**Virtual Lab Documentation**

*yellow highlighting within this document represents a feature/features that we recommend implementing or need to be implemented.*

**Introduction**

As our group initially determined, the basic idea for a virtual lab is a collection of different activity views that can be navigated through and completed by a student. These activities should ideally engage a student in their learning more than a traditional, more monotonous exercise (such as reading from a textbook, watching lectures, etc).

To transform the idea of a virtual lab into a basic working application, our group decided that the virtual lab would take on the structure of a **logical tree**, with each activity view represented as a **node** descendant of the root (the root being the initial node). As you navigate this documentation, it will be important to remember these terms as they are used quite frequently to describe various functionalities/behaviors within the virtual lab structure.

We have also decided as a group that two types of users shall participate in the virtual lab application – administrators (**instructors**) and standard users (**students**). Ideally, an instructor should be able to build their own lab(s) for students to participate in using an administrator-only lab-builder view, and view analytical information collected from labs as they are completed by students. Students should only be able to view and participate in a lab that they wish to complete; they cannot build labs, view analytical data, etc.

**Set up & Installation**

1. Clone/download the antibuddies project to your machine from the github repository: <https://github.com/bradleypeterson/antibuddies>
2. Using the command-line, navigate into the virtual lab project’s base directory (antibuddies/virtual lab)
3. Ensure that you have node and node package manager (npm) installed on your machine. If not, follow [this guide](https://docs.npmjs.com/downloading-and-installing-node-js-and-npm) on installation.
4. Run the following command to install project dependencies (without the quotes): “**npm install**”. (*Side note:* Any time you install another dependency in your project and push changes utilizing said dependency to the repository, be sure to notify your team members to run this command again after they pull the new code - this will ensure the new dependency is also installed for them – and that their local version of the application won’t crash.

Keep in mind that when pushing to the github repository, you should avoid pushing the node\_modules directory that is created/populated with all of the project dependencies. Rather, anyone who wants to build the project locally should install the dependencies on their machine using the ***npm install*** command).

1. If not already open, bring up your IDE of choice and open the project to start making any changes.
2. When you are ready to build and view the project live, run the command “**npm start**” (without the quotes). This will bring up a new tab in your browser running the development server at localhost:4200. It supports hot-reloading so that any changes you make to the code will render in the browser as you make them.
3. To shut the development server down, hit **ctrl-c** with your command line window active.

Before diving into our code, we would highly recommend that you view the [Angular’s documentation](https://angular.io/docs) as to at least become familiar with the basics of this front-end framework. We would also recommend taking a peek at the [D3 documentation](https://github.com/d3/d3/wiki) as this is a library that was utilized in the analytical/tree structure display part of the project.

**Lab Data Structure & API**

**Overview:**

The data structure and the backend-service is intended for the components to all work with the same data structure to store and retrieve data. This is persistent throughout the application, but will expire when the application ends. To make the data persistent the plan is to use a restful API to store the data structure.

The API was completed the last week of development so we didn’t finish all of the put, get functionality from the backend-service, but as far as we got is detailed below.

**Data Structure:**

The data structure is structured in typescript classes: LabsContainer, lab, node, Quiznode, answerClass. The declaration of these classes is in app/interfaces.ts.

* class LabsContainer:
  + Attributes:
    - labs:lab[] = []
      * Array to hold all of the labs, populated by the createLab method
  + Methods:
    - constructor - empty
    - createLab(labName:string):lab
      * Creates a lab and adds it to the LabsContainer Labs array. It auto increments the labID so the labID is also the same as the array index in the Labs array.
      * Additionally this checks to see if a lab of the same name exists so ensure there are no duplicates. This will throw a javascript error if it already exists which can be seen from developer tools.
      * Also it populates the created date
    - findLabByName(name:string):number
      * As the name says, it finds a lab by name and will return a -1 if not found or an index in Labs array if it is found.
* class lab
  + Attributes:
    - labID:number
      * this is set by the createLab method in LabsContainer and is also the index of the lab in the Labs array in LabsContainer
    - createdDate:date
      * the date created, set by the createLab method in LabsContainer
    - Course:string
      * A string to represent the course the lab belongs to
    - Nodes:any[]
      * An array of nodes, this is of type any because it is intended that as more different node types are created, they will populate this array
    - Description:string
      * A description of the lab
    - Name:string
      * Name of the lab. Name must be unique when created in LabsContainer createLab method
  + Methods:
    - constructor(labName:string,labID:number)
      * Used by LabsContainer createLab method, sets the name and the id after ensuring the name is unique and the labID has been generated as its index number in the labs array in LabsContainer
    - createQuizNode(nodeName:string):Quiznode
      * Creates a quiz node and adds it to the nodes array in the lab.
      * Name must be unique, an error will be thrown if not
      * ID is auto incremented on creation
    - findNodeByName(name:string):number
      * This will find a node by its name and return the index if found or a -1 if not found, this is used in the createQuizNode method to ensure uniqueness of lab names.
* class node //parent class for other node types
  + Attributes:
    - Name:string
      * Name of the node, set by the createQuizNode method from lab, name should be unique
    - nodeID:number
      * ID which is also the index number in the nodes array in the lab class. This is set by the createQuizNode method in the lab class.
    - outGoingNodes:number[]
      * This is used to keep an array of all connecting outGoing nodes for creating the D3 tree diagram
  + constructor(name:string,nodeID:number)
    - Constructor, takes name and ID. It is intended for this to be called from lab.createQuizNode as it ensures the name is unique and the ID is also the index number in the nodes array in the lab class.
* class Quiznode extends node
  + Attributes:.
    - Description:string
      * Description of the quizNode
    - Question:string
      * The question part of the quiz in the node
    - answers:answerClass[]
      * An array of answers for the question, each one holds an outgoing node, if that answer is chosen then the respective outgoing node is where the user would be navigated to.
  + Methods:
    - constructor(nodeName:string,nodeID:number)
      * Extends its parent for setting name and ID. As before, name is to be unique and ID is the array index. These are set by the createNode method in the lab class
    - createAnswer(answerText:string,connectingNodeID:number):answerClass
      * This creates a new answer for the quiz question.
      * The text of the answer is to be unique and this is checked. If it is not unique an error is thrown and can be viewed in developer tools.
      * The id is the array index and is auto-set by this method
    - findAnswerByText(name:string):number
      * This finds the answer by text and returns the array index if it is found or a -1 if it is not.
      * This is used by the createAnswer method to ensure answer texts are unique
  + class answerClass
    - Attributes:
      * answerID:number
        + The ID is also the index, it is set when created by the createAnswer method in the quizNode class
      * answerText:string
        + This is the answer string, it is to be unique and is set by the createAnswer method in the quizNode class
      * connectingNodeID:number
        + The node to navigate to if this answer is selected
    - Methods
      * constructor(answerText:string,connectingNodeID:number,answerID:number)
        + The constructor is utilized by the createAnswer method in the quizNode class.

**Backend service:**

* The backend service is located in the typescript file: app\back-end-service.service.ts
* The backend service utilizes the classes of the data structure by importing app/interfaces.ts
* A labsContainer is created and is persistent throughout the app with the code:labsContainer = new LabsContainer()
* makeExample
  + This creates mock data and populates the data structure for testing purposes.
* Restful API integration:
  + This wasn’t completed, the api was finished in the final week and the correlated functions weren’t completed.
  + The API functions use the HttpClient and HttpHeaders imported at the top
    - API location:<https://antibuddies-api.glitch.me/api>
    - putLab(lab: lab): Observable<lab>
      * Puts a new lap to the restful API
    - putNode(node: any): Observable<any>
      * Puts a node to the specified lab by id
    - getHTTPLabsContainer(): Observable<LabsContainer>
      * Intended to populate the data structure after a get request
    - getLabNo404<lab>(id: number): Observable<lab>
      * Intended to get a lab by id and not return a 404 if it is not found and instead return a message, did not find

**Utilizing the backend service from a component:**

1. Ensure you import the backend service example:

import { BackEndServiceService } from '../../back-end-service.service';

1. Add to the constructor of your component, here the example is private so it can only be used in the class. If you want to use these variables in your html component make it public example:

private data: BackEndServiceService

1. Access the data, example:

this.data.labsContainer.createLab(this.labName)

**Example Population of the Data Structure:**

This can be found in the makeExample method in the backend service:

app\back-end-service.service.ts

**Administrator (Instructor) View**

Based on your credentials when in the labview page when you click on a lab if you are a student you will be taken to the student view page to start taking the lab. If you are an administrator you will be taken to the admin page. From the labview page if you are an administrator you will have the ability to create a lab. The labview component is located at /labview folder.

The admin page is where the administrator can edit a lab or create nodes. The admin page also contains the analytics component, the analytics is just there, it isn’t connected to anything, you will have to create a way to collect analytics from the lab. The components files are In the /Admin folder.

When you create a node it will take you to when you can type in the description then you have to click save. After that you will select which type of node you want to create as of now you create. For now the quiz node is the one that you can create. When you click on the quiz node you can enter a question then you can click on add new answer then you can enter an answer next to the answer you can click which node it will take you to. After that you click save Node then click on go back. You will then see a node on the node list. What you will need to do is connect the node to the tree structure.

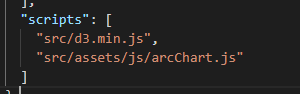
**Visualizing The Data Structure**

As a part of the administrative section of our site, we created a visualization for the data structure. A simple branching tree wasn’t enough to display the information required so we decided to go with an Arc Chart diagram. We compared and contrasted similar ways to visualize our lab data, but determined that the arc diagram was probably best to convey our message given the data we needed to show. To help with the visualization, we used a javascript library called d3.js (found at: <https://d3js.org/>) (with the arc diagram exampled here: <https://www.d3-graph-gallery.com/arc.html>). While the library is perfectly capable of achieving the look we desired, angular wasn’t happy with us using external libraries written in vanilla javascript vs using angular’s native typescript.

Once Angular is fully installed, from the angular CLI, you can install d3.js with the command: npm install d3.

After struggling to convert example code to work with angular and typescript which we had attempted previously, we decided to try to get the example code to work with angular by importing the JS file directly into our angular component. It is important to show how we got it to work so that future additions will be much smoother or so that modifications can be applied.

To import an external file written in vanilla javascript, we must modify the angular configuration file /virtual%20lab/angular.json:



Under “scripts”, we add the link to the script file which is stored in /virtual%20lab/src/assets/js/arcChart.js. Then within the angular component (**admintreepan.component.ts**) we can now start to use the script by adding the script tag using angular’s typescript for use later: 

The arc chart currently works with angular despite the inelegant addition of the vanilla javascript, but contains some visual bugs. Some modification to the script might be in order, especially if the data structure changes to work with future lab nodes.

**Student’s View Of Lab**

At the time of our simple implementation of the project, a virtual lab is currently made up of quiz-nodes, each of which is simply a question and a set of answers that the student can currently only select one single answer from. Basically a quiz node is equivalent to a multiple-choice exam question with radio button choices for answers.

**User Interface Interaction – Student Completing a Virtual Lab**

To begin a lab, a student will navigate to it by selecting the lab from the list of available labs that is populated when they first start the Virtual Labs application. (This is vital to the application and needs to be implemented instead of the dummy data currently being used, described below).

Currently, our “student” can complete a lab by clicking the **StudentLab (Demo)** tab at the top of the application view (this lab originally used dummy data and is strictly for testing purposes). This will bring them to a screen displaying the title and description for that particular lab, followed by a button beneath the description labelled “**Begin lab**”. To proceed, the student will activate this button to start the lab. To cancel, the student can return to the list of available labs by clicking on the **labview** tab at the top of the application view.

Once a student has started a lab, the descriptive information & “begin lab” button disappear, and the screen will display the first node’s information in its place, which includes the following:

* The question text
* A current node identifier (this was for testing only - should be removed)
* A series of answers to the question, each displayed as a selectable bar with a radio button to indicate whether it is the currently selected answer.

Beneath these elements lies a horizontal toolbar of actions related to the lab that a student can perform whilst visiting a given node. For the purposes of a quiz-node-only lab, we implemented three main actions for the toolbar: **previous node**, **next node**, and **exit lab**. Each is a button on the toolbar that performs what is implicated by the name:

* **Previous node** allows a student to traverse backwards to the last node they visited, and becomes disabled if the student decides to perform this action. In other words, the student can revisit the previous node, but they cannot traverse backwards any further than that.
* **Next node** allows a student to traverse to the next node, and becomes a function available to the student only after an answer has been selected for the presented question. Note – for fallback/security purposes, this action remains disabled for any selected answer that does not lead to another node (this should not be an issue).
* **Exit lab** allows a student to exit the lab (back to the list of labs) at any time while going through it.

Upon reaching the final node of the lab, the very last answer they select will replace the next node button with a button labelled “**next**”. This action simply takes the student to a view congratulating them on completing the lab, along with a button labeled “**return to labs**” that will take them back to the list of available labs.

**Logical/Code Structure of Student Virtual Lab**

Within our code, there are **three components** that work together to present the student with a working view of the lab they are completing, each a directory containing component files. All can be found within the **student/student-lab** subdirectory of the app directory (virtual lab->src->app->student/student-lab):

* **student-lab directory**:
  + student-lab.component.ts (the **typescript class file** for a student lab)
  + student-lab.component.html (the **HTML template** file for a student lab)
  + student-lab.component.css (**CSS** specific to the student-lab component)
  + **quiz-view subdirectory**:
    - quiz-view.component.ts
    - quiz-view.component.html
    - quiz-view.component.css
  + **student-toolbar subdirectory**:
    - Student-toolbar.component.ts
    - Student-toolbar.component.html
    - Student-toolbar.component.css

Directly within the student-lab directory, the student-lab.component.ts class file (containing the component logic) is linked to the student-lab.component.html template file (describing the viewable UI markup) and the student-lab.component.css file (describing the UI styles). **All other components of the application follow this same format within their respective directories**, as per the default development structure of an Angular-based web application.

To get up and running with the student lab logic, here are some major items to become familiar with/aware of in the **student-lab.component.ts file**:

**Properties:**

* **labName** - the name of the lab to be loaded into the component.
* **labDescription** - the description of the given lab.
* **nodes** - the array of nodes to be loaded in from the given lab.
* **isBegin** - A flag to check if a student has pressed the “begin lab” button (shows the content of the initial lab node when true).
* **isFinal** - A flag to check if a student has selected an answer on the final node (shows/activates “next” button on the student toolbar when true).
* **isFinished** - A flag to check if the student is done with the lab (when true, shows the congratulatory card to the student).
* **currentNode** - an ID number of the node that is currently being viewed by the student. This is often checked within node traversal methods.
* **nextNode** - an ID number of the node that will be traversed to given the currently selected answer to the node’s question. Often used in traversal methods.
* **prevNode** - an ID number of the previous node to that being currently visited. Used for backwards node traversal.
* **disabledNext / disabledPrevious** - Sets the “disabled” attribute of the HTML button to either “disabled” or “” (empty string) depending on whether or not the toolbar action should be disabled for forward/backward node traversal.
* **finalNode** - the ID number of the last node of the lab.

**Methods:**

* **handleBegin()** - trips the **isBegin** flag variable to true, showing initial lab node.
* **getNextIndex(next)** - returns the array index in **nodes** array of the node with the supplied node ID argument
* **goNextNode(next)** - sets **nextNode** variable to the supplied node ID argument, which is passed as an event from the quiz-view component as the student selects one of the available answers. The ID of the node that connects to the selected answer is what is passed and set for forward tree traversal.
* **handleTraverseForward()** - handles forward traversal by updating **prevNode** and **nextNode** variables to new node IDs - prevNode becomes currentNode and currentNode becomes nextNode. The **next node** action of the student toolbar becomes disabled, whereas the **previous node** action becomes enabled.
* **handleTraverseBackward()** - handles backward traversal by setting **nextNode** to null and **currentNode** to prevNode. Both toolbar actions (**next node** & **previous node**) become disabled, and if **isFinal** was true, it will now be false.
* **handleLabEnd()** - trips **isFinished** to true, triggering view of lab end card.

**Data Passing:**

Data is frequently passed around from parent to child and vice versa in the student-lab component, which also requires logic in both the **student-lab.component.ts** file and the **student-lab.component.html** file. Here is a brief overview of how this occurs logically:

* **quiz-view**:
  + **Receives** the ID of the current node, the current node’s question data & an array of answer data from student-lab.
  + **Emits** the ID of the next node to student-lab, which is determined by the selected answer to the question. This occurs in the **determineNextNode(nextNodeAsSelected)** method of quiz-view.component.ts.
* **student-toolbar**:
  + **Receives** the value of the **isFinal** flag variable to determine visibility of the final “next” button in the toolbar, and receives disabled attributes of the **next node / previous node** buttons to determine whether these actions are available to the student.
  + **Emits** events when the actions are available and clicked - the methods **handlePreviousNode()**, **handleNextNode()** (and **handleFinish()**) are called after a student clicks one of the respective toolbar buttons. Each event handler method sends a boolean EventEmitter object to the student-lab parent, and when the value of it is true, the corresponding parent method will be called. (For example, one EventEmitter is named “traverseForwardIsOkay”, and once this is true (i.e., after the click occurs) it will trigger the **handleTraverseForward()** method of the student-lab parent component. (See <https://angular.io/api/core/EventEmitter> for more details on event handling - there are a lot of moving parts here).
* **student-lab**:
  + **Receives** the ID of the next node (determined in quiz-view based on the answer the student selects).
  + **Passes** the values of **disabledNext** & **disabledPrevious** variables to the student toolbar - determines whether the **next node** & **previous node** buttons will become enabled and clickable.

**Application CSS Styles**

Styles for this project were adapted from those of another subgroup to keep all visuals consistent for the overarching Antibuddies project. Styling for our application was done on an as-needed basis, therefore it is perhaps not as structured as some would prefer. However, styles were added in a way that should be straightforward and easy to understand, and kept to a minimum for cleanliness and conciseness. Most items are in the global **styles.css** file in the root source-code directory of the project (**virtual lab->src**), although some component-specific style rules can be found in individual component css files (e.g., **student-lab.component.css**). More details about specific styling rules can be found in the global styles.css file.

**Unfinished Items for Future Implementation**

**Linkage of lab data service to an external data API**

Currently unfinished, the components are not all linking with the back-end service’s data structure for persistence throughout the app. The intent is to have the admin page create the nodes, add them to the data structure and restful API(Identified in the data structure and API section). Then on the student view pulling from the data structure and the restful API. The student lab demo shows this working from example data that was put into the data structure from the makeExample method in the app\backend-service.services.ts. This just needs to be mirrored on the admin creation page and sync with the restful API.

**Analytics**

Add analytics for students that have started the lab and their completion percentage. Within the admin view, we have left a space for lab data to be visualized, but we have not added the tracking API or other requirements to make this work in scope.

**Multi-choice quiz nodes**

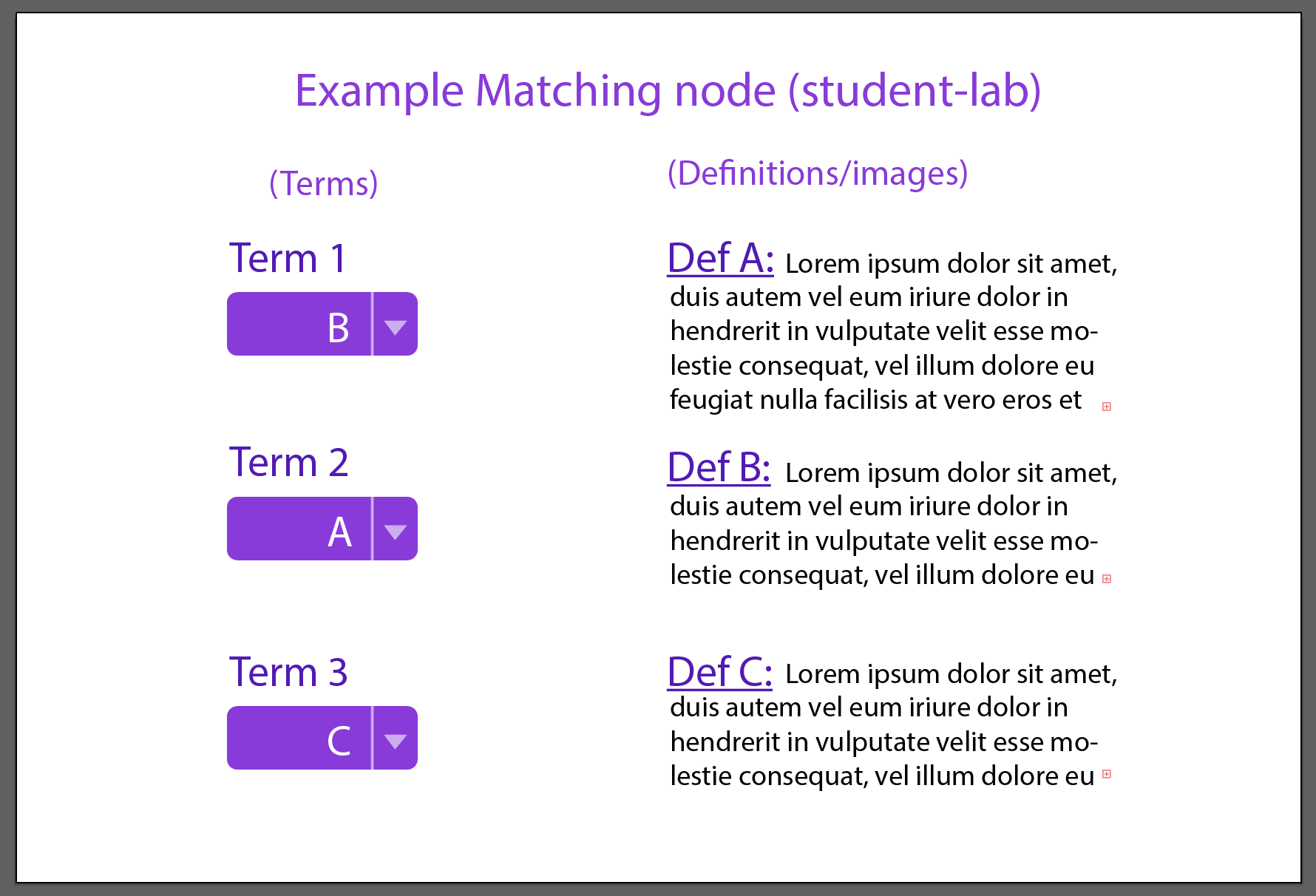
check-box answers, so the student could select multiple answers in a particular quiz node. Certain combinations of answers will need to be taken into consideration in determining whether they are compatible in traversing to another node.

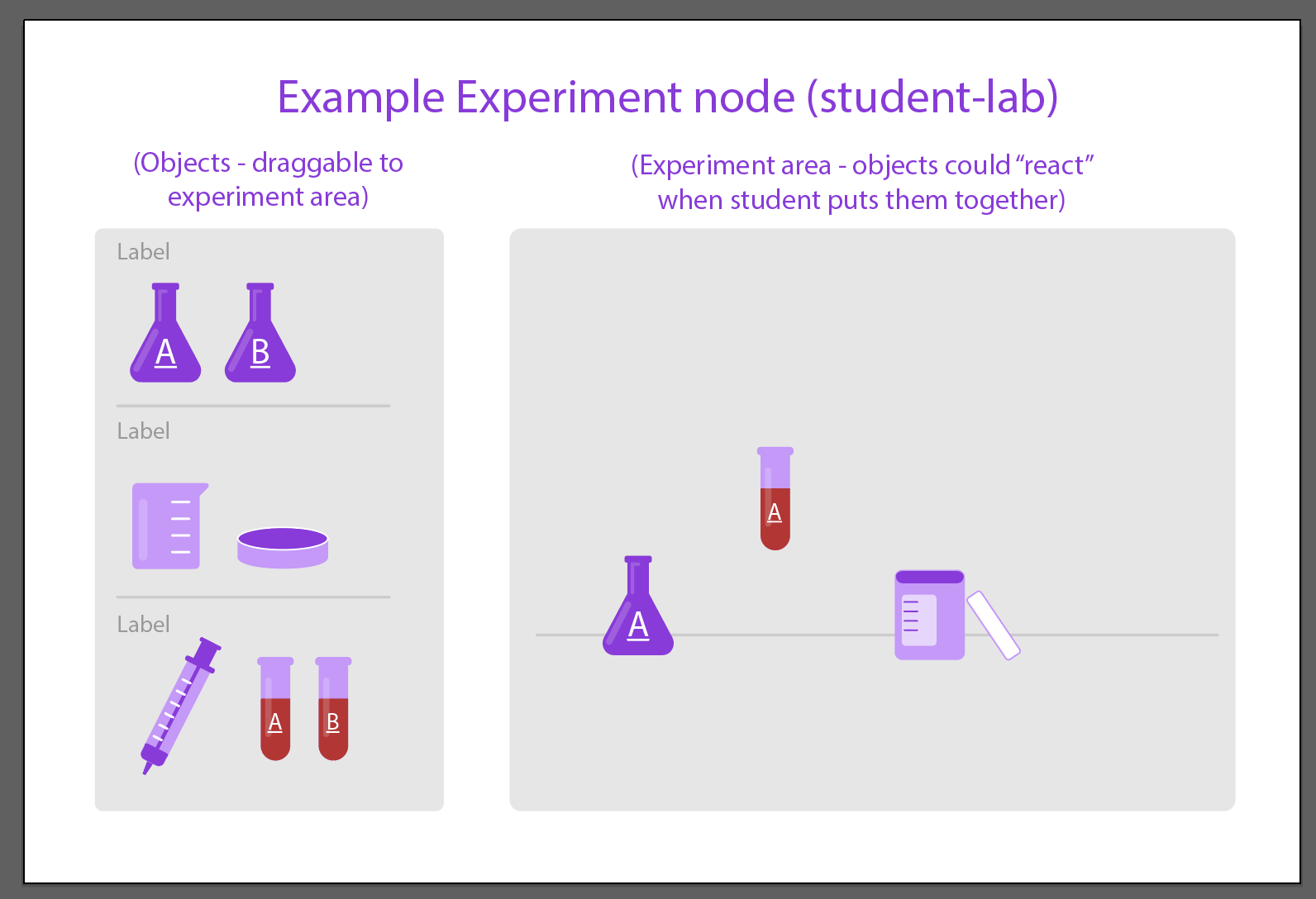
**Image/video uploading capability for nodes**

The clients were very interested in this feature. They wanted to be able to give a brief lecture between nodes to explain to students what they would be doing in the following nodes.

**Different types of nodes**

We chose to make our nodes only of the quiz question variety to simplify the process of implementation, as the other node types we considered (matching nodes, experiment nodes, and video nodes) were deemed to be too work-intensive in the amount of time we had to complete this set-up portion of the virtual lab project. Here are some examples of what these could look like:

****

****

*yellow highlighting within this document represents a feature/features that we recommend implementing or need to be implemented.*