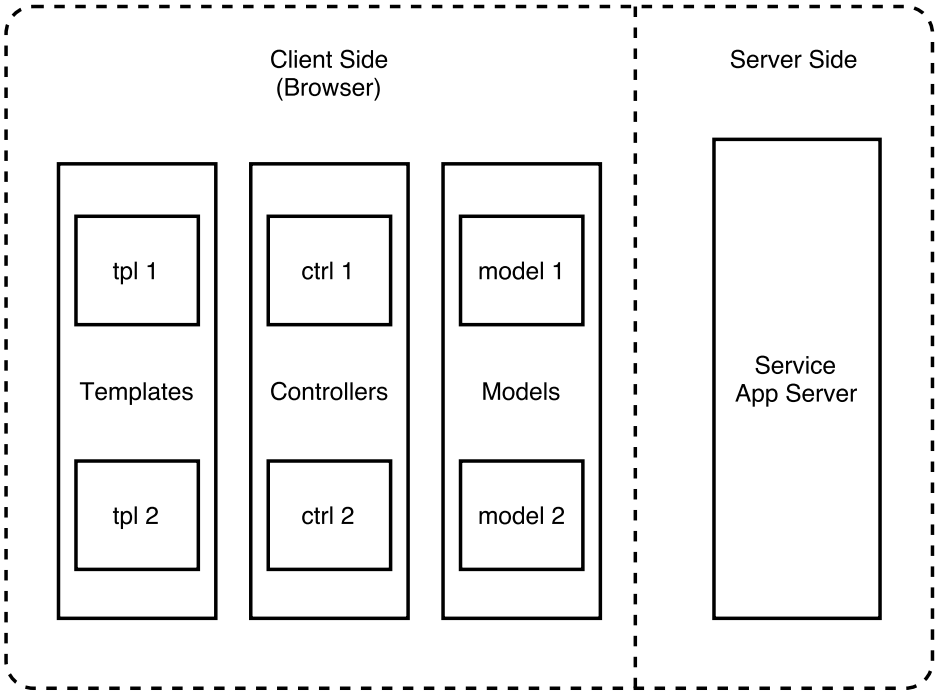

Figure 2.4.: Components in SPA

and development pattern for developers to build modern web apps. With this approach, a giant and complex front-end app is broken up into fine grained components, therefore, components are easy to reuse if the components are well abstracted in a proper way. In addition, the maintenance of each component is also effortless.

TRADE-OFF

To summarize, a single-page app has a lot of benefits:

1. Rational seperation of works from front-end and back-end: client takes charge of view rendering and data representation, as well as slight data processing if needed; the server focus on providing services of the core logics, persistence of data, and also computational tasks.
2. High interactivity and user experience in client side: asynchronous data fetching and view rendering implies no more need of hard reloading the page which user is viewing and the current states of the page could also be preserved.
3. Efficiency in server side: rendering tasks are stripped off from server side.
4. Ubiquity: with the separation of services provided by server side, not only the web browser, but also other clients in other platforms such as Android, iOS apps is able to access and consume the services.

But SPA also has its deficiencies:

1. SEO unfriendly: because the page are not directly rendered by server side, and the web crawlers are not able to run JavaScript codes like a browser does, the site could not be crawled properly under normal circumstances. So if SEO results really matter for the app, SPA is obviously not the best choice.
2. Excessive http connections: all the data is acquired from different services through diverse APIs, thus multiple HTTP connections are established and performed parallelly, whose initial time of connections for partial data could be much more than a single connection in the traditional way. So it's highly needed to merge the services and find a balance between data model complexity and time consumption.

2.1.2. RESTFUL WEB SERVICE

REST stands for REpresentational State Transfer. More than a decade after it was introduced, REST has become one of the most essential technologies for Web applications[1]. REST is web standards based architecture which uses the HTTP Protocol for data communication. It centers upon resource and a resource is accessed by a uniform interface using HTTP standard methods.

In REST architecture, a REST Server simply provides access as well as operations of resources, while a REST client accesses and manipulates the resources by using different methods. Here URIs are used to locate the resources.

The key principles, which make RESTful applications to be lean and fast, are listed as follows[2].

- URI as an identifier for resource: Through a RESTful web service, clients interact with the targets of the resources exposed. URIs are used as the identifier for resources, which provide a global addressing space for resource requesting and acquisition.
- Uniform interface: A fixed set of four create, read, update, delete operations is used for manipulation of resources, which could also be represented by HTTP standard methods: PUT, GET, POST, and DELETE. Retrieving the current state of a resource could be achieved by GET. PUT is for creating a new resource, which can be removed by using DELETE. A new state onto a resource is transferred if POST is used.
- Self-descriptive messages: Resources are decoupled from their representation. As a result, the content of the resource can be accessed in a variety of formats, such as HTML, plain text, JSON, and others.
- Stateful interactions through hyperlinks: Every interaction with a resource is stateless. However, several techniques are able to be applied to achieve the exchange of state, such as URI rewriting, cookies, and hidden form fields. State can be embedded in response messages to point to valid future states of the interaction[1].

2.2. GRAPHICS ON THE WEB

2.2.1. CANVAS

Canvas, which is added in HTML5 as a standard, is an element defined in HTML code with width and height attributes. Graphics can be drawn on Canvas by using HTML5 Canvas APIs via scripting in JavaScript. A full set of drawing functions and helper functions could be used for accessing or rendering pixels on the Canvas area, which also means graphics can be generated dynamically and programmatically.

Every HTML5 canvas element must have a context. All HTML5 Canvas API to be used are defined within the context. There exist two different types of context in Canvas: 2d context for drawing 2D graphic and 3d context for 3D graphics. The latter is actually called WebGL and it's based on OpenGL ESs[3].

The coordinate system of Canvas sets the origin offset at the upper-left corner of the canvas, with X coordinates increasing to the right and Y coordinates increasing toward the bottom of the canvas. Which means, the Canvas space doesn't have points with negative coordinates.

The significant features of Canvas are listed as follows:

- Interactivity: Listeners for keyboard, mouse or touch event are able to be created in the context of Canvas. Users' actions can be captured and response will be made according to the action.

- **Flexibility**: A variety of shapes like line, rectangle, circle or even text are able to be painted on the Canvas using the native methods. It is also possible to add animations, even video or audio could to the Canvas[4].
- **Browser/Platform Support**: Unlike Other graphic technology like Flash or Silverlight, which has a very restricted platform support, Canvas is supported by all major browsers which follows the HTML5 standard. In addition, it can be accessed on mobile devices via browser environment.
- **Performance**: Rendering pixels on Canvas is much faster comparing to other graphic technologies on Web[5].

2.2.2. SVG

SVG is an XML language, similar to XHTML, which can be used to draw graphics, such as the ones shown to the right. It can be used to create an image either by specifying all the lines and shapes necessary, by modifying already existing raster images, or by a combination of both. The image and its components can also be transformed, composited together, or filtered to change their appearance completely[6].

Similar to HTML, which provides various elements for defining different styles, SVG provides elements for circles, rectangles, and simple and complex curves. A simple SVG document consists of one *<svg>* element as its root element and several children elements of basic shapes. The composition of elements in SVG builds the graphic. The code listing 2.1 shows a simple example of a SVG element with one rectangle, one circle and a text inside it.

```
1 <svg width="300" height="200" >
2   <rect width="100%" height="100%" fill="red" />
3   <circle cx="150" cy="100" r="80" fill="green" />
4   <text x="150" y="125" font-size="60" fill="white">SVG</text>
5 </svg>
```

Listing 2.1: Simple Example of SVG elemnt

SVG is natively supported in all modern browsers and also has a good compatibility with old broswers[7].

2.2.3. COMPARISION

REFERENCES TO ALREADY DRAWN ELEMENTS

Since HTML5 Canvas is simply a drawing surface for a bit-map and renders the graphics pixel by pixel, Canvas has no knowledge of the graphics it drew: It doesn't persist any information of the graphics' properties such as shape, position or size after the graphics were successfully drawn.

On the other hand, SVG maintains references to each object that it renders. Because each SVG element is created and appears in real DOM element on HTML. By default, this allows

tracking the SVG elements easily and manipulating the every existing element, for example, changing the size of an element or moving the element to another position.

RENDERING PERFORMANCE

A testing of rendering performance with both Canvas and SVG in different dimensions is performed by Boris Smus[8]. The figure 2.5 illustrates the results.

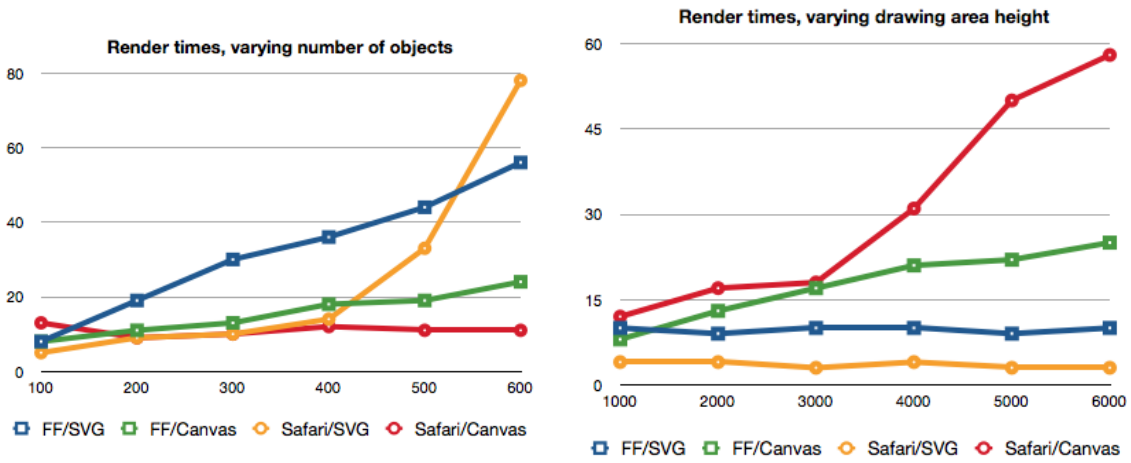

Figure 2.5.: Performance comparison of Canvas and SVG[8]

The result of the first experiment, which tested the rendering time of Canvas and SVG with increasing amount of components drawn on it, clearly shows that SVG performance degrades dramatically in the number of objects. However, Canvas performance remains at a near-constant low. The reason is that Canvas is just a bitmap buffer, while SVG has to maintain additional references to each element of the object rendered.

On the right of the figure, the second experiment reveals that canvas performance degrades significantly with varying the size of the drawing area. On the other side, SVG performance is not affected completely. Canvas rendering performance seems to decrease linearly with the amount of pixels in the canvas area. However, combining two dimensions, Canvas still achieves the better rendering performance.

2.3. REAL-TIME COMMUNICATION

2.3.1. LONG POLLING

HTTP Long-polling is a technique used to push updates from server to client. Establishing a connection to the server using long polling is like AJAX, but the difference is that the keep-alive connection opens for a certain time period. During the connection persisted, the client can retrieve data from the server connected. In case that the connection is closed or timeout unexpectedly, the client has to keep requesting periodically in order to reconnect to the Server. On the server side, long polling requests are still treated as HTTP requests same as AJAX.

Since long polling only uses normal HTTP requests, it is supported in all major browsers.

2.3.2. WEBSOCKETS

WebSockets is an advanced technology that provides the possibility to open an interactive communication session between client and server[9]. With the WebSockets API, messages are able to be transmitted to a server and event-driven responses can be returned to the client without having to poll the server for a reply.

Starting a WebSockets connection will create a TCP connection to server in the first place, and keep it as long as needed. The connection can be easily closed by either by server or by client. After the HTTP compatible handshake process has succeeded, data could be exchanged bi-directionally between server and client. Therefore, WebSockets is suitable for the heavy requirements on frequent data exchange bi-directionally. In addition, message sent through WebSockets is simply encrypted[10].

2.3.3. WEBRTC

WebRTC is an industry and standards effort to put real-time capabilities into browser to browser communication and make these capabilities accessible to web developers via standard HTML5 tags and JavaScript APIs[11]. WebRTC is used to enable the communication between multiple clients. By design, WebRTC allows to transport data in reliable as well as unreliable ways. This is generally used for high volume data transfer such as video/audio streaming where reliability is secondary and few frames or reduction in quality progression can be sacrificed in favor of response time. Both sides (peers) are able to push data to each other independently.

2.3.4. ADVANTAGES

The primary advantage of WebSockets is that the connection is not a normal HTTP request, but the proper message based communication protocol. That allows you to achieve huge performance and architecture advantages. Comparing to long polling, there is no need to start connecting multiple times while using WebSockets. Since the time consumption of establishing a connection is the major part of the total time consumption of a request, reducing the connection numbers will significantly improve the performance and efficiency.

However, WebRTC is only used for peer to peer connection, but not client to server. Therefore, it is out of the scope of this thesis in general.

3. CONCEPTION

In this chapter a conception of the discuss system, including both client and server side will be described. In the first section, the requirements of the system are analyzed. Mockups with expected functionalities are attached during the analysis of requirements. Afterwards, the general architecture of the system and the data definitions within the system are concepted. At last, the approach of serializing the graphical data for persistence purpose is introduced.

3.1. AIMS AND OBJECTIVES

Before starting with the concept the graphical discuss system, it's necessary to analyze requirements and objectives behind the origin motivation in the first place. It should be defined at first, what kind of functionalities should be achieved and how the system behaves.

3.1.1. BASIC FUNCTIONALITIES

As a graphical discuss system for the educational purpose, the system should contain basic functionalities on the prototype of a forum which could be organized by classes. So class management, question management and answer management are the three essential parameters to be designed at the start.

COURSE MANAGEMENT

Each question should have a certain domain of its content, so the questions are organized by classes initially. The features of course management should be:

1. Create Course: The user who is identified as a tutor is able to create courses and maintain the courses he created. While creating the course, the tutor can define the name of the course and upload an image as a background of the course for better recognition. In addition, concrete description of the course could also be added to the description area.

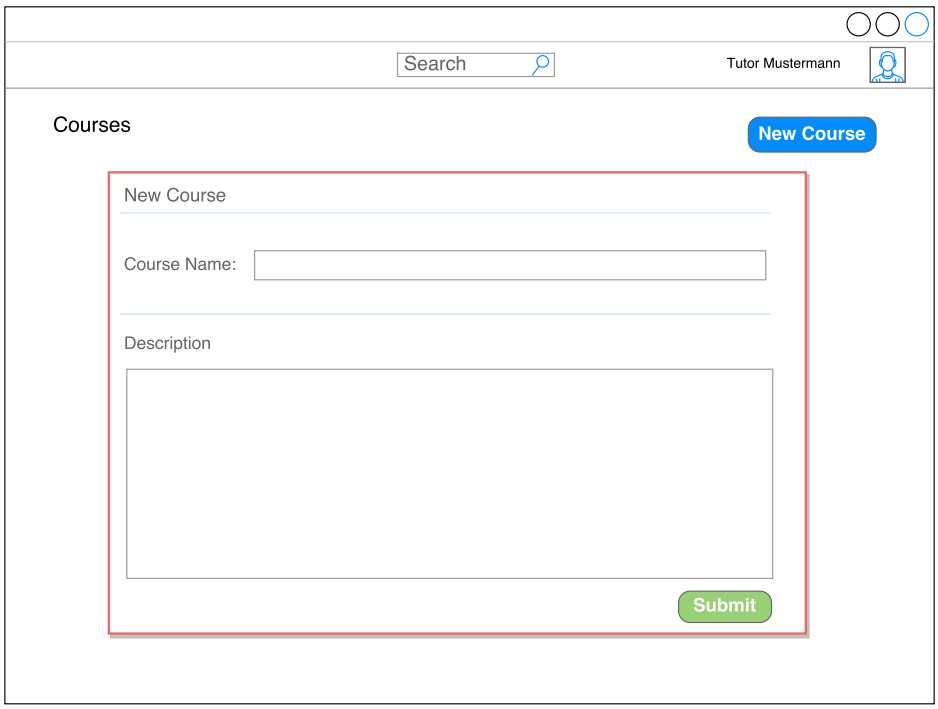

Figure 3.1.: Submit a new course

2. **Search Course**: After a course is created, a corresponding unique identifier code for the course will be generated at the same time. The students are able to find the course through the identifier code.

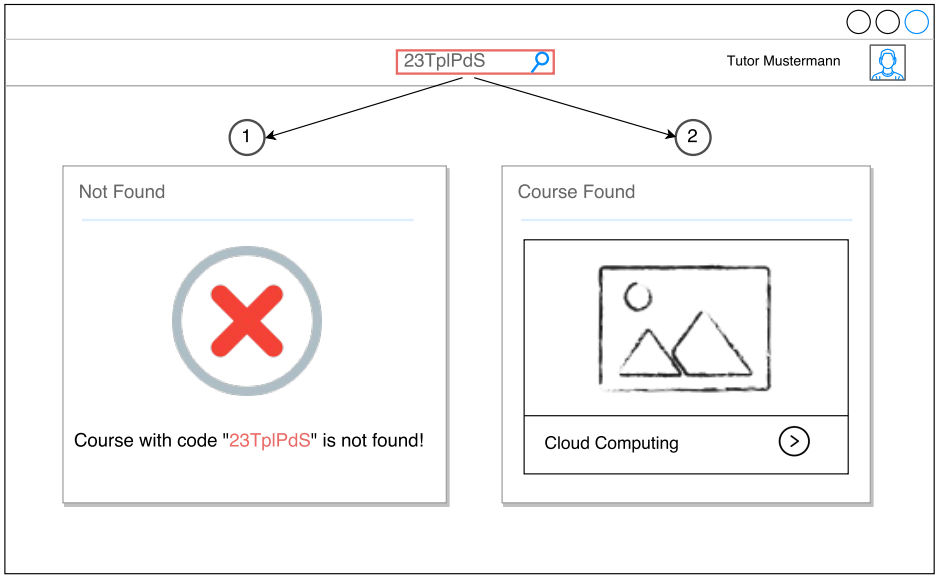

Figure 3.2.: Search course with code

3. **Favor Course**: If a student is interested in a certain course, he is capable to add the course to his favorites list so that it's easy to find and access the course he liked later.

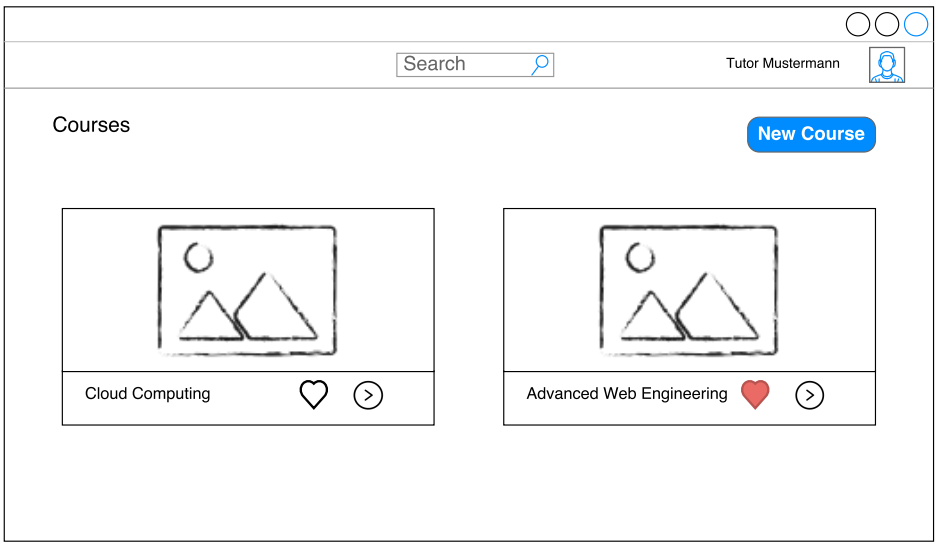

Figure 3.3.: Favor course

QUESTION MANAGEMENT

1. **Submit/Edit/Withdraw Question**: The student who is confused with the teaching content can submit his own question with detailed description in a certain course. The user is also permitted to edit the question if he wants to add more precise informations or modify the unclarity he made to the question. Withdrawing of his own question is also possible, but only when there're no contributes made to the question.

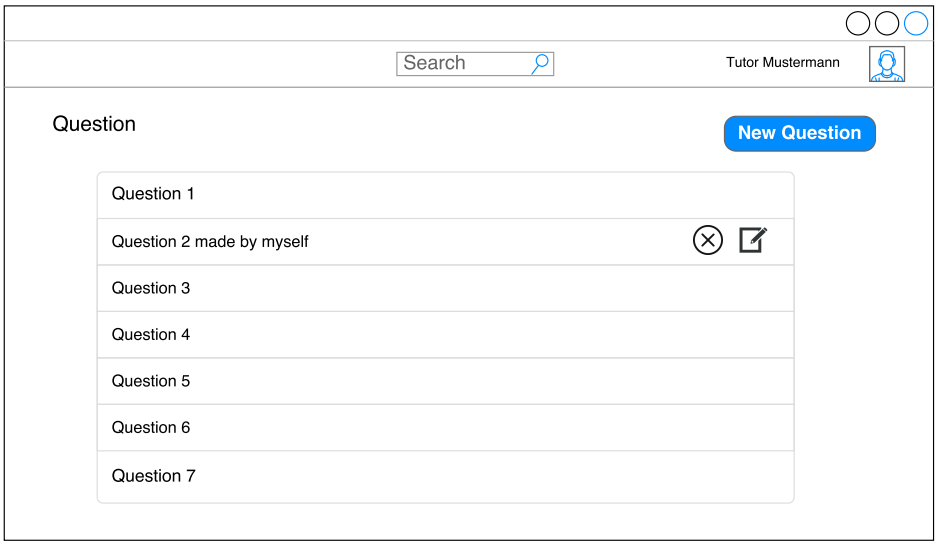

Figure 3.4.: Submit a new question; withdraw or modify own question

2. **Upvote/Downvote Question**: An assessment of a question is decisive for building a better community with high-quality contents. So the user is able to upvote or downvote of a question and determines if the question is helpful for other members in the community or not.

3. **Favor Question**: If the student considers the question as a helpful and useful content and want to review this question in the future, he can favor the question and locate it

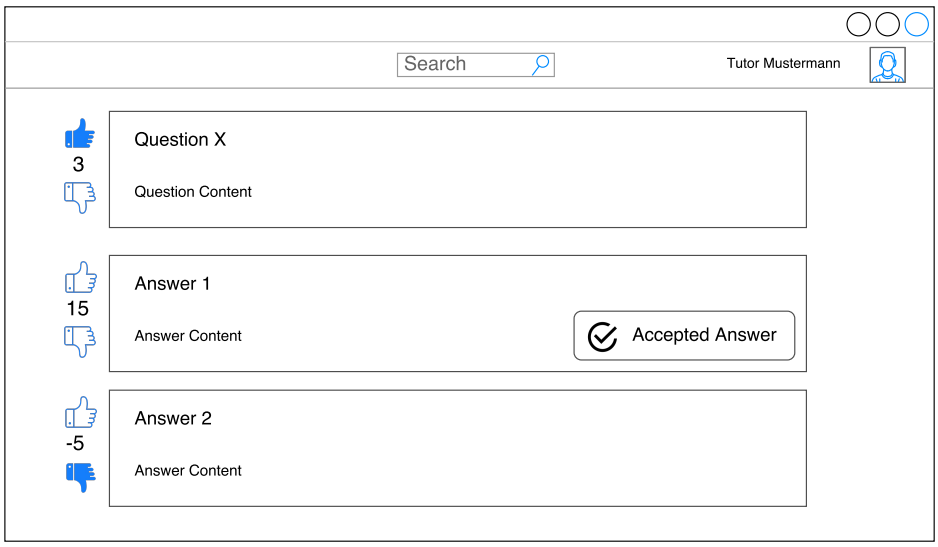

Figure 3.5.: Upvote/Downvote a question or answer

in a certain list.

4. **Accept Answer**: The owner of the question has the right to accept the most useful answer in his opinion, which will be shown up at the top of the answer list.

ANSWER MANAGEMENT

1. **Submit/Modify/Remove Answer**: User who has experience with the question can submit his answer to the question. After the submission, the modification or removal of the user's own question is possible.
2. **Upvote/Downvote Answer**: As mentioned above in subsection of question functionality, a similar idea of assessment should also be applied to answers. Answer with the highest vote will be listed at first.
3. **Quote Answer**: Answers are able to be quoted so that the user can supplement information on the top of the original post or point out the deficiency of the contribute.

3.1.2. HIGH INTERACTIVITY

Building with the basic functionalities is far not enough. To fit the system for educational purpose and improve the interactivity for arousing enthusiasm of students, a drawing tool and real-time functionality should be integrated into the system.

DRAWING TOOL

Normally, some of the thoughts can't be simply expressed by textual description, so a drawing tool should be designed to enable the user to compose not only text but also different components such like rectangle, circle, line and so on, which helps the user to express his question more precisely. The ideal drawing tool should have following features:

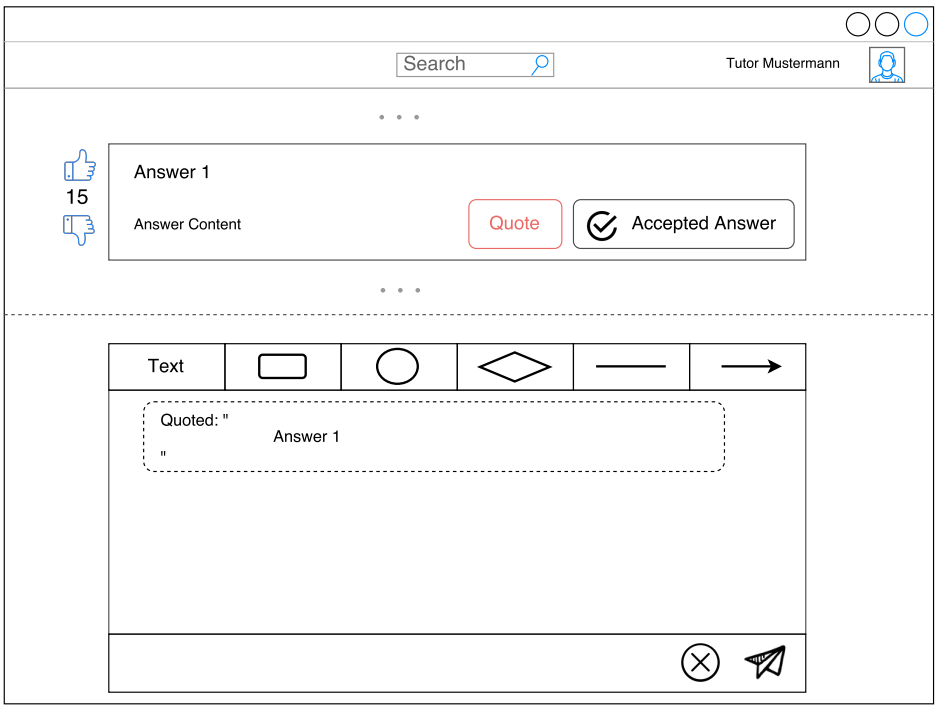

Figure 3.6.: Submit/Quote an answer

1. **Drawing Diverse Components**: Not only text but also diverse components could be drawn while posting a contribution. The styling of a component such as size tuning, color changing is also the essential, which will help emphasize the important part the user expressing.
2. **Drawing History**: During drawing, the user might make mistakes or change mind after placing a component or text. So a history list of drawing actions bundled with undo and redo functionalities will dramatically improve the usability of drawing process.

REAL-TIME

How to ease the approach of content acquisition and improve the interactivity for arousing enthusiasm of students, is also a key point while designing the discuss system. So two major real-time functionalities are featured as follows:

1. **Real-time Question List**: Without requesting the question list initiatively, all new questions posted by other users will be pushed to user automatically. The user doesn't have to concern himself with acquisition of the new content anymore.
2. **Real-time Answer Ordering**: Without refreshing the page, the answers will be re-ordered as new vote action is triggered.

3.2. GENERAL CONCEPT

Before the whole conception of the system, a general conceptual architecture of the system should be defined initially. In order to help understanding how the system works, the

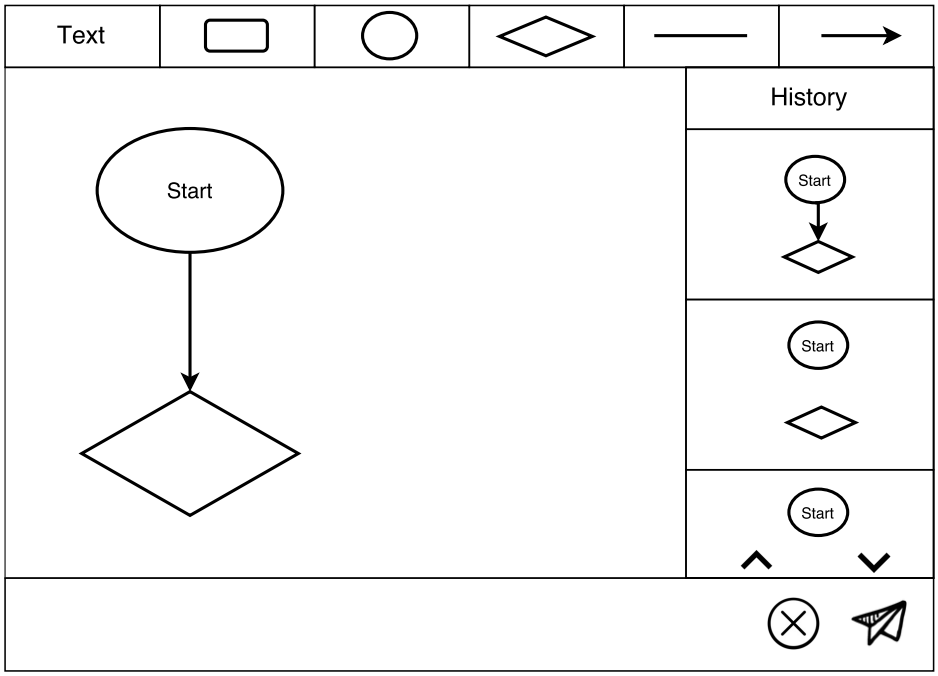

Figure 3.7.: Drawing editor with drawing history

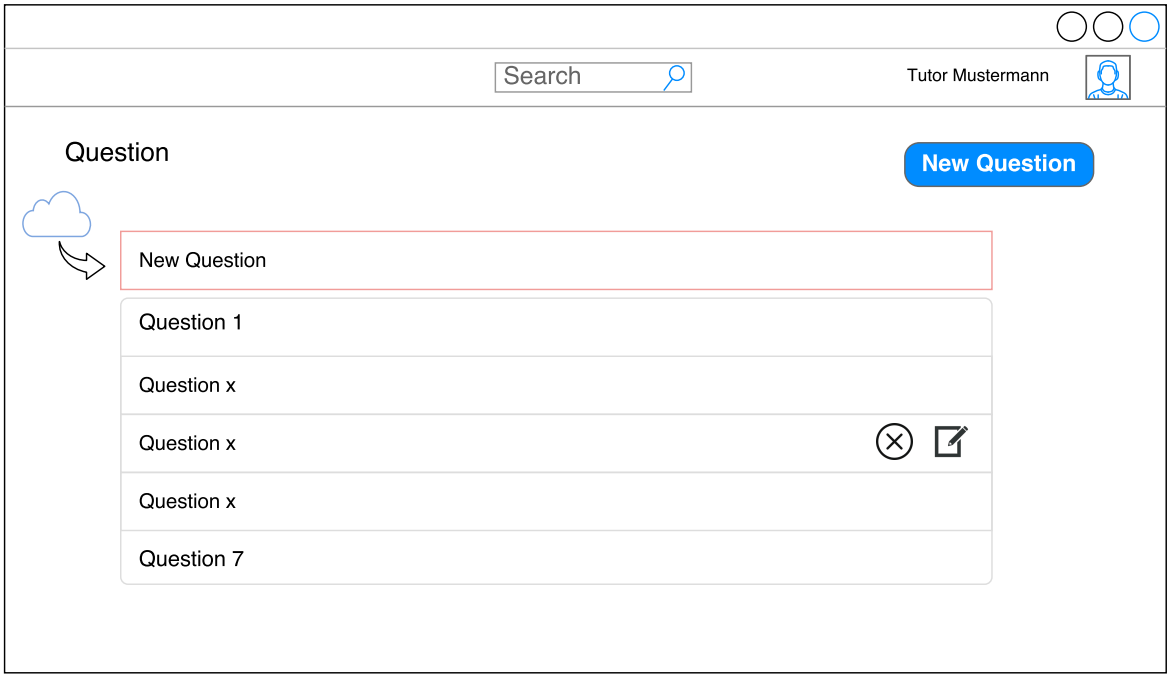

Figure 3.8.: Notify with new question automatically

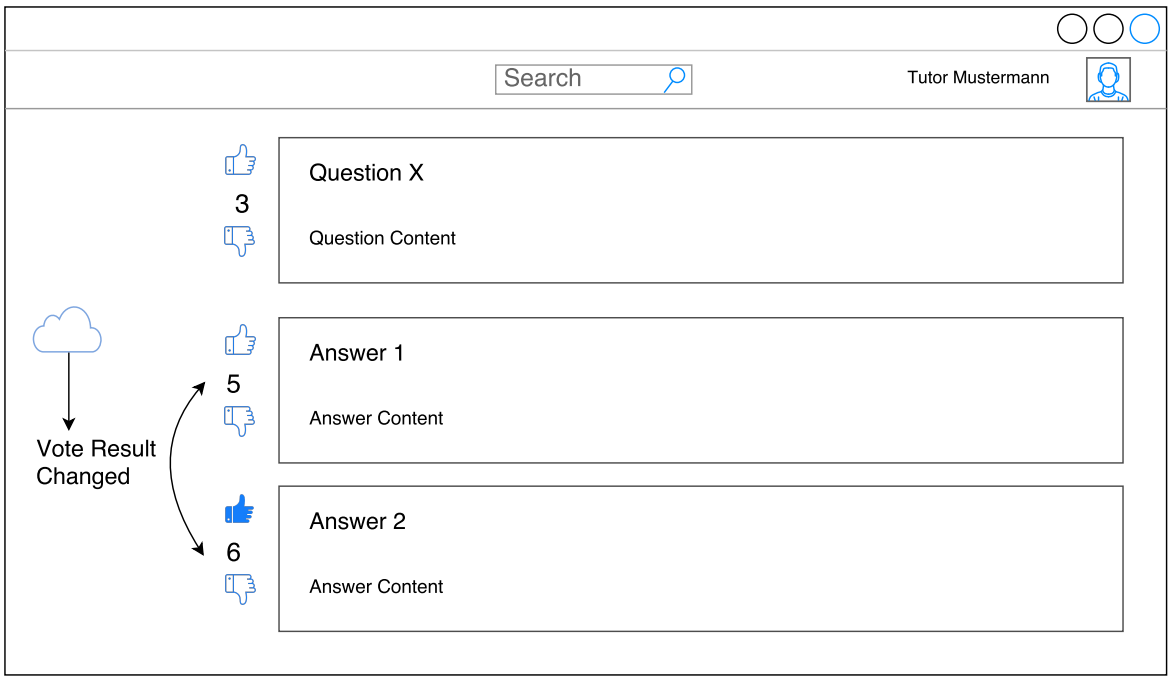

Figure 3.9.: Auto re-order answer if vote contributions changed

primary data flow between different domains will also be described.

3.2.1. ARCHITECTURE

According to the analysis result of Single-Page-App in chapter 2, and considering the demand on high interactivity and ubiquity as well as scalability in the graphical discuss system, leveraging SPA architecture will benefit a lot and accelerate the implementation of the system.

In general, the entire system will be divided into two parts: namely client and server-side. Each side is basically fully independent to the other and has its own responsibility.

- **Client**: The client is totally responsible for initial view rendering and view re-rendering as the view model changes.
- **Server**: The server is in charge of core business logic, data processing, data persistence and also provides the client interfaces for data acquisition.

The general architecture of the system is described in figure 3.10, the only bridge between the client and server-side is data transmission service. Complete separation of both sides will also accelerate the development flow in the implementation phase. Once the protocol of data transmission services is fully confirmed and defined, development of each side is able to be performed parallelly. Furthermore, technical choices on both sides are more flexible. Both sides are able to apply the technologies which fit them most without coupling to each other, the only thing they should obey is to follow the protocol of data transmission.

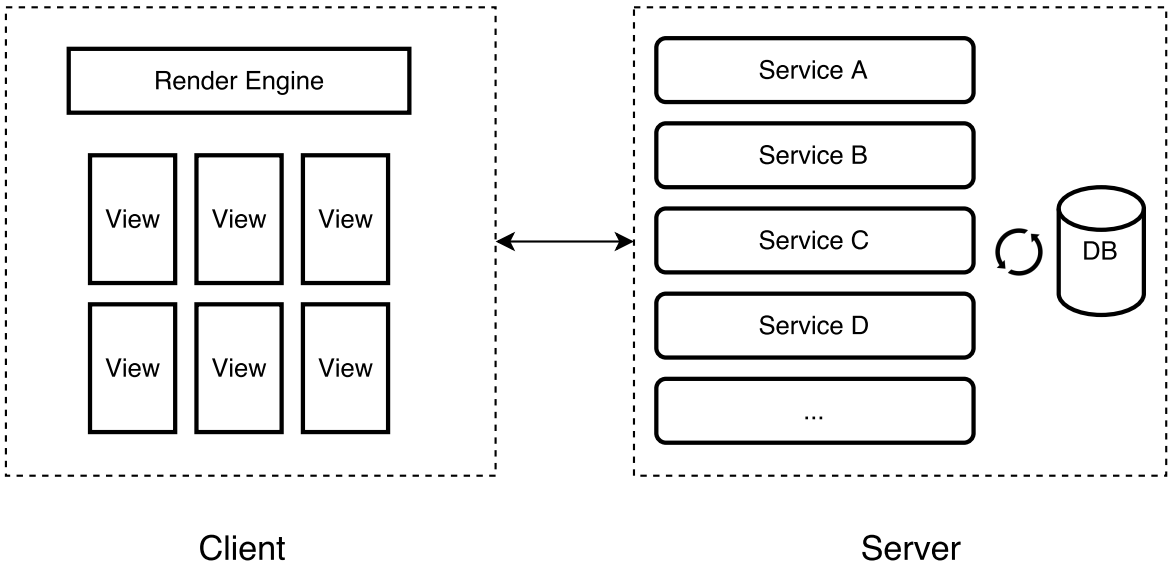

Figure 3.10.: General architecture in conception

3.2.2. COMMUNICATION

As mentioned above in section 3.2.1, data communication is the only coupling factor in the general architecture. In this system, there exist two different types of protocols: standard HTTP using REST architecture and WebSockets with persistent connection. Each data transmission protocol has its own responsibility and usage scenario.

- HTTP with REST architecture: Data which is requested initiatively is transferred over HTTP. The HTTP connection will be closed as soon as the data is successfully transferred.
- WebSocket with persistent connection: Reactive data with real-time need is transferred over WebSocket. After the persistent connection is established, clients are able to receive the data at the first moment as the state of data is updated.

As the figure 3.11 shows, in case data for view model is acquired from the server despite transferred over HTTP or WebSocket, the views are re-rendered. Comparing with HTTP, the specialty of data acquisition over WebSocket is: a listener for specific resource with unified processes of data processing and automatic view re-rendering will be created.

3.3. DATA DEFINITIONS

In this section, data modelling of the system, including definitions of data domain, data fields for each domain, and relation between domains will be performed in the first place. In the following subsection, APIs corresponding to related operations on data models will be assigned.

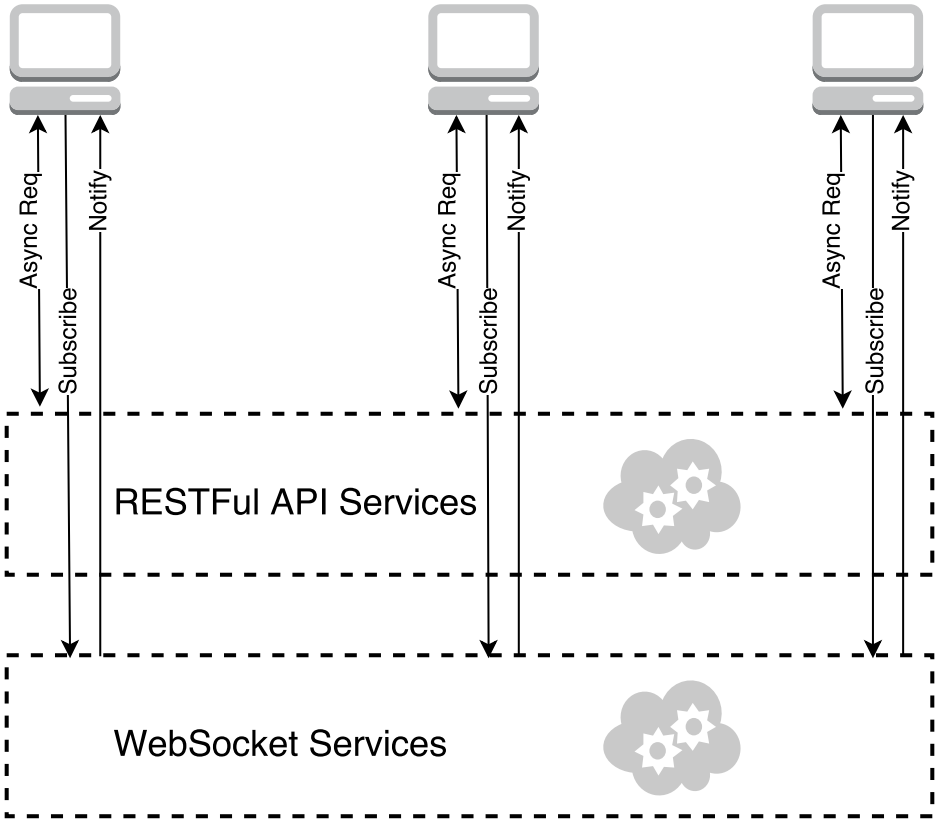

Figure 3.11.: General data communication

3.3.1. GENERAL DATA MODEL

DATA DOMAIN

In general, the data in the system could be divided into 4 primary domains:

- User: personal information as well as user identifier for accessing the system.
- Course: a container with own course information as well as a collection of questions classified in this context.
- Question: data with information of questions submitted by users.
- Answer: data with information of answers, also has graphical data within the data model.

Users are able to assess the contributions made by other users and mark it as useful or useless, which will affect the contributions' order priority while rendering the views. Voters should also be aware of what kind of vote he has marked to the contribution. Therefore, additional 2 data models VoteAnswer and VoteQuestion should also be considered.

The relation between domains is illustrated in the following figure 3.12. Each data model has a unique identifier, which is referenced in another data model when they have a connection.

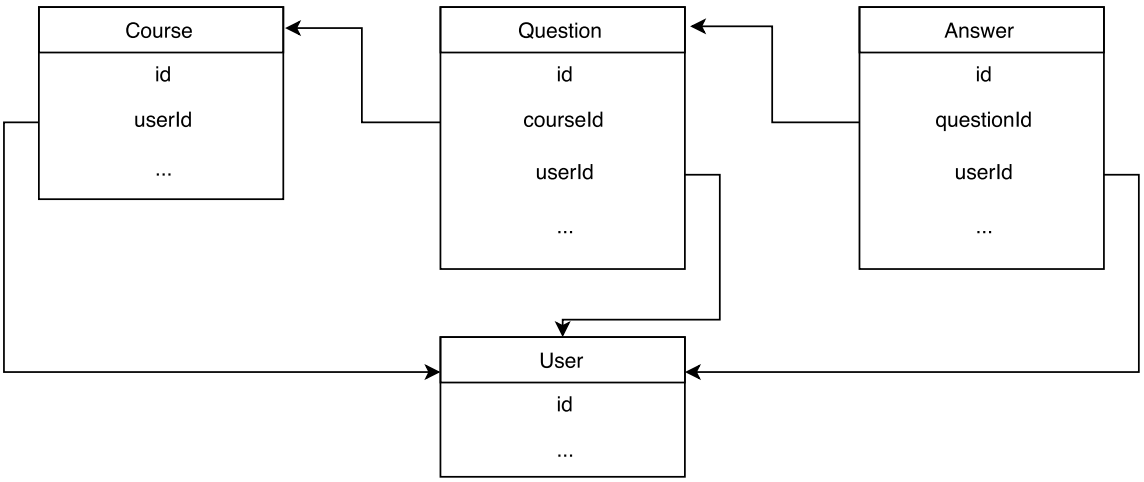

Figure 3.12.: Relations between data domains

DATA FIELDS

More detailed definition and description of fields in each data model should be made for a better understanding of the structure as well as the behavior of a data model. Table 3.1 describes key fields of general domains.

3.3.2. RESTFUL API DEFINITIONS

As mentioned in section xxx, RESTful architecture is a an excellent technical choice for data transferring. Because of its simplicity and clear semantic description of HTTP methods comparing to other protocols such like SOAP, it will dramatically simplify and clarify our data transmission services.

MAPPING OF HTTP METHODS TO DATA MODEL BEHAVIOR

The data model defined above can directly map to the definition of resources in RESTful. The HTTP methods on each resource domain can also represent the data model behaviors, *User Model* is taken as an example:

In table 3.2, resource entry in persistent storage can be executed with specific action while requesting resource URI through different HTTP methods. A semantic description of connection between CRUD and HTTP methods on RESTful will make the data transmission services more understandable and unified.

GENERAL RESTFUL API DEFINITIONS

Requesting a specific resource can only succeed through its URI, through which the client and server-side could connect to each other actually. Therefore, a definition of APIs which describes URI of the resources and its functional responsibility should be proposed in the first place.

Domain	Field	Description
User	email	unique identifier of a user for authentification, also as contact way to user.
	password	string for user authentication to prove identity or access approval
	username	user identifier shown to other users
	isTutor	flag which determines if the user is a student or tutor
Course	name	name of the course
	desc	description of the course
	creator	id of the user(tutor) who created this course
	code	unique code for the quick search purpose which is generated automatically as the course is created
Question	title	title of the question
	content	content of the question
	course	id of the course to which the question belongs
	vote	vote count of the question
	creator	id of the user who submitted this question
Answer	content	content of the answer
	question	id of the question for which the answer is made
	quoted	id of the original question which is quoted
	vote	vote count of the answer
	creator	id of the submitter
Vote	type	enum values of up-voting or down-voting actions
	handler	id of the handler
	question/answer	id of the question/answer to which the vote action is applied

Table 3.1.: Fields for Each Data Domain

Method	Operation of data model collection
GET	Query and return a specific user from the user model collection.
POST	Create a new user entry and insert into the user model collection.
PUT	Update a specific user in the user model collection.
DELETE	Delete a specific user in the user model collection.

Table 3.2.: HTTP methods on User resource

- **User Authentication**: the major actions of user authentication include signup, login, logout. To protect the user information, POST method which doesn't expose information via the URL, is highly recommended.

URI	Method	Description
/auth/login	POST	User login action, request with login information.
/auth/signup	POST	User signup action, request with registration information.
/auth/logout	GET	User logout action, no data submission is needed.

Table 3.3.: User Auth APIs

- **Courses**: acquisition of courses and new submission of a course is possible. In addition, CRUD operations on a specific course should also be achieved through a single URI with various HTTP methods.

URI	Method	Description
/courses	GET/POST	request the whole collection of courses; create a course with data submitted
/courses/:courseId	GET/PUT/DELETE	request, modify, remove specific entry of course

Table 3.4.: Course Resource APIs

- **Questions**: in a real sense question resource is attached to the course resource. According to the best practice of RESTful API design [reference xxx], question resource could be touched under course URI, */courses/:courseId/questions/:questionId*. But in the real world, question resource has its own collection, and *questionId* is the unique identifier, through which a specific question entry could be selected without using *courseId*. So an optimized conception is simply using */questions* as URI instead. And pass *courseId* as a query parameter while requesting collection of question entries under a specific course.

URI	Method	Description
/questions?courseId=:id	GET/POST	request the whole collection of questions belonging to a specific course; create question under a specific course
/questions/:id	GET/PUT/DELETE	request, modify, remove specific entry of question
/questions/:id/vote/:type	POST	vote actions with different vote types applied to specific question

Table 3.5.: Question Resource APIs

- **Answers**: the general API design of answer is totally same as the approach applied in question resource. A independent API for voting functionality should also be designed. And multiple possibilities of vote types could also be passed through the API.

URI	Method	Description
/answers?questionId=:id	GET/POST	request the whole collection of answers belonging to a specific question; create answer under a specific question
/answers/:id	GET/PUT/DELETE	request, modify, remove specific entry of answer
/answers/:id/vote/:type	POST	vote actions with different vote types applied to specific answer

Table 3.6.: Answer Resource APIs

Once all APIs with different HTTP methods are defined, a more concrete data structure over the APIs between two sides should be promised and confirmed. By following defined APIs and promised data structure, developments on both client and server-side could be executed parallelly.

3.3.3. WEBSOCKET DEFINITIONS

As the requirements defined in section 3.1, users could be informed as new question is posted or the order of answers with rating priority changes. Basically, only two different types of listeners are needed in this case: one for listening to the new questions under a specific class and one for responsive order of answers under a specific question. With WebSocket, URI should also be defined as an identifier for the persistent connection between client and server. And different events within a connection of one URI should also be designed. Table 3.7 defines these two WebSocket specifications.

URI	Event	Response
/ws/courses/	questions-changed	data of new question posted by others
/ws/questions/	answers-changed	data of answers in new order

Table 3.7.: Answer Resource APIs

3.4. GRAPHICAL DATA SERIALIZATION

Graphical content is the most efficient and intuitive way to deliver the explanation of an answer to other users comparing with pure textual content. In this section, Choices for graphical technologies on the Web will be analyzed in order to figure out which is the ideal and fit the graphical discussion system most. And a feasible approach of storing graphical data will be proposed. At last, a drawing tool will also be designed to offer user interfaces for drawing elements on the drawing board.

3.4.1. CANVAS OVER SVG

It seems that both Canvas and SVG are good candidates as a graphic technology for the graphic discussion system. Both of them provide native methods to render rich varieties of elements like the path, circle, rectangle and so on. Which would be a better choice above the context of discussion system, should be analyzed at first.

As mentioned in section 2.2.3, the rendering efficiency is the primary deficiency happening to SVG.

And graphic technologies are not only simply used for rendering a static graphical content in the system. Dragging, resizing or deleting an element are the basic features of a drawing tool, which provides input of graphical content. All these features are only able to be accomplished by re-rendering the elements on the drawing board.

Considering that all contributions in the system are made with graphical contents, efficiency plays a significant role, especially on mobile devices with old hardware. Choosing Canvas will give users better usability while viewing the contributions as well as using the efficient drawing tool without janky feeling.

3.4.2. STORABLE AND REVERSIBLE CANVAS DATA

A focus point in the thesis is how to store the graphical content submitted by users. In the traditional way, graphical data could only be stored either in the file system or in the database with encoded format, for example Base64 encoded images.

DEFICIENCY OF CANVAS - STORABLE IMAGE DATA

Canvas provides a native method called *getImageData()* to export the whole Canvas context, including the size of Canvas and pixels on Canvas to an image data. The image data exported by canvas represents the underlying pixel data with the format of *Uint8ClampedArray*. The *Uint8ClampedArray* typed array represents an array of 8-bit unsigned integers clamped to 0-255 [reference], which implies the position of the pixel as the index of array and the color of pixel as value from 0 to 255. An example is taken in figure 3.13 shows a Canvas with size of 100x100 and its simplified ImageData.

Restoring with the exported ImageData is also possible in Canvas by using its native method called *putImageData()*. Basically, the concept of *putImageData()* is traversing the ImageData exported by Canvas, and re-drawing each pixel at the position according to the index in the *Uint8ClampedArray* and applying the color to the pixel based on the value from 0 to 255 stored in the array.

Natively exported result of image data could basically meet the storing demand, however, the data redundancy in the native exported format representing the properties of each pixel is still very huge. Storing such kind of data will cause high demand on storage space when plenty of graphical contributions are made in their system.

Additionally, it is expected that users are able to remove, resize and modify the elements in the canvas while quoting others' contributions. However, the natively exported result

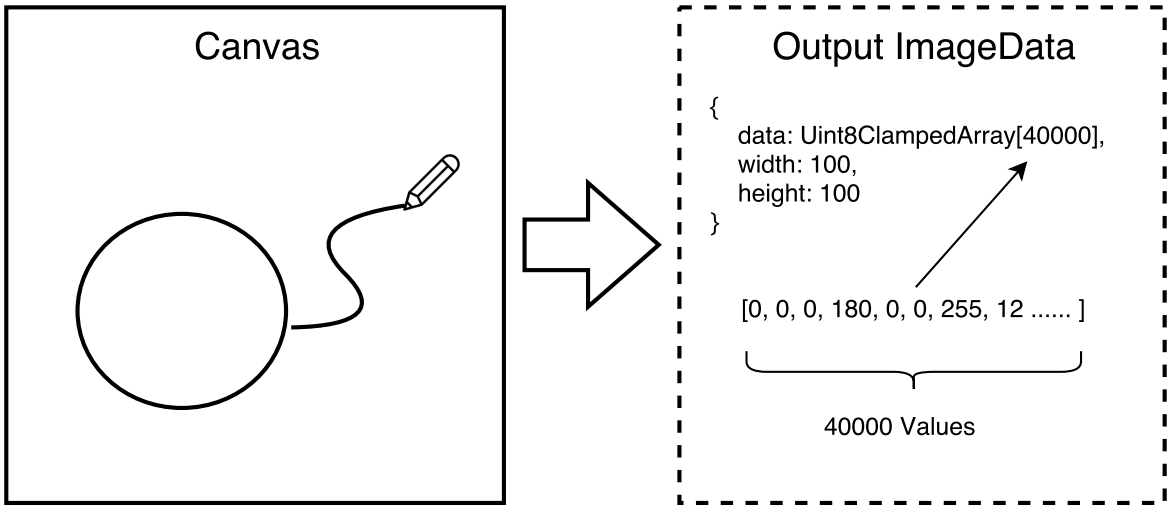

Figure 3.13.: Canvas to native ImageData

of image data could only describe each pixel, but not each element on the Canvas, which means the modification of the elements is not possible even though the whole canvas could be reproduced with graphical content by others.

Therefore, a workable solution should be concepted and new data model describing the graphical content in Canvas should be designed to meet the demands mentioned above.

SOLUTION - OBJECTIFICATION OF ELEMENTS IN CANVAS

Even though Canvas has native methods to draw different shapes of elements, but Canvas only renders them pixel by pixel, it knows nothing of the shapes that are drawn. Therefore, removal or modification of a already drawn element is not possible. In this condition, a feasible solution is to wrap the Canvas and objectify the original elements which could be stored and persisted in a stack. Furthermore, the wrapper on Canvas should also provide methods to render, modify, remove the custom elements.

The general conception of the objectified Canvas is illustrated in figure 3.14.

The objectified Canvas has a list of objectified elements which are visible on Canvas. Now there are new definitions for rendering, modification of an objectified element:

- Rendering Canvas: the list of elements maintained by objectified Canvas will be traversed and each element will be rendered by calling the native drawing method of Canvas. Position and style of the element in Canvas refer to the properties of its object.
- Modification or Removal: In case the methods for modifying or removing provided by element object in objectified Canvas are fired, the whole Canvas will be re-rendered in the same way of rendering the Canvas initially.

Now rendering means that the list of elements maintained by objectified canvas is traversed and each element is

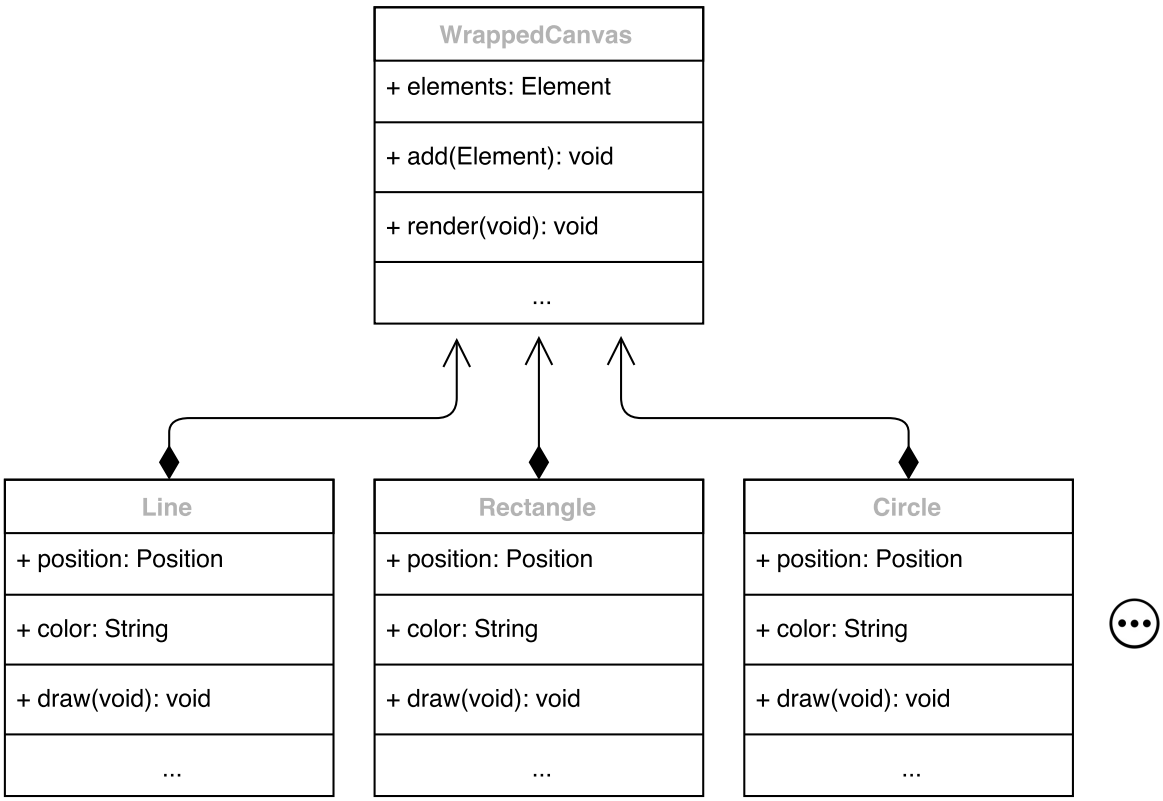

Figure 3.14.: Concept of objectified Canvas

DATA MODEL EXPORTED BY OBJECTIFIED CANVAS

Since the objectified Canvas maintains a list of objects for elements, which also contain the properties such as position, color, size and so on, so the exporting of image data is now really simple. A composition of all objectified elements' properties is already enough to describe the whole canvas. Figure 3.15 reveals the approach and data model output from objectified canvas.

After converting all elements in objectified canvas, the new data model is much more efficient for storing comparing to the raw image data. The objectified Canvas will also provide a method to restore the output data, create new objectified elements by giving the properties of the data model, and re-render the elements into original Canvas pixel by pixel. With this approach, the feature of modification on elements while quoting other contributions could also be achieved.

3.4.3. DRAWING TOOL

After the conception of a feasible objectified canvas as the base of the container for all elements, a drawing tool which provides user interfaces to draw variable shapes as well as texts should be designed in the next step.

First of all, user interfaces for selecting different drawing mode like drawing circle, rectangle or line should be designed. Buttons for toggling various drawing modes are the best choice in this case.

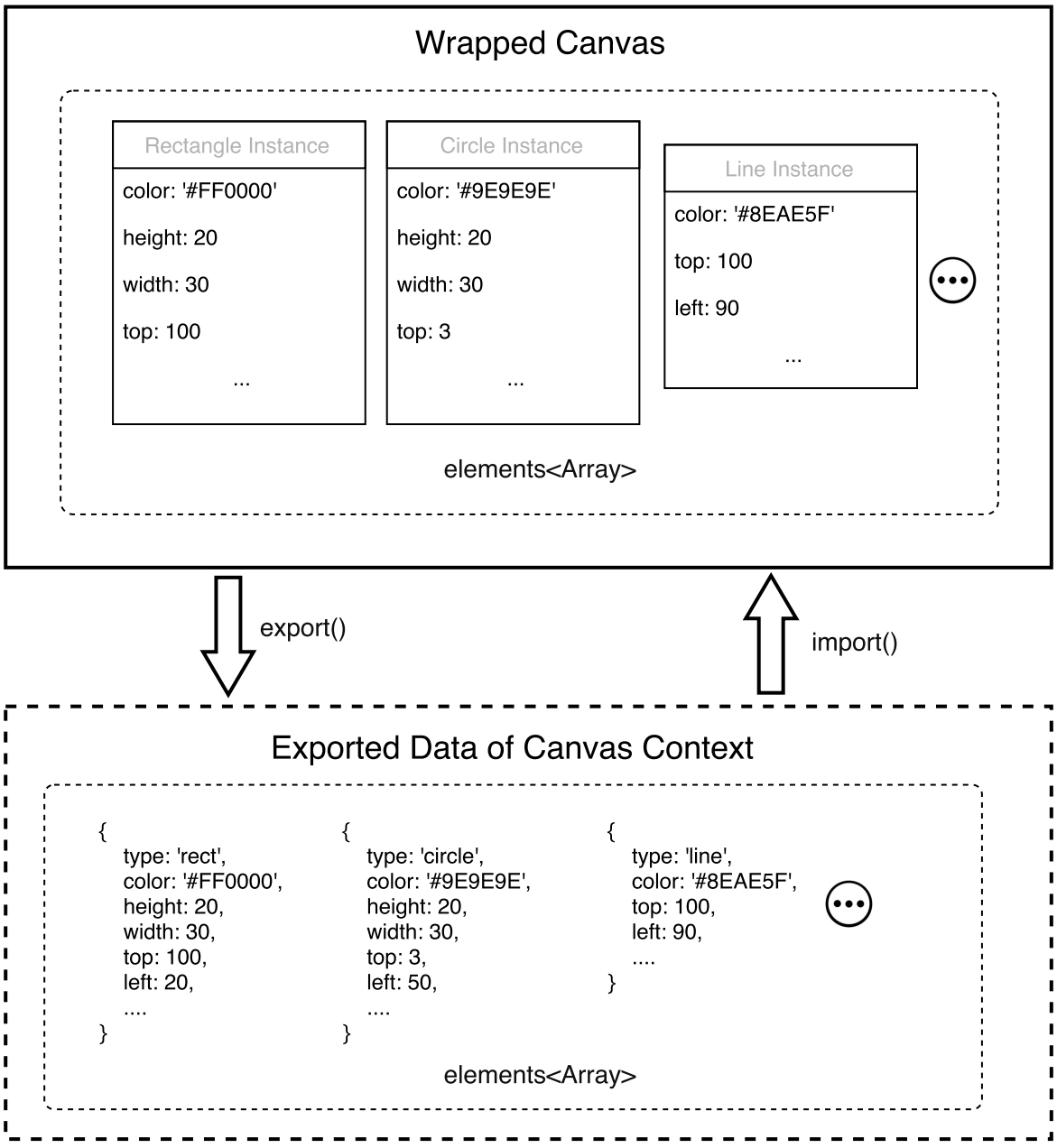

Figure 3.15.: Export and import of Canvas context with objectified elements

Because drawing elements in different shapes has distinct behaviors, so listeners for each specific drawing mode should be defined. When the button for toggling drawing mode is clicked by users and specific drawing mode is activated, the correlative listener will be initiated. Clicking events or moving events of the mouse on Canvas will be captured and processed. Meanwhile, drawing behaviors are also performed according the mouse events fired by users.

Figure 3.16 illustrated the conception of drawing tool with user interfaces and life cycle of event listeners for each drawing mode.

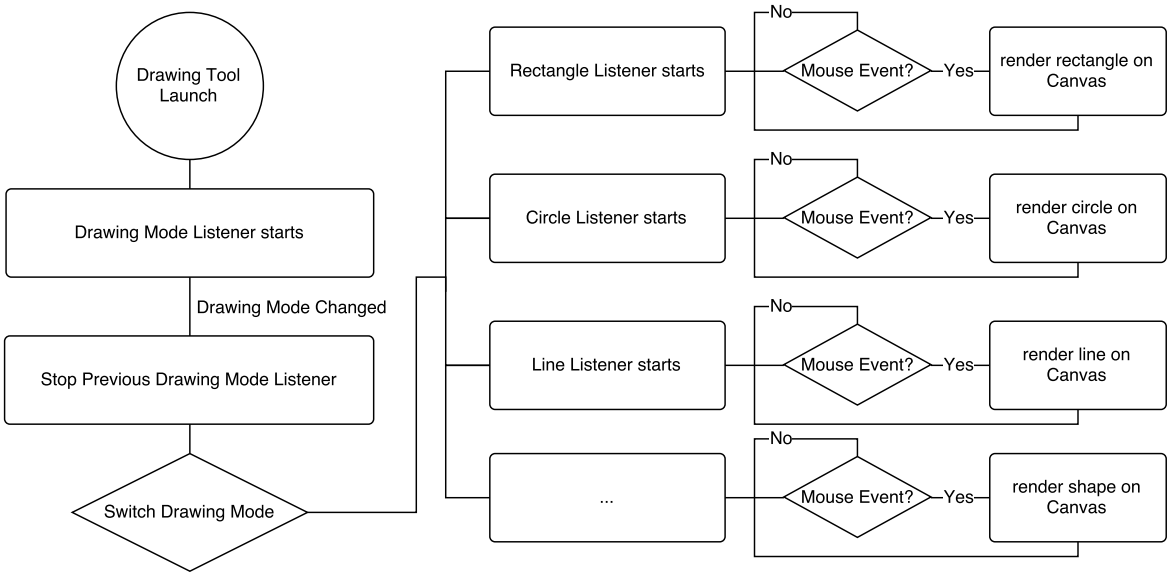

Figure 3.16.: Lifecycle of drawing tool

3.5. REAL-TIME DEMAND

Real-time communication as mentioned in subsection 3.2.2 is used for reactive data, which requires WebSocket for establishing persistent connections to enable the bi-directional communication between client and server.

All users are able to subscribe arbitrary course for new submission of a question as well as arbitrary question for updated order of answers, and server could also push real-time data to those users who has subscribed the resource with specific identifier. However, only two WebSocket services: */ws/courses* and */ws/questions* are defined as the entry points according to the definition in subsection 3.3.3. The approach how the server broadcast data precisely to the users who subscribe resources they require should be resolved.

A feasible solution is that the server maintains a list of user ids and resource ids subscribed by users. Afterwards, as a specific resource is updated, the server can get all users who have subscribed this resource from the maintained list using the resource identifier.

Figure 3.17 represents the whole workflow of establishing the connection over WebSocket. In the first place, the server starts listening for requests over WebSockets protocol with specific URI. Then the user starts a connection to server for subscribing the resource he requires. As soon as the connection is successfully established, the client will emit the

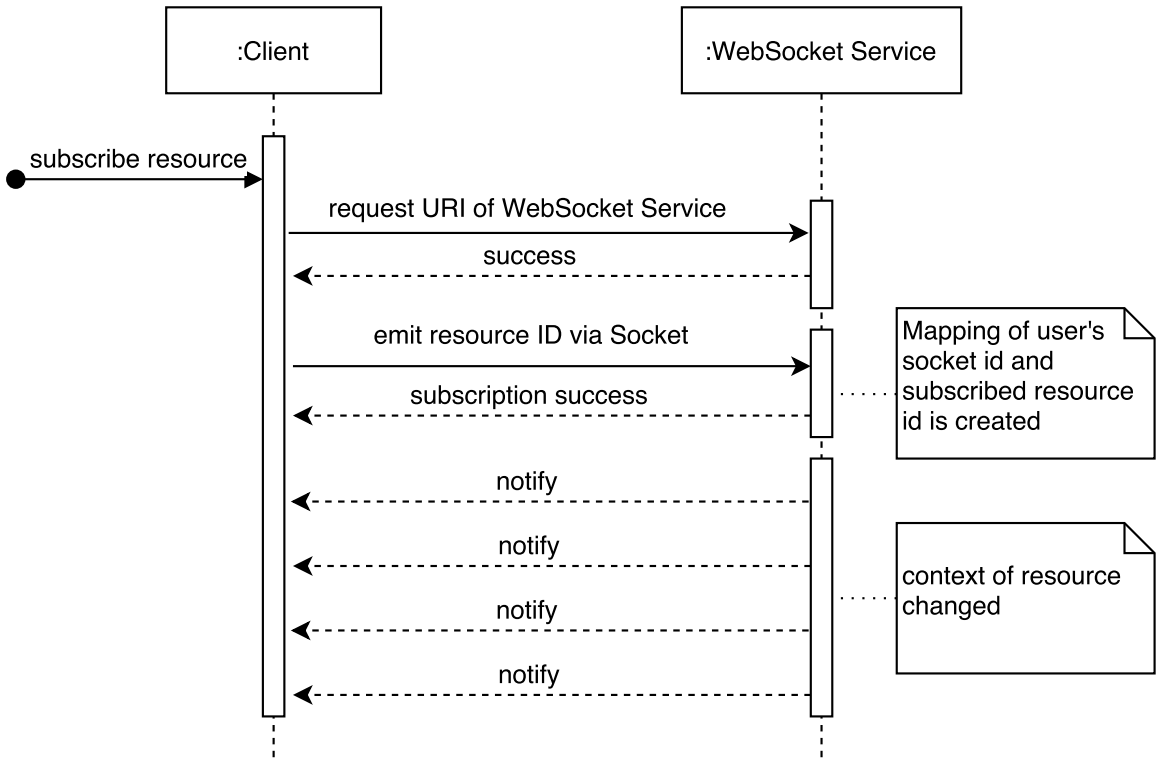

Figure 3.17.: Sequence diagram of establishing a WebSocket connection

resource id to the server side, and resource id from the client will be mapped to the user id in a list maintained by the server.

After that, the server has the information of which user has subscribed which resource, and is able to emit real-time data precisely.

4. IMPLEMENTATION

In this chapter, a prototype of graphical discussion system will be created and make use of the previously developed approach as a "proof of concept". Firstly, the implementation details of both client and server application are described. Afterwards, the implementation of a drawing tool with the functionality of drawing various shapes takes place.

4.1. GENERAL

On the whole, the implementation can be divided into two parts: client and server. Since they are fully separated, each part is considered and structured as an independent project. The implementation on the client side is basically data fetching and template rendering, while data persistence and core business logic is implemented on the server side. For convenience, the graphical discuss system is named "Graphicuss", which stands for graphical plus discuss.

4.1.1. PLATFORM AND FRAMEWORK

To achieve a better performance of view rendering on client side running in the browser, React.js[1] is taken as the front-end framework. Componentization, the main philosophy of ReactJS, also helps organize the views and view model logics. On the server side, ExpressJS[2] as a web framework is adopted for its efficiency and productivity of building RESTFul APIs.

Since both client and server projects are primarily implemented in JavaScript, NodeJS[3] is the single development environment for either project implementation or project management on both sides.

[1] https://facebook.github.io/react/ - accessed 10 July 2016
[2] http://expressjs.com/ - accessed 12 July 2016
[3] https://nodejs.org/ - accessed 12 July 2016

FILE STRUCTURE

To have a basic understanding of the whole project, including the server side and client side, a file structure of the project Graphicuss is listed in figure 4.1:

- client/: independent front-end project built on top of ReactJS
- server/: independent back-end project implemented by using ExpressJS
- dist/: compiled back-end project integrated with compiled and compressed static view files from front-end project
- node_modules/: source of referenced third party libraries
- package.json: definition of third party libraries for client and server side
- webpack.config.json: config of specific behaviors in automated development or building process

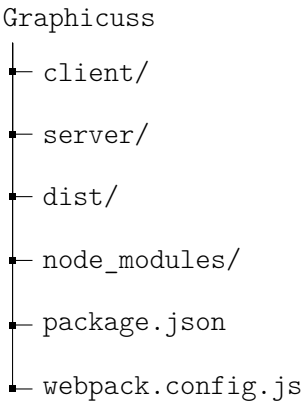

Figure 4.1.: Overview of Graphicuss' file structure

MODULE MANAGEMENT

Node.JS provides native module management, which is called *npm*[4]. Third party libraries, which are published in the official remote repository, can be installed conveniently by only using npm's command line. There is also a list of names of all modules and dependencies in the file *package.json*. Installing all dependencies and modules can be simply achieved by using only one command line *npm install*, which will significantly ease the setting up process of the project freshly on a new machine.

4.1.2. ARCHITECTURE

An architectural overview of Graphicuss is illustrated in figure 4.2.

While the client starts requesting a specific URL for data from the server, a HTTP connection will be established. The server program receives the HTTP request and forwards it to

[4] https://www.npmjs.com - accessed 12 July 2016

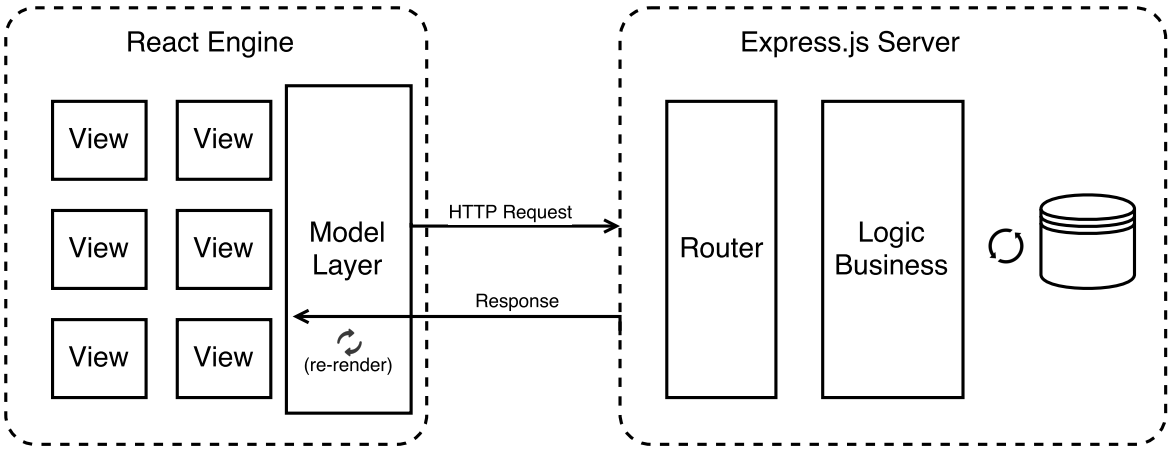

Figure 4.2.: General architecture

its router, in which rules for matching URLs have been pre-defined. By analyzing headers of HTTP request, the router will check if the request matches any pre-defined rules.

Not only the URL but also the parameters passed by the client, for example the identifier of a resource, could also be extracted from HTTP headers. The server runs correlate business logics according to the rule of matched URL and executes operations of databases for data persistence. Afterwards, the results are returned from the server.

As soon as the data is successfully returned, the HTTP connection will be closed. The client processes data acquired from the server, and represents it by re-rendering views. So far, an entire request over HTTP is accomplished.

4.1.3. AUTOMATIZATION

To accelerate the developing as well as building process of the project, an automatization tool called *Webpack*[5] is used.

Webpack is a tool which could analyze the dependencies of the project and bundle modules with the app. In addition, it can also do tasks like compressing JavaScript codes to reduce the size of the client app, or compiling modern JavaScript as well as CSS codes to achieve the compatibility for old browsers.

AUTOMATED DEVELOPMENT PROCESS

To make the development of client app independent, it will start a dev server on its own for development purpose. However the dev server started by client app and the actual server are running on different ports. Which means that the communication between them will cause CORS problem.

CORS means, a resource makes a cross-origin HTTP request when it requests a resource from a different domain than the one which the first resource itself serves. For security reasons, browsers restrict cross-origin HTTP requests initiated from within scripts. [12]

[5] https://webpack.github.io/ - accessed 13 July 2016

Through configuring the dev server started by Webpack, a proxy could be established to forward requests to the actual back-end server. As figure 4.3 shows, the client is able now to request APIs under the same domain, and requests will go though the dev server, after that they are forwarded to the actual server.

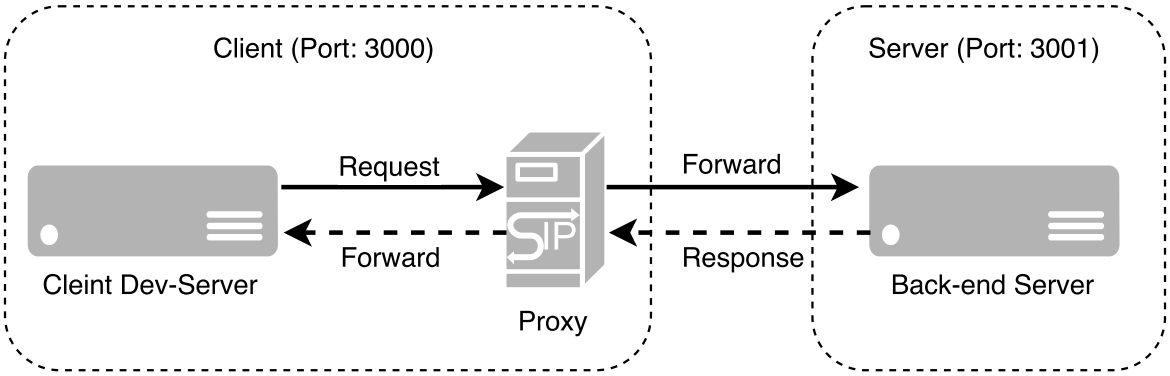

Figure 4.3.: Proxy for client development server

AUTOMATED BUILDING PROCESS

For the client app, multiple tasks are executed during the building process by using Webpack: transforming modern JavaScript code, pre-processing the modern CSS code and bundling the static files. All these tasks will significantly reduce the size of client app and also improve the compatibility of the app.

Webpack also helps accelerate the building process by defining various automated building tasks. It will bundle all dependencies with the server app and client app. After processing on both sides, the final output of the files will be extracted into the *dist* directory mentioned in 4.1, which is now ready for deploying and serving. The whole building process is represented in figure 4.4.

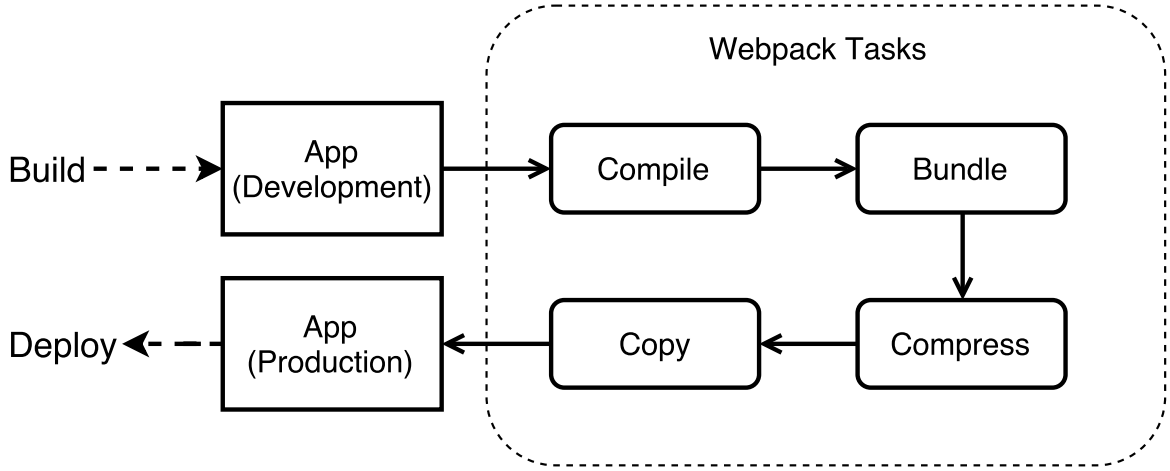

Figure 4.4.: Automated building process with Webpack

4.1.4. STORAGE STRUCTURE

In section 3.3, data domains and fields of data domains have already been defined.

For all data persistence storage on the server side a MongoDB[6] database is used. In figure 4.5, more concrete definition of the data table model is defined. MongoDB is a non-SQL database, which uses document oriented storage and JSON style data model. That will make it easy to implement as well as scale data models. In addition, An ORM framework called Mongoose[7] is also applied to the implementation, which encapsulates the native database operations of MongoDB. With the help of the ORM framework, definition of schema and query on database will be quite simple.

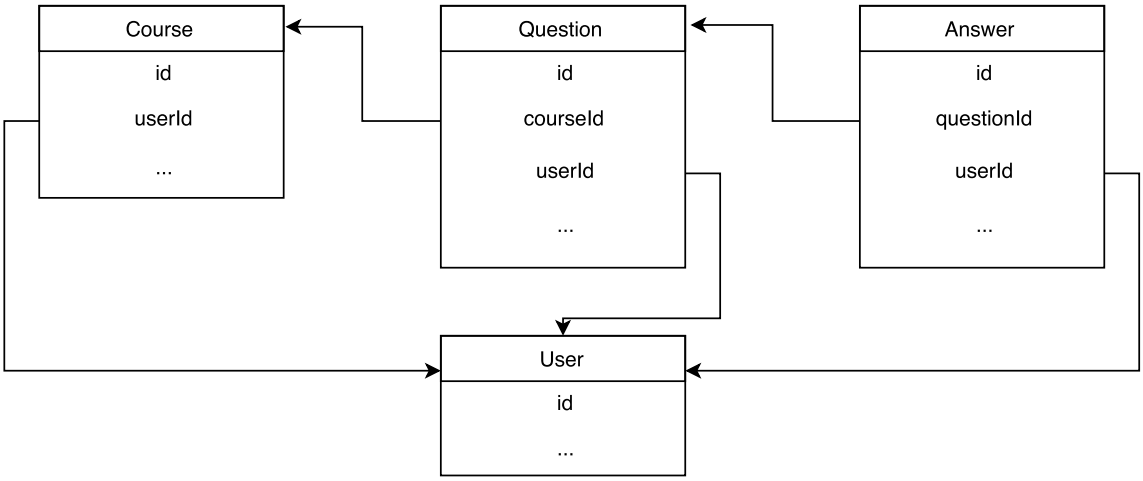

Figure 4.5.: Table of data model

4.2. SERVER OF GRAPHICUSS

4.2.1. ARCHITECTURE

MVC PATTERN & PROJECT STRUCTURE

To separate the different layers of model, view and controller, MVC pattern is used as the basic pattern of the architecture. In the model layer, all data model related concerns such as data schema definitions, data model validation as well as database operations are defined. And Controllers contain the core domain logics, process the data from model layer, and pass the result to view layer.

Since the templates are rendered on the client side, the view layer is just simply stripped. Therefore, basically the controllers response processed data to client side directly without rendering it to views. Figure 4.6 shows the overview of the server's file structure which is featured with MVC pattern.

[6] https://www.mongodb.com/ - accessed 13 July 2016
[7] http://mongoosejs.com/ - accessed 13 July 2016

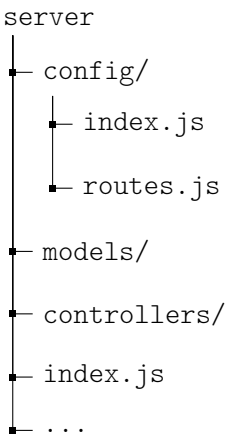

Figure 4.6.: Overview of server app's file structure

- index.js: the entry point of the whole server app. It will create a server instance and set up configurations for the server. In addition, a connection from server instance to database will be established. After all configurations are done, the server instance will start listening port and waiting for the requests from client.

- config/index.js: config as well as constants for the server. It persists *apiConfig* for example the common prefix of API URL and version of the API. And config for database including the database URL will be defined here as well. In addition, keys for encryption are also stored in the config file.

- config/routes.js: rules for URL matching. All URL matching rules are defined in this file. Controllers are referenced here and a dispatcher for router will be instantiated. If any request meets the defined rules, the request will be forwarded to a correlative controller.

- controllers/*: controllers for processing specific requests.

- models/*: data model definitions. Files under this directory are organized by different data domain.

ACHITECTURE OF SERVER

The figure 4.7 illustrates an overview of the server architecture. Requests from the client side are handled with *Router* component in the first place. HTTP headers are parsed and analyzed, afterwards, the request is dispatched by *Router* according to the URL matching rules and then forwarded to the correlative *Controller*.

Controller layer requests as well as operates the data records stored in database by using the methods defined in *Model* layer. As the request is successfully handeled and data is prepared, *Controller* responds with an HTTP response message to the client.

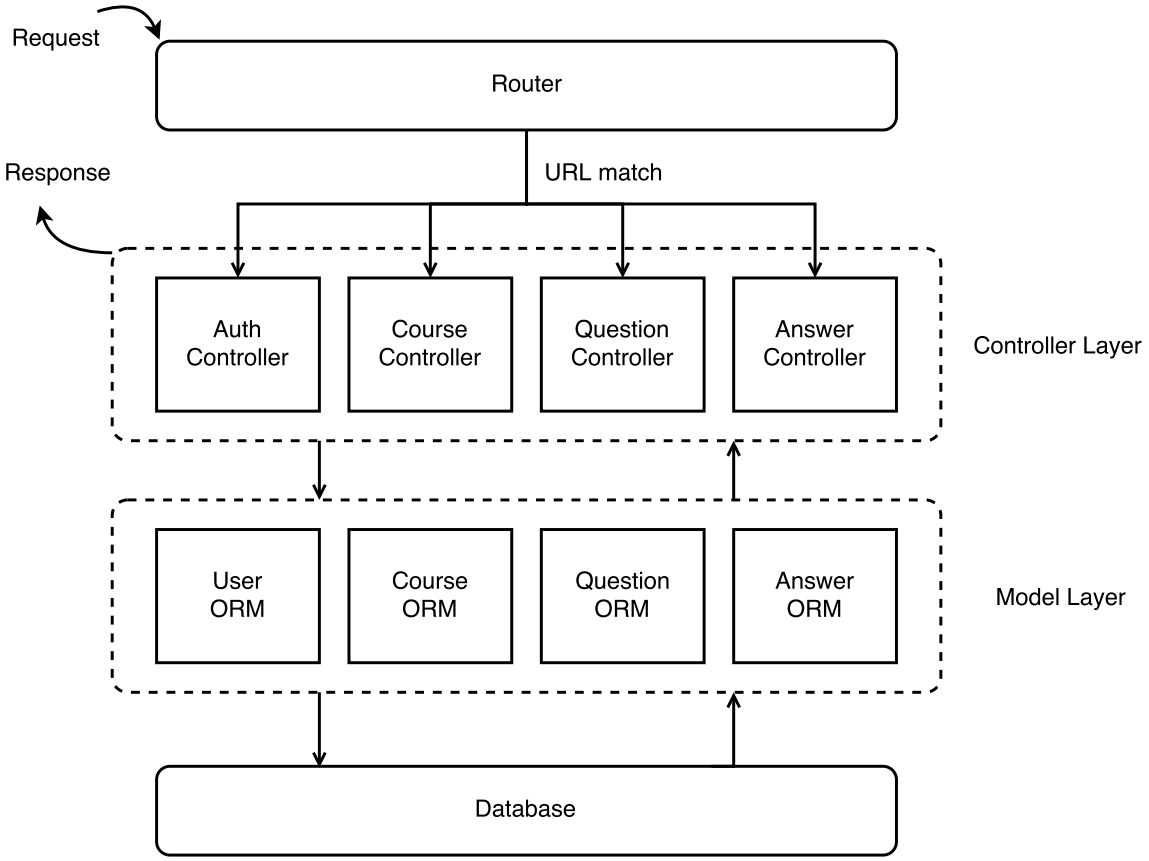

Figure 4.7.: Server architecture

4.2.2. MODEL LAYER IMPLEMENTATION

In subsection 4.1.4 an overview of the storage structure has been described. In following subsections, more concrete implementation of data model layer is explained and example codes are represented.

DATA MODEL SCHEMA

Not only the fields of each data model are defined, but also the data type of each field should also be restricted. Mongoose will check the type of fields within a data model according to the definitions before a record is inserted into the database.

```
1 import mongoose from 'mongoose'
2 const userSchema = mongoose.Schema({
3   username: String,
4   email: {type: String, lowercase: true, trim: true, unique: true},
5   password: {type: String, select: false},
6   faculty: {type: String, default: ''},
7   tutor: {type: Boolean, default: false},
8   admin: {type: Boolean, default: false}
9 }, {timestamps: true});
```

Listing 4.1: Example: user schema definition within Mongoose

Code list 4.1 takes *UserSchema* definition as an example. Fields like *username*, *email*, *password* and *faculty* are defined as *String* type. Fileds like *tutor* and *admin* are using Boolean type for determining the role of a user. In addition, the default value of a field could also be given if record is added without value in this field. And Mongoose also provides a set of configurations for processing the entry inserted. In the example, setting *lowercase* on *email* will transform all characters to lowercase, while setting *trim* will stripped space symbols out of a string. Mongoose will not only validate the type of data model, but also check the uniqueness of the data field in the entire database if *unique* is set to *true*.

METHODS ON MODEL LAYER

In most cases, various operations are executed to acquire data from the database, modify data and even remove data in the database. Therefore, these data fetching or data processing tasks are implemented in the data model layer, which will also achieve the goal of separation of concern. An example of *Course* data model is taken in the following code list 4.2.

```
1  courseSchema.statics.list = function() {
2    return this.find().sort('-createdAt').populate('creator').exec()
3  }
4  courseSchema.statics.load = function(_id) {
5    return this.findById(_id).populate('creator').exec()
6  }
7  courseSchema.method.edit = function(fields) {
8    return this.update(fields).exec()
9  }
10 courseSchema.method.remove = function() {
11   return this.remove().exec()
12 }
```

Listing 4.2: Example: user schema definition within Mongoose

Other data models like *Question* and *Answer* are quite same as *Course*. Method *list()* is defined for querying all records of a collection under the data domain. In addition, sorting and referencing other model with foreign key will also be done before returning the result. *load()* method is for querying a specific entry. Methods *edit()* and *remove()* means modification or removal of an entry in database.

4.2.3. AUTHENTICATION

The goal of authentication is confirming the identity of a user and controlling the access of resources by checking the privilege of a user. So an authentication system is also designed for security reasons.

JWT BASED AUTHENTICATION

JSON Web Token (JWT)[8] is an open standard (RFC 7519) that defines a compact and self-contained way for securely transmitting information between parties as a JSON object.

[8] https://jwt.io/ - accessed 15 July 2016

This information can be verified and trusted because it is digitally signed.

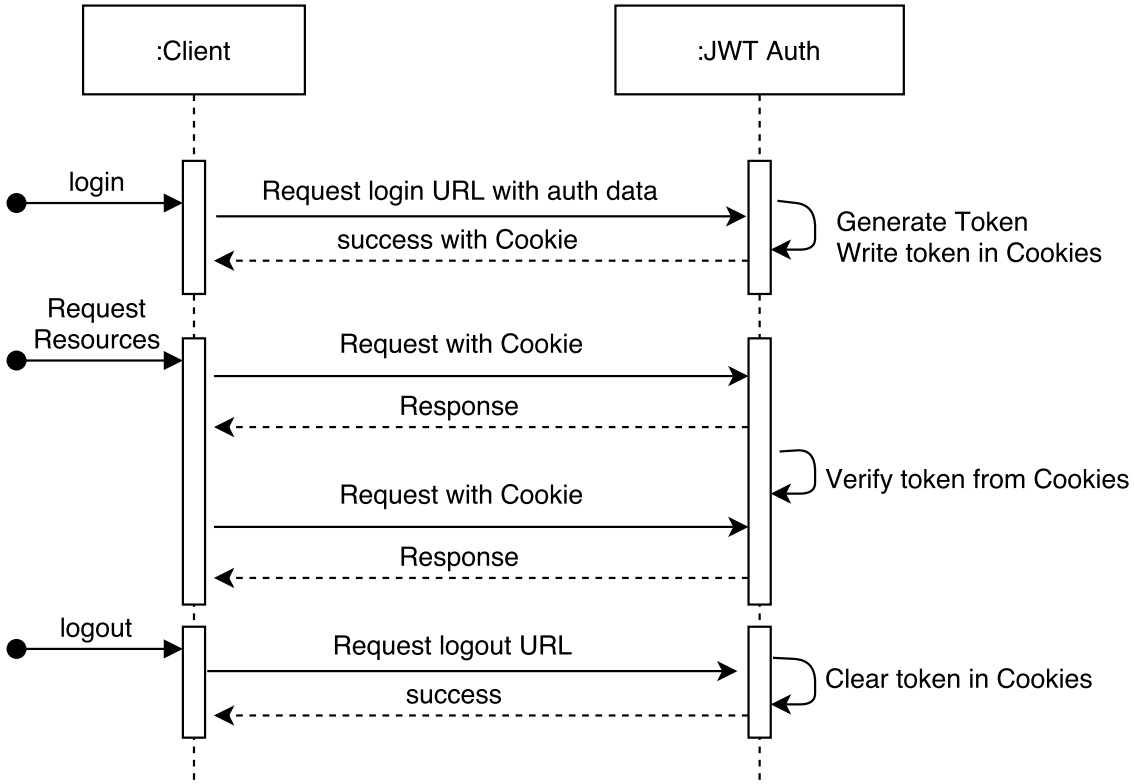

Figure 4.8.: JWT authentication process

The basic idea of JWT based Authentication in the server side of Graphicuss is shown in figure 4.8. After that the user sends a request to login with its identifier and password. After that the authentication is successfully verified, JWT will encode the user information with a secret key to a token.

```
1 var token = jwt.sign(userInfo, authConfig.jwtSecret)
2 response.cookie('token', token);
```

Listing 4.3: JWT encodes user information with secret key

After that the authentication is successfully verified and Cookies are written with JWT token, every request started from the client side will be sent with Cookies. All the requests will go through the *authMiddleware* and the tokens inside Cookies from requests will be decoded with the secret key. With the payload of user information which is decoded from the token, server is able to deal with resources for the specific user.

```
1 jwt.verify(token, authConfig.jwtSecret, function(err, decoded) {
2   var userInfo = decode
3 })
```

Listing 4.4: JWT decodes user information with secret key

ROLE CONTROL & ACCESS CONTROL

As defined in section 4.1.4, each user has two fields called *tutor* and *admin*. An admin has full control of all resources while a tutor is able to create a course and manage all resources

under his course. Same as a normal user without any privileges, he could only maintain resources submitted by himself.

So a *if* statement will be executed before the operations on resources in order to determine the role of the user, or to verify if a user has the access to the specific resource.

4.2.4. WEBSOCKET IMPLEMENTATION

For the implementation of real-time functionality, a library called Socket.io[9] which a fast and reliable real-time engine is used.

As described in section 3.5, at the start of server, an instance of Socket.io will be created for listening for WebSockets requests within a specific namespace. Following code list 4.5 shows the main process of listener created for real-time questions under a specific course.

```
1  var courseWS = io.of('/ws/courses/');
2  var courseSocketMap  = {};
3  courseWS.on('connection', function(socket){
4    socket.on('course-to-listen', function(courseId){
5      if(courseSocketMap[courseId]){
6        courseSocketMap[courseId].push(socket)
7      }
8      else{
9        courseSocketMap[courseId] = [socket]
10     }
11   });
12 });
13
14 courseWS.on('disconnection', function(socket){
15   Object.keys(courseSocketMap).forEach(function(courseId) {
16     var index = courseSocketMap[courseId].indexOf(socket)
17     if (index > -1) {
18       courseSocketMap[courseId].splice(index, 1);
19     }
20   });
21 });
22 /* --- Listening for Question is the same approach--- */
```

Listing 4.5: Server starts listening for requests over WebSocket protocol

After the WebSocket listener is started, it will monitor the the event called *course-to-listen*. As it is triggered, *courseId* received from client will be mapped to the a list of socket objects which represent the users who are listening to this resource. After the client disconnects from the WebSocket, his socket object will be removed from the listening list.

```
1 var sockets = courseSocketMap[courseId];
2 sockets.forEach(function(socket){
3   socket.emit('questions-changed', newQuestions);
4 });
5 /* --- Listening for Question is the same approach--- */
```

Listing 4.6: Server starts listening for requests over WebSocket protocol

[9] http://socket.io/ - accessed 15 July 2016

If the questions under the specific course are changed, all sockets which could be referenced from *courseSocketMap* by using the *courseId* will emit an event with payload of changed question resources. Those clients who has subscribed this course will be informed and receive the new questions passively. Code list 4.6 demonstrates the process.

The approach of real-time order of answers under a specific question is quite same as the approach mentioned above.

4.3. CLIENT OF GRAPHICUSS

4.3.1. ARCHITECTURE

For the implementation of the client application, *React.js* which aims to solve the challenges involved when developing web application with complex user interfaces, is applied. To have a better concept of how Graphicuss's client application is implemented, an overview of the file structure is listed in the figure 4.9.

PROJECT STRUCTURE

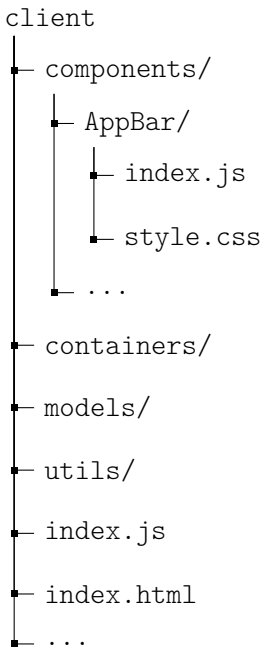

Figure 4.9.: Overview of client app's file structure

- components/: all components are defined by extending basic *React.Compoent*. Each custom component has an *index.js*, which processes the logics of view rendering and applies view models to the template. *style.css* defines the CSS style of the HTML DOMs within a component.
- containers/: containers are compositions of components.

- **models/**: in model directory, data models for the components are defined. In addition, the definition of APIs and processing after data acquisition also take place here.
- **index.js & index.html**: *index.js* is the entry point of the app, which will instantiate the React instance and render the views into a specific DOM defined in *index.html*

ACHITECTURE OF CLIENT

An overview of the client application's architecture is revealed in figure 4.10.

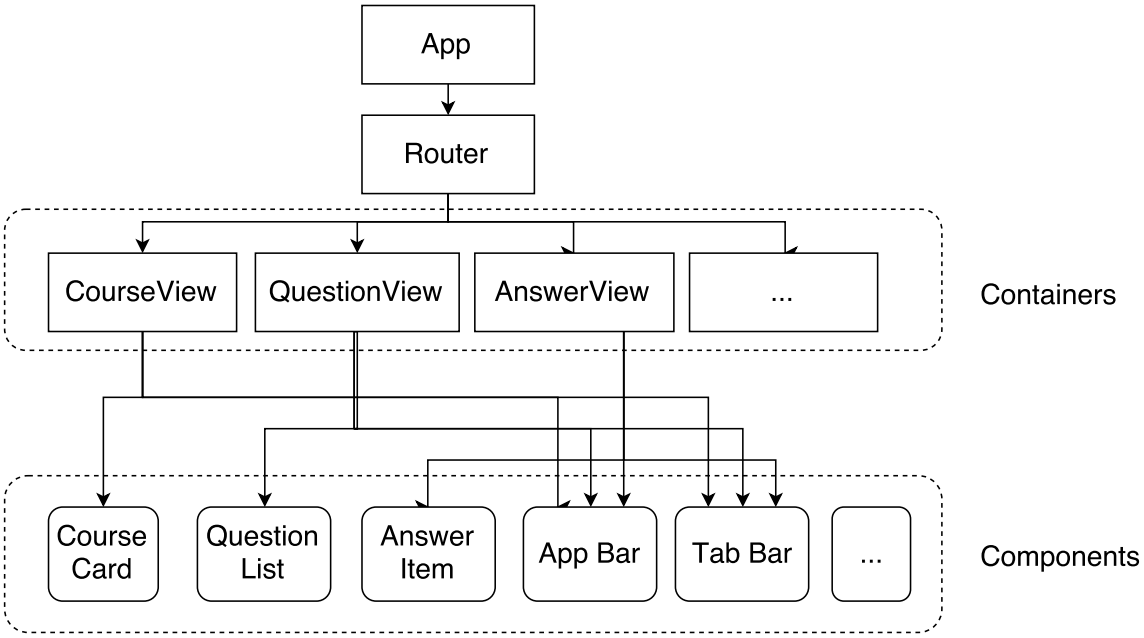

Figure 4.10.: Overview of client architecture

In the React application, *Router* is also regarded as a component, in which different matching rules of the URL are defined. If the URL requested by the user is matched, a correlate view will be rendered according to the definition of routes. Code list 4.7 shows the implementation of defining a *Router* component.

```
<Router history={history}>
  <Route path="/" component={App}>
    <Route path="auth" component={AuthView} />
    <Route path="courses" component={CoursesView} />
    <Route path="courses/:courseId" component={QuestionsView} />
    <Route path="questions/:questionId" component={AnswersView} />
  </Route>
</Router>
```

Listing 4.7: Router in client app

Containers such like *CourseView*, *QuestionsView* or *AnswersView* are compositions of components in fact. The way how component acquires view model is that the parent component passes values to its child component by defining the properties of the child component. So the data flow starts from the root component and goes through every child component.

How to manage the data flow and control the rendering behavior will be discussed later in the sub section 4.3.3.

4.3.2. COMPOSITION OF COMPONENTS

REACT COMPONENT

Defining a new *Component* with React.js must extend the *React.Component* class and implement the *render()* function, which will be called when the component is instantiated. Afterwards, the template as well as the composition of components are rendered to plain HTML. A simplified example of building a *CourseView* component is represented in code list 4.8.

```
1  class CoursesView extends React.Component {
2    render() {
3      const courses = this.props.courses
4      return (
5        <div>
6          {
7            courses.map( (course) =>
8              <CourseCard course={course} key={course._id}></CourseCard>
9            )
10         }
11       </div>
12     );
13   }
14 }
```

Listing 4.8: Rendering *CourseView* with multiple *CourseCard* components

As mentioned above, the parent components pass data through as the properties of child components. Within the *CoursesView*, *courses* could be read from its properties which is defined while *CourseView* is composed. In the *render()* function of *CourseView*, *courses* are traversed and each single *course* will be passed into the *CourseCard* as its property. Which means, as *CourseView* is rendered, all *CourseCard* components within it will also call their own *render()* functions with the data model passed in. Data flows from top to bottom, likewise, views are rendered from parent to child components.

COMPOSITION

Components are the core of React. All each view and its view model of the client application is represented as a React Component. And the whole client app is actually a composition of React components. An example of *CoursesView* is taken in figure 4.11.

At the top of the view is a component called *AppBar*, which is also composed with another component *SearchBox*. *ContentSection* is a container for the main content, which will be replaced and re-rendered if the context of router changes. In this example, the route */courses* is applied, and the component *CourseView* is rendered into *ContentSection*.

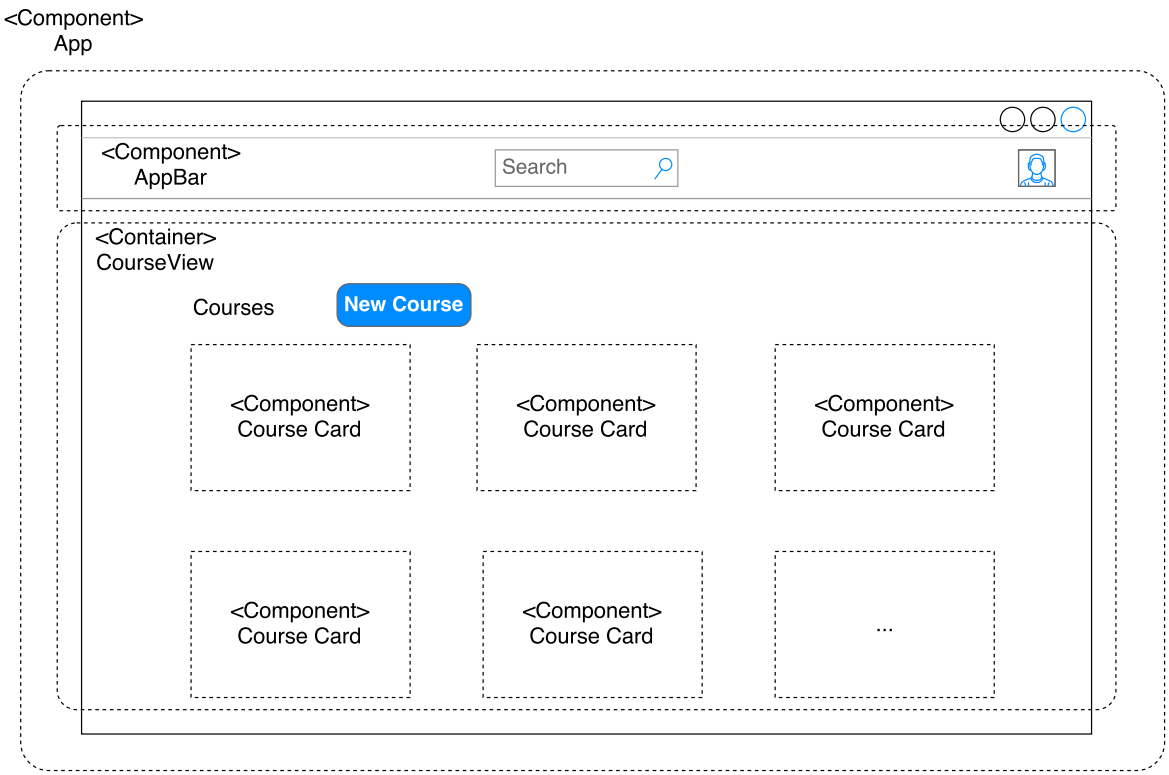

Figure 4.11.: Composition of components in courses' page

CourseView is also a composition of components: a list of *CourseCard* components and also other components such as submit button component and popover component for creating new courses.

In principle, building other views is the same approach. Composition of components constructs the all views. With fine-grained components, the client app becomes much extensible and maintainable.

4.3.3. DATA FLOW

Since data is passed as properties of components from top to bottom in the React application, maintaining data models between components and the data flow through components is a problem. *Flux*[10] is a architecture which aims to solve this problem. Data flows in a single direction, which keeps the process simple and ensures the correctness of view rendering.

There are four main concepts of Flux architecture:

- **View**: view layer which references the data model and renders the data model into the template.
- **Action**: action made by view, trigger for processing data model, for example a mouse click event.
- **Dispatcher**: receives the actions and run callback functions to modify the data model.

[10]http://facebook.github.io/flux/ - accessed 18 July 2016

- Store: stores the states of data, if the states of data are changed, store will notify the views to re-render with the new state of data.

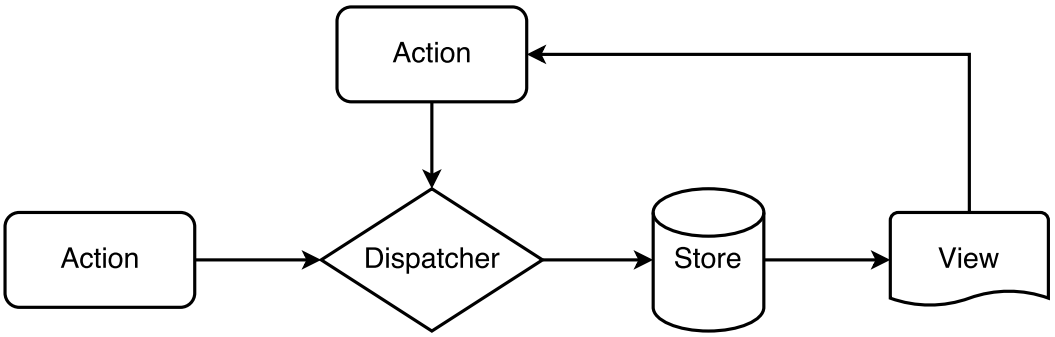

Figure 4.12.: Data flow in Flux architecture

The main process of data flow in Flux architecture is illustrated in figure 4.12. The key of Flux is unidirectional data flow. For example, a user wants to submit a new answer to a certain question in Graphicuss system, as soon as the new answer is synchronized with the server side, the new answer will be rendered into the answer list attached to the question. The process of data flow in this case is described as follows:

1. User submits a new answer, *Action(NEW_ANSWER)* is triggered.
2. *Action* requests the API for submitting new answer.
3. *Dispatcher* receives the *Action(NEW_ANSWER)* and inserts an entry of this new answer into the *answers* state which is stored in *Store*.
4. Since the state of *answers* is changed, *Store* starts notifying the views to re-render.
5. Views re-render the templates with the new state of *answers* which contains the new submission of answer.

4.4. DRAWING TOOL FOR GRAPHICUSS

In this section, the implementation of converting Canvas to a storable data with JSON format is introduced. Afterwards, a drawing tool which provides user interfaces to draw various elements on Canvas is implemented.

4.4.1. OBJECTIFIED CANVAS

FABRIC.JS

Fabric.js[11] is a powerful and simple Javascript HTML5 canvas library, which provides interactive object models on top of canvas elements. Since native Canvas only provides low-level APIs for creating elements, but not maintains the life cycle of elements on itself. Fabric.js solves this problem with objectifying native elements and encapsulating native methods for drawing elements.

[11] http://fabricjs.com/ - accessed 18 July 2016

Instead of dealing with low-level APIs natively provided by Canvas, Fabric.js provides objectified model for elements with different shapes on top of native methods. It takes charge of canvas state and rendering, make it possible to manipulate objects directly.

SERIALIZED CANVAS

Since all elements on the Canvas drawn by Fabric.js can be maintained as an object with properties like position, size and styles. So the Canvas within Fabric.js can be simply serialized to a JSON object or other formats.

Fabric.js provides a helper function called *toJSON()*, which will serialize the canvas with canvas properties as well as all object models on the canvas. Code list 4.9 is an example that shows how the serialized output looks like if a rectangle object is created by using Fabric.js.

```
1  var canvas = new fabric.Canvas();
2  canvas.backgroundColor = 'red';
3  canvas.add(new fabric.Rect({
4    left: 50,
5    top: 50,
6    height: 20,
7    width: 20,
8    fill: 'green'
9  }));
10 console.log(JSON.stringify(canvas));
11 /* --- Output of serialized Canvas ---
12 {"objects":[{"type":"rect","left":50,"top":50,"width":20,"height":20,"fill":"
       green","overlayFill":null,"stroke":null,"strokeWidth":1,"strokeDashArray
       ":null,"scaleX":1,"scaleY":1,"angle":0,"flipX":false,"flipY":false,"
       opacity":1,"selectable":true,"hasControls":true,"hasBorders":true,"
       hasRotatingPoint":false,"transparentCorners":true,"perPixelTargetFind":
       false,"rx":0,"ry":0}],"background":"rgba(0, 0, 0, 0)"}
13 */
```

Listing 4.9: Serialized Canvas by Fabric.js

Comparing with output generated by native Canvas mentioned in section 3.4, this serialized JSON object is not only efficient for storing, but also has the possibility for restoring all object models and re-rendering them on Canvas.

4.4.2. DRAWING TOOL

The drawing tool is developed on top of the library Fabric.js. It provides the functionalities such as drawing, styling, dragging and resizing of various elements. Not only graphical elements, texts could also be rendered and styled on the Canvas while using drawing tool.

Figure 4.13 illustrates an overview of the drawing tool's architecture.

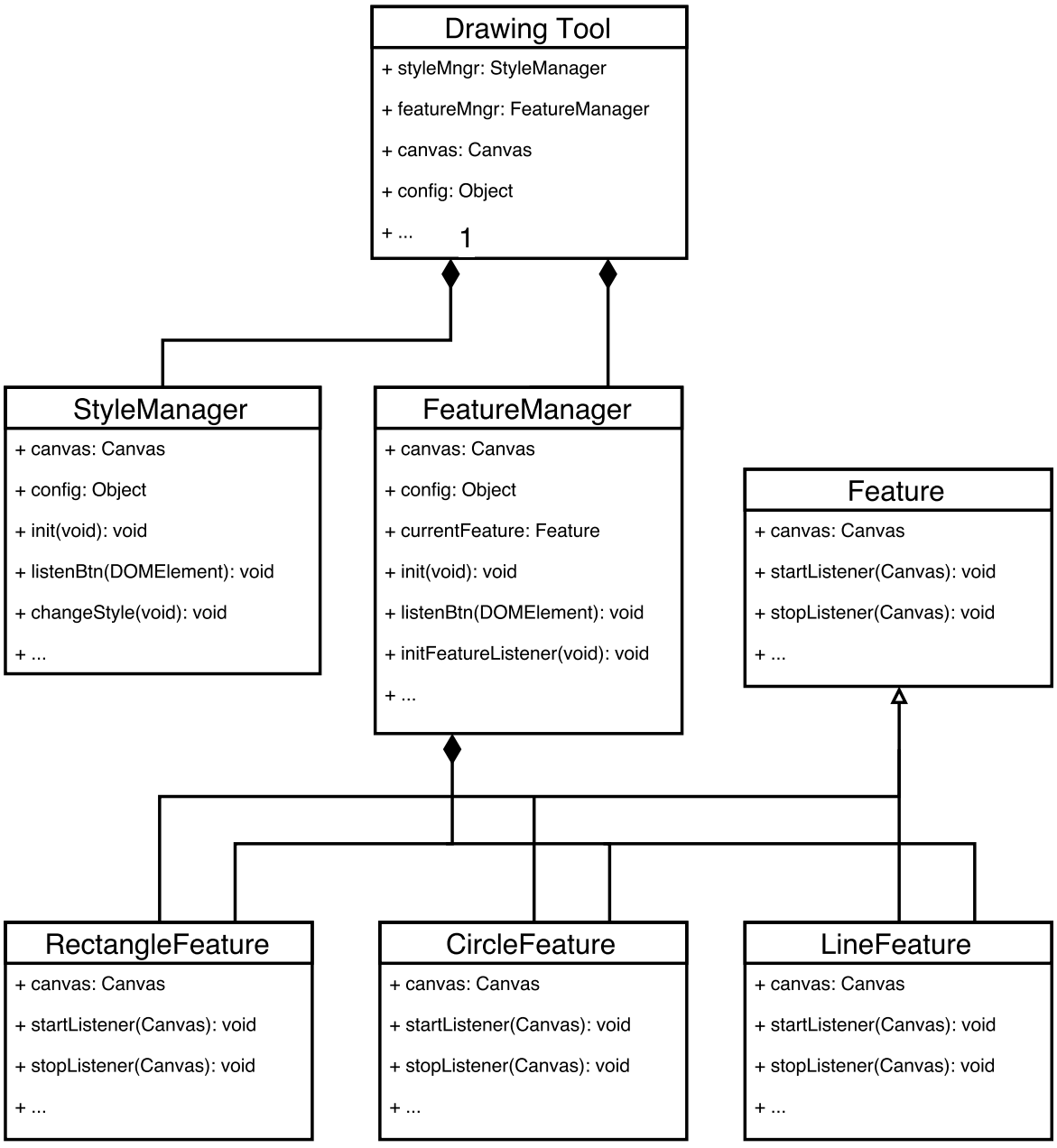

Figure 4.13.: Architecture of drawing tool

STYLE MANAGER

At the startup of drawing tool, *StyleManager* is instantiated. *StyleManager* receives the config instance, in which the DOM elements with relevant styling functionalities are defined. And in the *init()* function of *StyleManager*, listeners for the DOM elements are created. If the DOM elements are triggered by the user, *StyleManager* will apply the chosen style to the active objects on the Canvas.

Code list 4.10 takes the listener for DOM element of color picker, which is used for changing the color of an object on Canvas. It acquires the reference of color picker DOM from the config instance, and starts listening for the *onchange* event. If the *onchange* event is fired, the listener will set the object's *fill* property to the color value. Afterwards, the canvas is re-rendered and the active object with new color is shown up.

```
1  class StyleManager {
2    constructor(canvas, config){
3      this._canvas = canvas;
4      this._config = config;
5    }
6    init(){
7      el = this._config['color-picker-dom']
8      this._listenColor(el)
9      // ... more styling listeners
10   }
11   _listenColor(el){
12     let self = this;
13     el.onchange = function(){
14       var obj = self.canvas.getActiveObject();
15       self._setObjStyle(obj, 'fill', el.value);
16       canvas.renderAll();
17     }
18   }
19   // ...
20 }
```

Listing 4.10: Main process of StyleManager

FEATURE MANAGER & EXTENSIBLE FEATURES

In addition, a *FeatureManager* is also created for managing different drawing functionalities. The basic idea of *FeatureManager* is quite similar as *StyleManager* mentioned above. It also listens for the DOM elements to toggle different drawing behaviours.

After that a specific drawing mode is triggered by user, *FeatureManager* will assign listeners for specific mouse events on the Canvas and track the drawing behaviours. According to the mouse events triggered by user on the Canvas, the correlate objects will be rendered into the Canvas context.

```
1  class FeatureManager {
2    constructor(canvas, config){
3      this._canvas = canvas;
4      this._config = config;
5    }
6    init(){
```

```
7       el = this._config['text-feature-dom']
8       this._listenText(el)
9       // ... more features' listeners
10    }
11    _listenText(el){
12      let textFeature = new TextFeature(this._canvas);
13      el.onclick = (e) => {
14        this._clickHandler(text)
15      }
16    }
17    // ...
18  }
19  class TextFeature{
20    constructor(canvas){
21      this._canvas = canvas;
22    }
23    startListen(){
24      // tracking mouse event
25    }
26    stopListen(){
27      // remove listeners
28    }
29  }
```

Listing 4.11: Main process of FeatureManager

Features, namely drawing modes are highly extensible on the drawing tool. *TextFeature* in code list 4.11 is an example. All feature classes need to implement two interfaces *startListen()* and *stopListen()* basically. In *startListen()*, listeners for tracking mouse events are defined. And in *stopListen()*, all listeners should be removed. Both functions will be called by *FeatureManager* when this drawing mode is toggled.

5. EVALUATION

After the development approach has been motivated, designed and implemented in the previous chapters, an evaluation for both usability and data model will take place in this part.

5.1. USABILITY

Usability testing refers to evaluating a product or service by testing it with representative users. In principle, during a usability test, participants will evaluate the system with quantitative metrics.

The goal of usability testing is to collect the quantitative data, analyze the result and issue the usability problems with tested system.

5.1.1. SYSTEM USABILITY SCALE

The System Usability Scale (SUS) offers a "quick and dirty", but relative reliable approach for measuring the usability[13]. It contains a 10 item questionnaire with five rating options for participants; from *strongly agree* to *strongly disagree*.

In order to calculate the SUS score, score contributions from each item should be calculated once separately at first. The score contribution will range from 0 to 4. For items with odd number, the score contribution should minus 1. For item with an even number, the contribution is 5 minus the score. The sum of all scores is multiplied by 2.5 to obtain the overall value of SUS, which has a range of 0 to 100.

$$SUS_{sum} = 2.5 \times \left(\sum_{i=1}^{5} (a_{2i-1} - 1) + \sum_{i=1}^{5} (5 - a_{2i}) \right) \tag{5.1}$$

For the SUS testing of Graphicuss system, 5 participants are involved in the interview with SUS questionnaire, which is listed in appendix A. Each participant gives his own score

Item(No.)	A	B	C	D	E	Average Score
1	3	4	3	4	4	3.6
2	2	1	1	1	2	1.4
3	5	4	4	5	4	4.4
4	1	2	1	2	2	1.6
5	4	4	3	4	5	4
6	4	5	4	3	4	4
7	5	4	4	5	5	4.6
8	3	3	2	1	3	2.4
9	5	4	3	4	5	4.2
10	2	3	2	1	2	2

Table 5.1.: Score of SUS table

contribution for each item, and the average score of each item is calculated. Table 5.1 shows the result.

According to the formula 5.1, the final sum SUS score of the system is **73.5**. An article represents the mapping of adjective ratings to SUS score[14]. And a **73.5** SUS score achieves the rating in a range of *Good* to *Excellent* when it is expressed by adjective ratings.

The result reveals that the Graphicuss system achieves a relative high score in the general usability test. In general, users are able to learn to use this system very quickly. Without significant help, they can operate the system smoothly and unproblematically.

5.1.2. INTERVIEW BASED USABILITY TEST

Interviews with college students have been conducted in order to evaluate the system's usability as well as the fulfillment of requirements.

Nielsen had a research about the relationship between the amount of test users and the percentage of problems they found. Figure 5.1 shows the result[15]. Only 5 participants are already enough for discovering 75% of usability problems in most cases. Therefore, for the interview of evaluation, 5 participants are invited. The key parameters of the interview are listed as follows.

- Interview type: Discussion and questions to be answered using a 5-point Likert scale with additional space for comments and feedback.
- Number of questions: 10
- Duration of each interview: 20-30 minutes
- Time period of the conduction: 27th June 2016. The question sheet is attached to this thesis in Appendix B.
- Interview conduction:
    1. Several courses are created at the very beginning before the interview is performed.

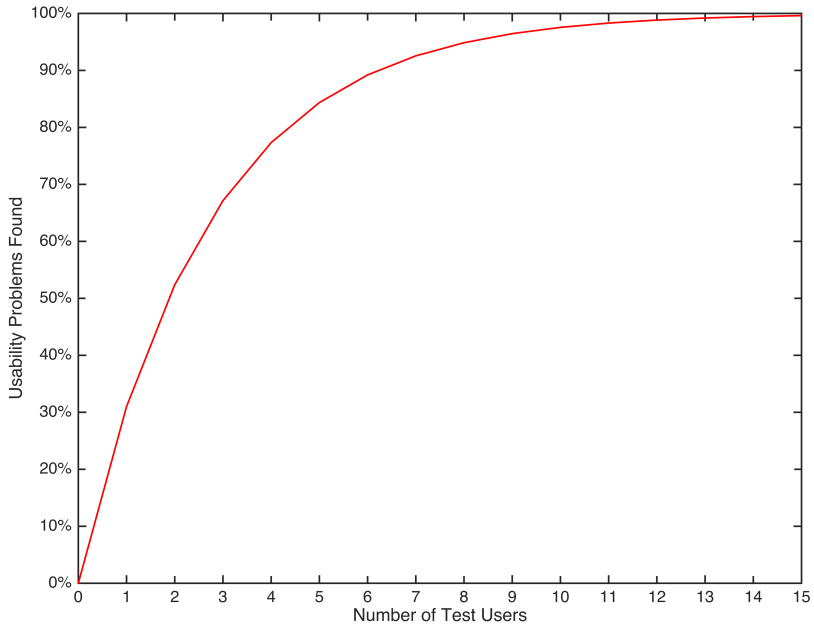

Figure 5.1.: Proportion of usability problems in an interface found by heuristic evaluation using various numbers of evaluators

2. After the interview is started, interviewees are requested to sign up with their own accounts and search the certain course by the course code.
3. Afterwards, they start questioning or answering within the certain course at the same time.
4. Interviewees are also demanded to use the major functionalities of the system, especially the drawing tool and quote functionality.
5. At last, the remaining 8 questions have been answered by the interviewees, followed by a final discussion of the results.

The overall results of the interviews are presented in table 5.2, which contains the score rated by each interviewee and the average score of each question.

ANALYSIS OF GENERAL FUNCTIONALITIES

Question 1-3 are designed for evaluating the main workflow of the system. Relative high scores are made by interviewees while testing the main functionalities of the system such like searching for a certain course, submitting questions and answers.

One tester has raised the issue that the auto-generated code (e.g. *dogP_lz8*) for querying the certain course is a little bit complex. Instead of a complex string with both alphabets and symbols, a simplified code which only has numbers would be accepted. In addition, interviewee C noticed that a pagination of the questions' list is required if the amount of question is getting greater.

Item(No.)	A	B	C	D	E	Average Score
1	5	4	4	5	5	4.6
2	4	5	5	4	4	4.4
3	5	4	4	5	4	4.4
4	4	3	4	3	4	3.6
5	4	3	3	3	2	3
6	4	5	4	5	4	4.4
7	5	5	5	5	4	4.8
8	2	3	3	2	2	2.4
9	3	4	5	4	3	3.8
10	2	3	1	2	1	1.8

Table 5.2.: Score of each question in the interview

ANALYSIS OF DRAWING TOOL

Question 4 is proposed to investigate the usability of the integrated text input in the drawing tool. Most interviewees are generally satisfied with the basic functionality of inputting a text string using the drawing tool. Tester B figured that the more stylings on the text should be implemented.

Most interviewees thought that the preset of the default elements were far not enough, which could be concluded from the Question 5. More shapes, which could be selected and drawn instantly, are highly required to be added into the preset. Otherwise, with the current elements of drawing shapes with the drawing tool, the expected graphical content is not able to be expressed precisely.

The history functionality of the drawing tool is productive according to the score rated in question 6. Undo/Redo function really helps the interviewees to correct the mistakes they made while using the drawing tool. Question 7 with the highest score shows that the modification on top of quoted content is convenient and useful.

ANALYSIS OF REAL-TIME FUNCTIONALITY

The real-time functionality is also investigated during the interview. All interviewees pointed out that the auto-ordering of answers is not seamless and inconspicuous, which could also cause distraction while viewing the answers. Therefore, the scores of question 8 and 10 are quite low.

However, the majority of interviewees has the opinion that the real-time feature will significantly improve the interactivity of the system despite of the distraction caused while auto-ordering is performed.

5.2. DATA MODEL EVALUATION

5.2.1. EVALUATION APPROACH

COMPRESSED IMAGE DATA FROM CANVAS

In the section 3.4 the raw output from native Canvas is briefly described. The data exported by the method *getImageData()* contains all values of each pixel without any compression. However, native Canvas also provides a method called *toDataURL()*, which is able to export the image data with the specific image format. The default type is *image/png*.

As a result, using *toDataURL()* could achieve the compressed image of a Canvas. Since the image format *PNG* uses lossless compression, which combines the LZ77-based DEFLATE algorithm[1] with a selection of domain-specific prediction filters, *PNG* format is chosen as the format of output image data from Canvas for the evaluation[16]. An example of the image data exported by calling *toDataURL()* is shown in code listing 5.1.

```
1 var canvas = document.getElementById("canvas");
2 var dataURL = canvas.toDataURL();
3 console.log(dataURL);
4 // "...
     ADElEQVQImWNgoBMAAABpAAFEI8ARAAAAAElFTkSuQmCC"
```

Listing 5.1: Example of image data while calling toDataURL()

MEASUREMENT PROCESS

The serialized graphical data model adopted in Graphicuss system is presented in section 4.4. To evaluate the data space of each data model, the metric *Byte* is taken for representing the length of data. In addition, two different dimensions are considered to evaluate the consumption of data space.

- **Size of Canvas**: two different sizes *800*600* and *400*300* of Canvas are defined.
- **Amount of Components**: various amounts of components are drawn on the native Canvas and objectified Canvas in two different sizes.

The comparison is following the measurement process as below:

1. Native Canvas or objectified Canvas with a certain size is initialized.
2. 3 methods for drawing elements *Rectangle*, *Circle* and *Line* in black color are predefined, which will be randomly chosen during the evaluation.
3. Random components from the preset are selected to be drawn at a random position with a random scale on the Canvas.
4. The length of the output data model is recorded for each amount of components in a range of 0 to 2000

[1] https://en.wikipedia.org/wiki/DEFLATE - accessed 119 July 2016

In general, four tests are performed for both types of Canvas in two different sizes. Figure 5.2 illustrates the results of the test.

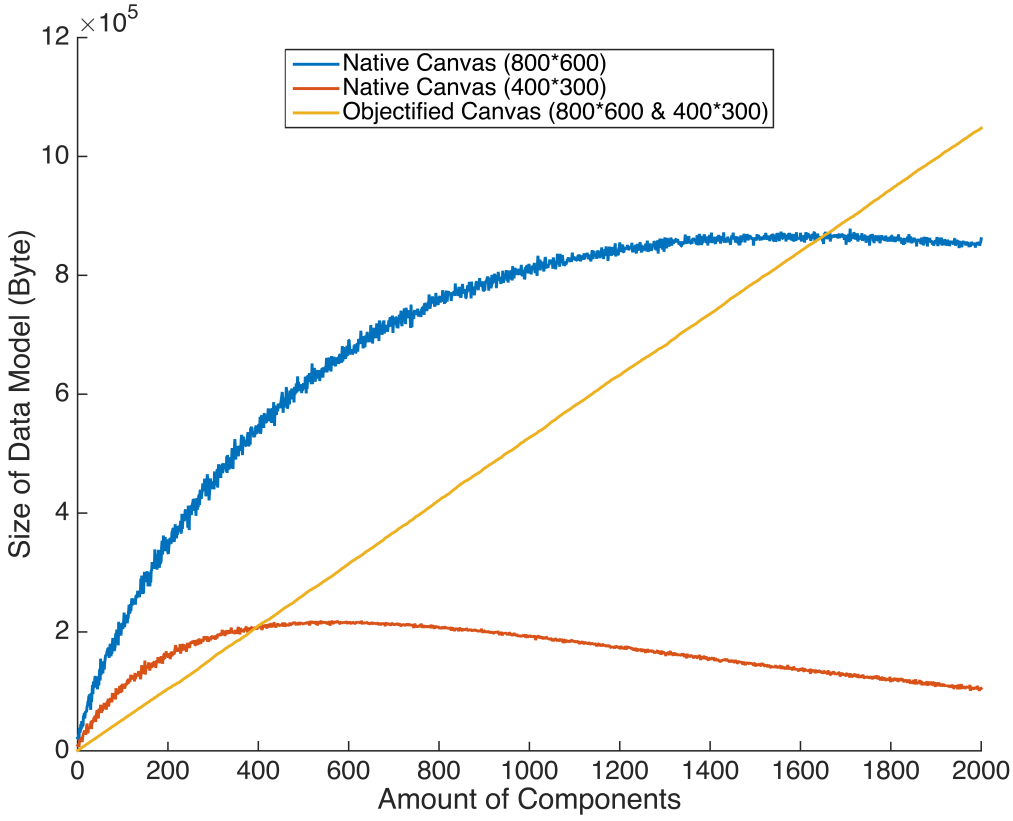

Figure 5.2.: Comparison of data spaces of image data from native Canvas in two sizes(800*600, 400*300) and adopted graphical data model in Graphicuss

5.2.2. ANALYSIS OF RESULT

Though four tests are executed, the figure 5.2 shows only three curves or lines. The reason is that the results of objectified Canvas in two sizes are nearly same and overlap themselves on the figure. So the yellow line represents both results of objectified Canvas in two sizes.

In overview, the data size correlates with the size of Canvas and amount of components drawn on Canvas. Therefore, both dimensions are analyzed separately in the following part.

DIMENSION: SIZE OF CANVAS

Size of data model from objectified Canvas is irrelevant to the size of Canvas. The yellow line on the figure shows both that results exported in different Canvas' size are generally same. Since the data model of objectified Canvas represents the components not pixels, the result is as expected.

Size of image data from native Canvas has a high-positive correlation with the size of Canvas. As the output image data from native Canvas describes each pixel of the Canvas,

size of the Canvas is one of the most important factors which affect the size of image data.

Max size of image data from native Canvas is approximately linear to the size of Canvas. The max size of image data from the blue curve is about 8×10^5 Byte , while the max size from the orange curve is nearly 2×10^5 Byte, which means, the former is 4 times as great as the latter. The size of *800*600* is also 4 times greater than the size of *400*300*, which also proves that the output image data describes each pixel.

DIMENSION: AMOUNT OF COMPONENTS

Size of data model from objectified Canvas is linear to the amount of components. Because the data model from objectified Canvas is a composition of components' description, the data size increases linearly with the amount of components.

Within a certain amount of components, the distribution of image data's size from native Canvas is logarithmic. As the test shows, if less than about 1400 components are drawn on the native Canvas in size of 800*600, or less than 500 components in size of 400*300, the increment of data size is logarithmic due to the algorithm of image compression. With the increasing amount of components, the Canvas is getting more and more complex, the data size becomes greater as well.

Size of image data from native Canvas starts decreasing slightly after the amount of components reached its threshold. Since all the randomly drawn components are all rendered in the single color black, almost all the pixels are in black if huge amount of components is drawn on the Canvas. Therefore, the complexity of the image is reduced. As a result, the compression of image data plays its role effectively and the size of image data decreases.

In general, data model from objectified Canvas is much more efficient for storing than native Canvas. As presented in the figure, if the amount of components is less than about 1650 components in size of 800*600, or less than 400 components in size of 400*300, the data model from objectified Canvas achieves much better efficiency in storing. When the amount of components reached the thresholds, then the native Canvas gains the upper hand. However, in the real world, normal users won't paint such a huge amount of components on a single Canvas.

5.3. GRAPHICAL RENDERING PERFORMANCE

While the graphical data model space means the efficiency and capability of server side, the graphical rendering performance plays a key role for the client side. Loading time of the web application is obviously an important part of user experience. To evaluate the graphical rendering performance, the rendering time in *millisecond* is measured as the amount of components increases.

- Test Environment: A representative size *800*600* is chosen and tested on *Chrome version 52*. In additional, the *Chrome Dev Tool* is used for inspecting the performance metrics. CPU, which also has an impact on the result, runs at *2.3 GHz*.

- **Metrics**: *Total Time*, which includes the scripting time, rendering time and painting time, and *Scripting Time* therefrom are measured along with increasing amounts of components drawn on the objectified Canvas.

The approach of measurement is performed as follows:

1. A graphical data model in JSON format is generated, which composes a certain amount of random components: *Rectangle*, *Circle* and *Line*.
2. Objectified Canvas starts parsing the graphical data, instantiating the components and render them on the Canvas.
3. *Chrome Dev Tool* is utilized to inspect the rendering timeline of step 2. Total time from loading to rendering and scripting time as a part of total time are recorded.

In general, four tests are performed for both types of Canvas in two different sizes.

Figure 5.3 illustrates the results of the test. As a result, two outcomes are concluded as follows.

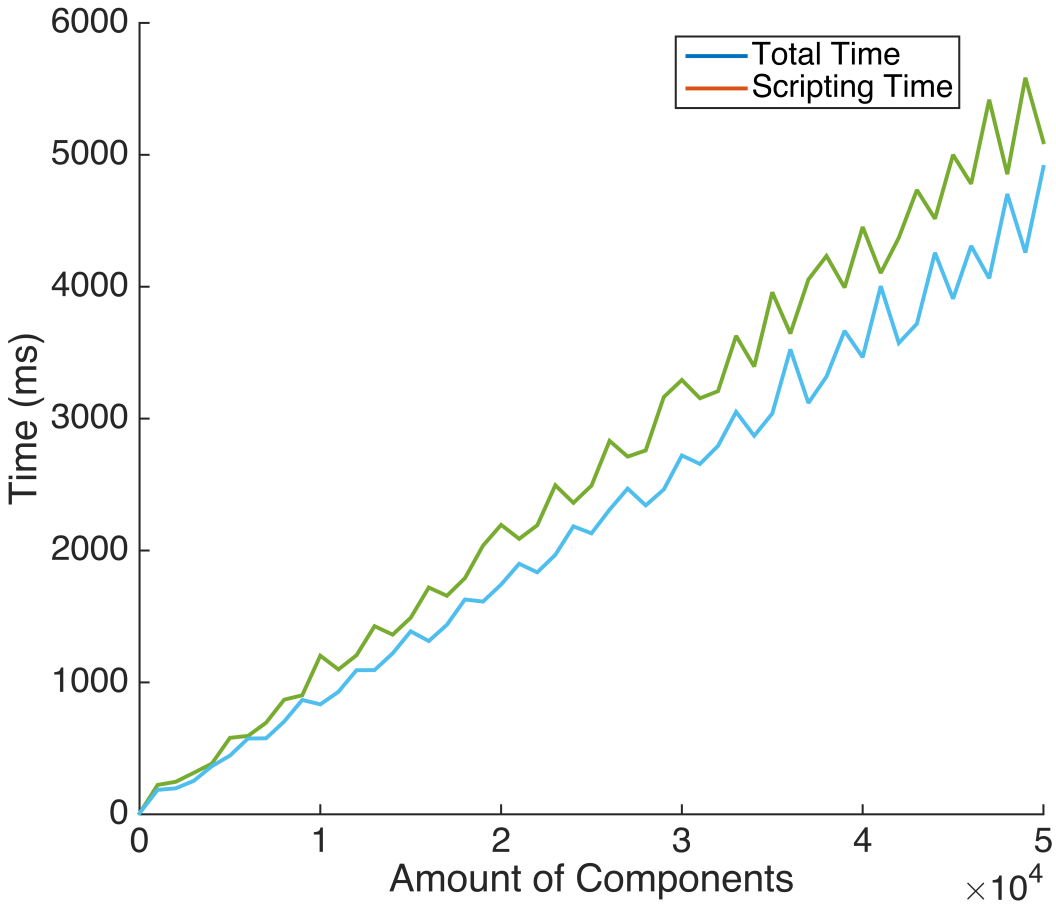

Figure 5.3.: Consumption of total time and scripting time in the rendering process of objectified Canvas

Rendering performance is approximately linear to the amount of components. As shown in the figure **??**, the total time, which includes scripting time, rendering time and

painting time is about 1000 ms when the amount of components is 1×10^4. If the amount of Components reaches 5×10^4, the total time increases to nearly 5000 ms.

Scripting time has accounted for the majority of total time. As a part of total time, the scripting time , which is also approximately linear, takes the most of time consumption while processing the rendering task. After the objectified Canvas has loaded the image data successfully, it will parse each component defined in the image data and instantiate them as well as add the objects into its own context. However, the actual rendering time, which is the difference of total time and scripting time, is only a fraction in the whole rendering process.

However, the performance test is performed with such a huge range of components' amount, which doesn't represent the usage in normal case. Considering the linearity of the rendering performance, drawing less than 1000 components, which are already huge enough for expressing the graphical content in the real world, will only spend less than 100 ms.

6. CONCLUSION AND FUTURE WORK

6.1. CONCLUSION

6.1.1. MODERN WEB APPLICATION

The discuss system is the cornerstone, which provides users the basic functionalities for discussion such like authentication, questioning and answering. The Single-Page-Application architecture is applied while concepting and implementing this discussion system. Since the client application and the server application are fully separated, the data definitions and data communication between two sides have also been discussed and implemented. RESTFul APIs within the system, which is used as a lightweight and universal web service for the data transmission, are designed and implemented on the server side.

Implementation details of building the client application using *React.js* are also described. The concept of *Componentization* is introduced, namely, all the views are actually compositions of various components. Fine-grained components are defined with templates and data representation logic inside.

In addition, in order to accelerate the development process, as well as the deployment process for further usage, an automated building workflow is also considered.

6.1.2. OBJECTIFIED CANVAS

The graphical discussion contribution made by users, which is efficient for persistence purpose and is able to be restored back to the Canvas for the quote functionality, is the main topic of this thesis. For exploring the possibility, deficiencies of native Canvas are revealed.

As a feasible approach to realize the feature mentioned above, an objectified Canvas is designed and implemented. Instead of exporting image data describing each pixel from native Canvas, the objectified Canvas outputs the graphical content to a serialized data of all objects with various properties on its Canvas.

In the evaluation phase, the storing efficiency of graphical data exported from objectified Canvas has been proved. Comparing to the image data outputted from native Canvas, the size of data model exported from objectified Canvas is much smaller, if the amount of components doesn't reach the thresholds which is basically a relative huge number.

The relationship of rendering performance of the objectified Canvas and amount of components has also been analyzed in the evaluation. As a result, the rendering performance is approximately linear to the amount of components. Moreover, as a part of the total time in the rendering process, the scripting time occupies the majority of time consumption.

6.1.3. REAL-TIME COMMUNICATION

Real-time communication, which enables the bi-directional communication between client and server, is also a focal point of this work. Arbitrary resources are able to be subscribed and users will get notified and acquire the newest data passively as the content of subscribed resource changes.

To achieve this goal, *WebSocket* is applied as the basic real-time communication protocol. To broadcast data precisely through WebSocket to the users who subscribe the certain resources, a list which maps user socket to resource id is maintained by server, after a WebSocket connection has been successfully established. As the state of resource is changed, users who have subscribed this resource are notified with the new state of resource by querying the mapping relation in the list.

6.2. FUTURE WORK

Although the developed prototype of graphical discussion system covers the requirements and realizes the basic functionalities, some future researches and improvements are still needed to be done.

Notification system could be extended for the discussion system. For now, users won't get notified if new answers are posted under their own questions. Therefore, a notification system is proposed. Users would also be able to subscribe a certain question or class he interested in for further notifications if new contributions are made under it.

More pre-defined shapes of components should be extended for the drawing tool. According to the result of evaluation in section 5.1, most users hold the idea that the preset of shapes in the drawing tool are far not enough. Therefore, the drawing tool should have provided more pre-defined components natively, which will significantly ease the drawing process and helps the user to express the precise; graphical content as expected.

Divers stylings of text on the drawing tool should be implemented. The developed drawing tool already provides the possibility to input textual content for now. However the current styling of the text is still circumscribed. At present, adjusting the size or color of the text is already possible. More stylings such as strikethrough, list format could be extended in the future.

LIST OF FIGURES

2.1. Web architecture in early age 14
2.2. Web 2.0 architecture 15
2.3. SPA architecture 16
2.4. Components in SPA 16
2.5. Performance comparison of Canvas and SVG[8] 20

3.1. Submit a new course 24
3.2. Search course with code 24
3.3. Favor course 25
3.4. Submit a new question; withdraw or modify own question 25
3.5. Upvote/Downvote a question or answer 26
3.6. Submit/Quote an answer 27
3.7. Drawing editor with drawing history 28
3.8. Notify with new question automatically 28
3.9. Auto re-order answer if vote contributions changed 29
3.10. General architecture in conception 30
3.11. General data communication 31
3.12. Relations between data domains 32
3.13. Canvas to native ImageData 37
3.14. Concept of objectified Canvas 38
3.15. Export and import of Canvas context with objectified elements 39
3.16. Lifecycle of drawing tool 40
3.17. Sequence diagram of establishing a WebSocket connection 41

4.1. Overview of Graphicuss' file structure 44
4.2. General architecture 45
4.3. Proxy for client development server 46
4.4. Automated building process with Webpack 46
4.5. Table of data model 47
4.6. Overview of server app's file structure 48
4.7. Server architecture 49
4.8. JWT authentication process 51
4.9. Overview of client app's file structure 53

4.10. Overview of client architecture 54
4.11. Composition of components in courses' page 56
4.12. Data flow in Flux architecture 57
4.13. Architecture of drawing tool 59

5.1. Proportion of usability problems in an interface found by heuristic evaluation using various numbers of evaluators 65
5.2. Comparison of data spaces of image data from native Canvas in two sizes(800*600, 400*300) and adopted graphical data model in Graphicuss 68
5.3. Consumption of total time and scripting time in the rendering process of objectified Canvas 70

LIST OF TABLES

3.1. Fields for Each Data Domain 33
3.2. HTTP methods on User resource 33
3.3. User Auth APIs 34
3.4. Course Resource APIs 34
3.5. Question Resource APIs 34
3.6. Answer Resource APIs 35
3.7. Answer Resource APIs 35

5.1. Score of SUS table 64
5.2. Score of each question in the interview 66

LIST OF LISTINGS

2.1. Simple Example of SVG elemnt 19

4.1. Example: user schema definition within Mongoose 49
4.2. Example: user schema definition within Mongoose 50
4.3. JWT encodes user information with secret key 51
4.4. JWT decodes user information with secret key 51
4.5. Server starts listening for requests over WebSocket protocol 52
4.6. Server starts listening for requests over WebSocket protocol 52
4.7. Router in client app 54
4.8. Rendering *CourseView* with multiple *CourseCard* components 55
4.9. Serialized Canvas by Fabric.js 58
4.10. Main process of StyleManager 60
4.11. Main process of FeatureManager 60

5.1. Example of image data while calling toDataURL() 67

A. SYSTEM USABILITY SCALE TABLE

	Strongly Disagree				Strongly Agree
1. I think that I would like to use this system frequently	1	2	3	4	5
2. I found the system unnecessarily complex	1	2	3	4	5
3. I thought the system was easy to use	1	2	3	4	5
4. I think that I would need the support of a technical person to be able to use this system	1	2	3	4	5
5. I found the various functions in this system were well integrated	1	2	3	4	5
6. I thought there was too much inconsistency in this system	1	2	3	4	5
7. I would imagine that most people would learn to use this system very quickly	1	2	3	4	5
8. I found the system very cumbersome to use	1	2	3	4	5
9. I felt very confident using the system	1	2	3	4	5
10. I needed to learn a lot of things before I could get going with this system	1	2	3	4	5

B. SYSTEM USABILITY INTERVIEW

	Strongly Disagree				Strongly Agree
1. I could find the certain course created by tutor.	1	2	3	4	5
2. The process of asking a question or answering a question was simple.	1	2	3	4	5
3. Voting score would help to locate the useful contribution.	1	2	3	4	5
4. Integrated text input in the drawing tool would helps express the textual content.	1	2	3	4	5
5. The drawing tool provided enough elements ready to be drawn.	1	2	3	4	5
6. Quoting and modifying on top of otherścontributions are useful.	1	2	3	4	5
7. I found the history function(undo redo) of drawing tool was really necessary.	1	2	3	4	5
8. Auto-ordering of answers was seamless and unconspicuous.	1	2	3	4	5
9. I found the real-time feature was interactive.	1	2	3	4	5
10. The real-time feature didn't cause distraction while viewing the answers.	1	2	3	4	5

BIBLIOGRAPHY

[1] L. Richardson and S. Ruby, *RESTful web services*. " O'Reilly Media, Inc.", 2008.

[2] H. Hamad, M. Saad, and R. Abed, "Performance evaluation of restful web services for mobile devices.," *Int. Arab J. e-Technol.*, vol. 1, no. 3, pp. 72–78, 2010.

[3] J. L. Williams, *Learning html5 game programming: A hands-on guide to building online games using Canvas, SVG, and WebGL*. Addison-Wesley Professional, 2012.

[4] D. Geary, *Core HTML5 canvas: graphics, animation, and game development*. Pearson Education, 2012.

[5] P. Corcoran, P. Mooney, A. C. Winstanley, and M. Bertolotto, "Effective vector data transmission and visualization using html5," 2011.

[6] MDN, "Svg tutorial - introduction," 2016.

[7] J. Ferraiolo, F. Jun, and D. Jackson, *Scalable vector graphics (SVG) 1.0 specification*. iuniverse, 2000.

[8] B. Smus, "Performance of canvas versus svg," 2009.

[9] MDN, "Websockets," 2016.

[10] V. Pimentel and B. G. Nickerson, "Communicating and displaying real-time data with websocket," *IEEE Internet Computing*, vol. 16, no. 4, pp. 45–53, 2012.

[11] A. B. Johnston and D. C. Burnett, *WebRTC: APIs and RTCWEB protocols of the HTML5 real-time web*. Digital Codex LLC, 2012.

[12] MDN, "Http access control (cors)," 2016.

[13] J. Brooke *et al.*, "Sus-a quick and dirty usability scale," *Usability evaluation in industry*, vol. 189, no. 194, pp. 4–7, 1996.

[14] A. Bangor, P. Kortum, and J. Miller, "Determining what individual sus scores mean: Adding an adjective rating scale," *Journal of usability studies*, vol. 4, no. 3, pp. 114–123, 2009.

[15] J. Nielsen, "How to conduct a heuristic evaluation," *retrieved November*, vol. 10, 2001.

[16] A. Barron, J. Rissanen, and B. Yu, "The minimum description length principle in coding and modeling," *IEEE Transactions on Information Theory*, vol. 44, no. 6, pp. 2743–2760, 1998.