BruinFeed Quizzes Technical Design Proposal

Team: server-ihardlyknowher

Author(s): Aparna Hariharan, Keyana Desai, Lauren (Missy) Bridgewater, and Natalie Lord

Objective

We are planning to build a quiz-style web application called BruinFeed Quizzes to give students a fun and engaging way to stay connected to the UCLA community. Unlike familiar student-led project ideas, like a dining hall review app or swipe exchange app, our project offers a twist by combining interactivity, nostalgia, and campus culture. Our primary goals are to use our web server and static file handler to render the frontend of our application, and to unify our design perspectives to create a clean, visually appealing interface. We want to avoid making the app overly complex, aiming instead for an intuitive experience that anyone can quickly understand and enjoy.

Background

This design is intended to be implemented using standard web technologies: HTML for structure, CSS for styling, and JavaScript for interactivity. The core frontend files—index.html, styles.css, and app.js—will be placed in the src directory of our project, allowing for a clean and organized structure. These files will be served using the custom static file handler we implemented in static_file_handler.cc, which efficiently maps mount points to dock roots. To integrate the application into our server architecture, we will add a new route in the dkr_config file, enabling our web server to recognize and correctly serve this new section of the application. This implementation demonstrates how our full-stack system supports the deployment of modular, frontend-driven experiences.

Requirements

The design for BruinFeed Quizzes must support core quiz functionality, including the ability to create quizzes with questions, answer options, and predefined result types. Once a user submits their responses, the platform will compute and display a personalized result instantly. Additionally, users shall be able to share their results through copyable links, encouraging social engagement. The interface will feature a visually fun and engaging design,

incorporating bright colors and recognizable icons. As stated in the Requirements section of the PRD, Accessibility is a key requirement as quizzes will comply with web standards including alt text, keyboard navigation, and color contrast for inclusive use.

The overarching goals of the platform are to create an entertaining, nostalgic experience for UCLA students, foster engagement through campus culture references, and encourage creative quiz contributions. A secondary objective is to provide a foundation for monetization through student-targeted ads. Explicit non-goals include expanding to other campuses, incorporating serious academic material, or collecting any personal data. As stated in the Motivation section of the PRD, these boundaries ensure the product remains simple, safe, and focused on social engagement within the UCLA community.

Development process

As detailed in the “Criteria for Processes” section of the IEEE 12207 standards, we must ensure clear links between outcomes, activities, and tasks, minimize inter-process dependencies, and allow each process to be carried out by a single organization. To ensure that the outcomes, activities, and tasks are strongly related, at each step of development, we will refer to our goals that we have set and ensure our development process is on the right track. If we decided to change one of our initially set goals, we will still ensure that those changes relate to our larger idea and motivation that we had initially set.

To minimize inter-process dependencies, we plan to build a modular application where different components are adaptable on their own without affecting others. For example, we will split up the styling and interaction of the app into their own styles.css and app.js files instead of creating one large html file to render these different aspects. Lastly, to allow each process to be carried out by a single organization, we will strategically split tasks in such a manner that we can complete different aspects in groups of 1-2 people and not have the entire group jointly manage internal tasks of a single process. This will be facilitated by a strong project management process, led by our Tech Lead.

Detailed Design

Quiz Configuration Format

Each quiz will be defined in a JSON file and placed in a static directory (ex. static/quizzes/)

```
{
  "title": "Which UCLA Dining Hall Are You?",
```

```
  "questions": [
    {
      "prompt": "Pick a late night snack:",
      "options": [
        { "text": "Chicken tenders", "value": "de-neve" },
        { "text": "Fruit bowl", "value": "bplate" },
        { "text": "Cookies", "value": "feast" }
      ]
    }
  ],
  "results": {
    "de-neve": {
      "title": "You’re De Neve!",
      "description": "",
      "image": "/images/de-neve.jpg"
    },
    ...
  }
}
```

Request Routing and Configuration

We will add location blocks to the server config to map URI paths to the new handlers.

Both handlers use the quiz_root argument to specify the location of the JSON configuration files. This root is passed during handler initialization and used to locate quizzes and result mappings.

```
location /quiz QuizHandler {
  quiz_root static/quizzes;
}

location /quiz/submit ResultHandler {
  quiz_root static/quizzes;
}

location /quiz/create CreateQuizHandler {
  quiz_root static/quizzes;
}
```

QuizHandler Implementation

The QuizHandler will render the quiz landing page and serve quizzes based on the user. It reads the quiz content from the JSON file and generates an HTML form that has the quiz questions. QuizHandler inherits from the RequestHandler interface and is registered with the factory using the name QuizHandler.

```
class QuizHandler : public RequestHandler {
 public:
  QuizHandler(const std::string& quiz_root);

  static std::unique_ptr<RequestHandler> create(const
std::unordered_map<std::string, std::string>& args);

  response handle_request(const request& req) override;

 private:
  std::string quiz_root_;
};
```

ResultHandler Implementation

The ResultHandler will handle the POST requests to /quiz/submit, evaluate user responses, and return the personalized result page.

```
class ResultHandler : public RequestHandler {
 public:
  ResultHandler(const std::string& quiz_root);

  static std::unique_ptr<RequestHandler> create(const
std::unordered_map<std::string, std::string>& args);

  response handle_request(const request& req) override;

 private:
  std::string quiz_root_;
};
```

The resulting HTML page includes a shareable URL (stateless sharing via query parameters), so users can copy and share their outcome with others.

CreateQuizHandler Implementation

The CreateQuizHandler will allow users to submit their own quizzes via a form. This handler will serve a quiz creation form at GET /quiz/create, accept form data on POST /quiz/create, and save the new quiz as a .json file inside the quiz_root directory.

```
class CreateQuizHandler : public RequestHandler {
 public:
  CreateQuizHandler(const std::string& quiz_root);
  static std::unique_ptr<RequestHandler> create(const
std::unordered_map<std::string, std::string>& args);
  response handle_request(const request& req) override;

 private:
  std::string quiz_root_;
};
```

Static Assets

All quizzes will be in the form static/quizzes/*.json and images will be at static/images/. These are served via StaticFileHandler.

Error Handling

If a quiz name is invalid or the JSON file is missing, the handler will return a 404 using NotFoundHandler. If the form submission is malformed or contains missing fields, a 400 Bad Request is returned.

Alternatives Considered

One alternative was combining both quiz display and result calculation into a single handler. This would have made routing a bit simpler and reduced the number of components. However, we chose to split this functionality into two separate handlers - QuizHandler for displaying the quiz and ResultHandler for processing the results. This modular approach makes the codebase easier to understand, test, and extend.