Labs 3 & 4: Welcome to *Qzicl*!

version 1.1 - 2025/10/20-11:16

Qzicl is a web application that enables users to select a topic from a list, then take a multiple-choice quiz to check their knowledge of that topic. It can be used for entertainment or as a flashcard study system for learning. Labs 3 and 4 will be focused on this topic:

- Lab 3 provides the browser-based **client** that handles quiz presentation and scoring using a *Mock Service Worker* (MSW) model instead of a server.

  Lab 3 is due Nov 2, 2025 at 22:00 ET.
- Lab 4 provides the **server** that manages all of the quiz data, interacts with the apps running in the client's browser, and keeps statistics by user. As the **server** is built, the **client** is updated to replace the MSW tools with accessing the *real* server instead.

  Lab 4 is due Nov 13, 2025 at 22:00 ET.

Lab 3 builds the client for the quiz management system as a complete, standalone app running in a browser. As is often the case in industry, the development of Lab 3 will be done without a server (perhaps someone else is building it or it isn't ready yet). We will learn how to use the *Mock Service Worker* substitute which allows the development of Lab 3 to proceed before Lab 4 is completed.

Lab 3 flow

User accesses the client app and is presented with a list of available topics. General information about each topic is provided and the user selects one. The user is then presented with a list of quizzes available for that topic. For example, if the topic is "math" the available quizzes might be "geometry", "calculus", "general math". The use selects a quiz and is then presented with a "get ready screen". When the user clicks "go" on that screen, the app begins posing the quiz questions one by one. All the questions are multiple choice, and the question and answers may be text or images. The use selects an answer, clicks "next", and is presented with the next question. After answering the last question and clicking "done", the results are displayed for the user and the quiz ends.

If the client decides to exit the quiz early, there needs to be a way to reset the system state so that the quiz is no longer thought to be running. A special route `api/exit` is used for this purpose. If the client app crashes, the simplest recovery is to restart the server.

Of course, the ***Pro*** way to handle this would be to set timeouts for each question such that if no answer is received before the timeout expires, the server automatically closes the quiz.

Architecture

Components

Component name	***Component Use***
`Topic`	Contains title text and a collection of `Quiz` components.
`Quiz`	Contains quiz data (title, introductory text, images, etc.) & collection of `Question` components. It may also tracks the number of times this `Quiz` was chosen.
`Session`	Contains data regarding a specific session with a `Quiz` : date/time started and ended, final score.
`Question`	Contains the question text and a collection of `Answer` components. It also has an indication of which `Answer` component is correct.
`Answer`	Contains an individual answer text or image.
`Settings`	Contains system-wide settings. Loaded at client startup, it has these values (and more if you need them): Number of questions in a quiz.

Routes

The server (or MSW) responds to the following routes and requests. Response status codes are `200` unless otherwise specified.

Route	*Request* (JSON)	*Response* (JSON)
`POST /hello`	`{"who": "qzicl", "action": "hello"}`	`{ "title": "text", ["topicid1":"text", "topicid2":"text", ... ]}`
`GET api/topics`	`{"topicID": number}`	`{[{"quizID1": number, "title1":"text"}, {"quizID2": number, "title2":"text"}, ... ]}`
		`404` : `{"error":"badtopicID", "topicID": "text"}`
`GET api/quizes`	`{"quizID":number}`	`{"sessionID": number, "quizIntro": "text"}`
		`404` : `{"error":"badquizID", "quizID": "text"}`
`GET api/go`	`{"sessionID": number}`	`{"questionID": number, "questionText": "text", [{"answerID1": number, "answer1": "text"},{"answerID2": number, "answer2": "text"}, ... ]}`
		`404` : `{"error":"badsessionID", "sessionID": number}`
`GET api/continue`	`{"sessionID": number,` / `"questionID":number, "answerID": number}`	`{"questionIDnext": number, "questionText": "text", ["answerID1": "text", "answerID2": "text", ... ]}`
		`404` : `{"error":"badquestionID", "questionID": number}`
`GET api/go`	`{"sessionID, number, `"answerID": number}`	If no more questions: `{"results": "text"}`
`POST api/exit`	`"sessionID"`	`{"exited": true}`
`ANY api/*`	`<ignored>`	Default route when others don't match. `400` : `{"error":"bad route"}`

Sample quiz file

```
{
  "topic": "General EU Knowledge",
  "title": "EU Capitals Quiz",
  "quizMessage": "Test your general knowledge with this sample quiz!",
  [
    {
      "question": "What is the capital of France?",
      "options": [
        "Berlin",
        "Madrid",
        "Paris",
        "Rome"
      ],
      "answer": "Paris"
    },
    {
      "question": "What EU capital is west-northwest of Brussels?",
      "options": [
        "Amsterdam",
        "Madrid",
        "Paris",
        "London"
      ],
      "answer": "London"
    },
    {
      "question": "The largest EU capital by area is?",
      "options": [
        "Rome",
        "Berlin",
        "Amsterdam",
        "Madrid"
      ],
      "answer": "Berlin"
    }
  ]
}
```

Minimum requirements for Lab 3

> **Note**
>
> This has been simplified from what was presented in class October 20-21.

Complete and operational client as specified earlier in this document.

- Full interface with all components implemented using a **MSW** so that switching to a real server in Lab 4 will be much easier.
- Quiz data is in JSON form and read in from a file using the Node.js `fs` module for delivery by the MSW.
- All request-response messages are implemented and sent to the appropriate route (e.g., "`api/<name>`").
- All routes defined and managed by MSW.
- Includes sample quizzes in JSON format in files processed by JavaScript's `fs` package.
  - Two or more topic areas
  - Two or more quizzes in each Topic area
  - Three or more multiple-choice questions in each quiz
  - Three or more possible answers for each question
  - Minimally, these add up to

    $$(2 \text{ topics}) \times (2 \text{ quizzes}) \times (3 \text{ questions}) \times (3 \text{ answers}) = 48 \text{ questions} \quad (1$$

    These can be simple or interesting ... I know which I would choose 😎.

Extra credit

- Use Vitest for testing (**2 to 5 points**, depending on thoroughness). Submit the associated additional files with the rest of the assignment. Note that `Vite` doesn't support the more traditional `Jest` testing framework.

Lab4

Produce the Qzicl Server and change the Client from Lab 3 to use live `http` request-response messages with the Server for all quiz interactions.

Qzicl client-server interaction

All interaction with the server is through the standard client-server request-response pattern over `HTTP`.

- The React application (the client) initiates requests to the server's API endpoints using the HTTP `GET` or `POST` methods. All other methods are errors.
- The server exposes specific URLs (endpoints) that the client can use for requests to perform the different actions, such as fetching a list of available quiz topics.
- The server responds to client requests with a status code and a JSON body. For a successful response, the React client app parses the JSON body into a JavaScript object using `response.json()`. See the **Routes** definitions above.
- The React application uses hooks like `useState` to manage state (e.g., the data received from the server) and `useEffect` to handle side-effects such as data fetching or manually changing the DOM.
- The frontend (Lab 3) and the backend (Lab 4) use Axios instead of the native `fetch`. Axios is a popular, promise-based `HTTP` client that offers more features than the native Fetch API. It includes automatic JSON parsing and better error handling. Here is an example on CodePen.io.

Base requirements for Lab 4

- Complete front end using React and working with the back end using a REST API.
- Using `axios` for `HTTP` requests and JSON unpacking.
- Provides sample quizzes:
  - Two or more topic areas
  - Two or more quizzes in each Topic area
  - Three or more multiple-choice questions in each quiz
  - Three or more possible answers for each question

...More soon

Extra Credit (maximum of +5 points)

Settings screen (+2 points)

- Allows user to change settings before a session
- Settings restored from a file at startup if it exists, defaults otherwise
- Saves settings in a file for next time

Act as a simple “spaced repetition tool” (+3 points)

Similar to the Anki tool.

“Report card” (+3)

- Report of latest scores across all topic quizzes taken.
- Stretch further to include graph of trends for each topic.

Topic/question creation (+2)

- Option from opening client screen
- Q/A editor to create the JSON file for a set of Q&A's to be submitted via email to (root@qzicl.com) for consideration by the *Qzicl* staff. Yes, that email does work - it will be forworded to me.

References

- MSW
- Vitest