Quiz Platform - Complete Architecture & Interview Guide

📋 TABLE OF CONTENTS

---

1. PROJECT OVERVIEW

What is this project?

A **timed quiz application** where users can:

- Select from multiple quizzes
- Answer multiple-choice questions
- Get instant scoring feedback
- See detailed results breakdown

Key Features

✅ Quiz Selection (view all available quizzes)
✅ Quiz Attempt (answer questions one-by-one)
✅ Answer Validation (server-side)
✅ Score Calculation (percentage-based)
✅ Result Summary (correct/incorrect breakdown)

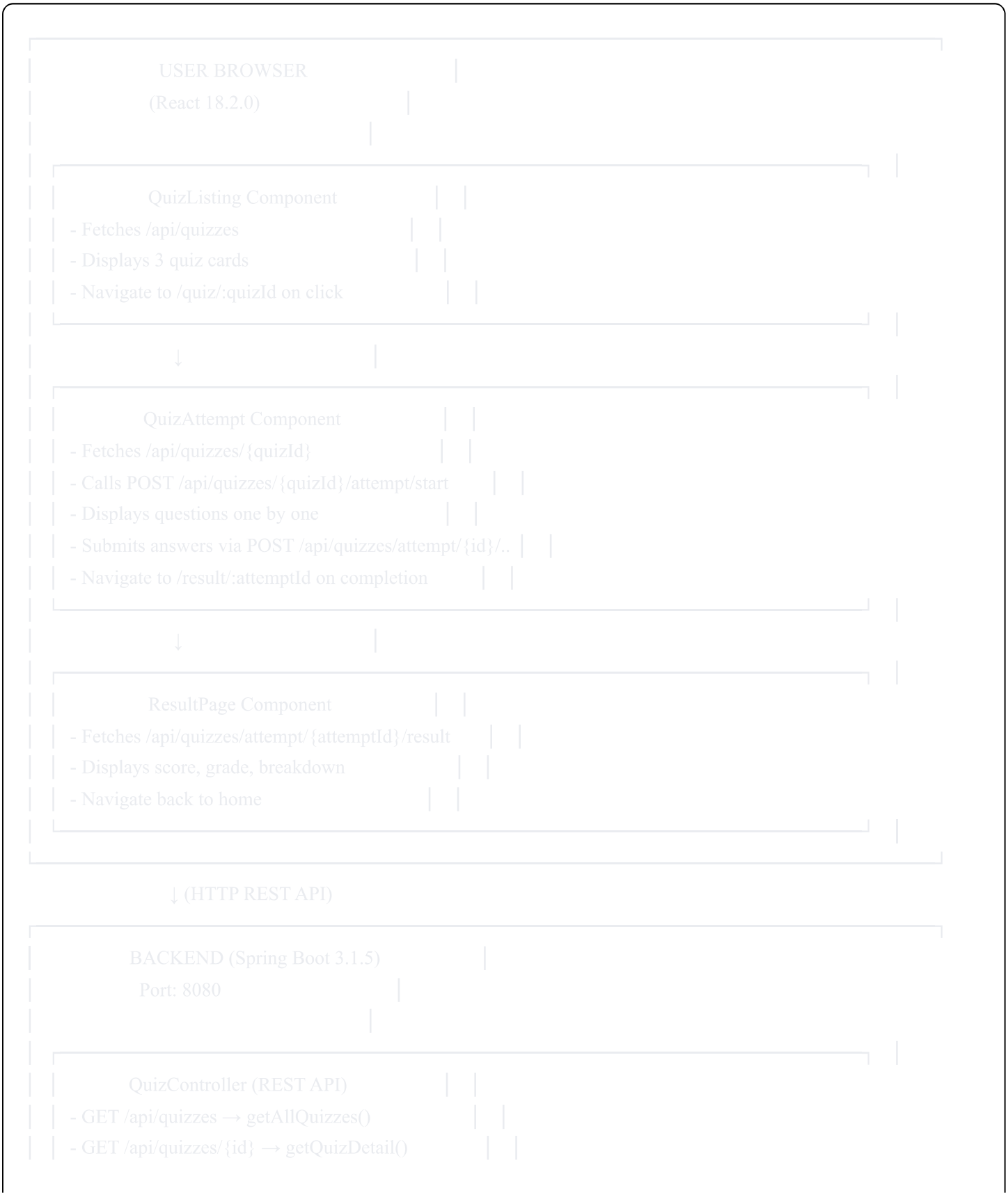
Tech Stack

- **Frontend**: React 18.2.0 (UI rendering, routing, state management)
- **Backend**: Spring Boot 3.1.5 (REST APIs, business logic)

- **Database**: H2 (in-memory relational database)
- **Communication**: REST APIs with JSON/Axios

---

2. ARCHITECTURE DIAGRAM



- POST /api/quizzes/{id}/attempt/start → startAttempt()
- POST /api/quizzes/attempt/{id}/response → submitResp()
- POST /api/quizzes/attempt/{id}/submit → submitQuiz()
- GET /api/quizzes/attempt/{id}/result → getResult()
Service Layer (Business Logic)
QuizService:
- getAllQuizzes(): Fetch all quizzes + count questions
- getQuizDetail(id): Fetch quiz + questions (no answers)
AttemptService:
- startAttempt(): Create UserAttempt record
- submitResponse(): Store user's answer
- submitQuiz(): Calculate final score
- getResult(): Retrieve attempt results
Repository Layer (Data Access)
- QuizRepository
- QuestionRepository
- OptionRepository
- UserAttemptRepository
- ResponseRepository
(All extend JpaRepository with custom queries)
↓ (JPA/Hibernate ORM)
DATABASE (H2 In-Memory)
jdbc:h2:mem:quiz_db
quizzes
id (PK)
title
description
duration
questions
id (PK)
text
quiz_id(FK)
options
id (PK)
text
is_correct
question_id
user_attempts
id (PK)
user_id
quiz_id(FK)
start_time

---

3. TECHNOLOGY STACK EXPLANATION

FRONTEND: React 18.2.0

Why React?

- Component-based architecture (reusable UI)
- Virtual DOM (efficient re-rendering)
- State management (track quiz progress)
- React Router (handle navigation)

Key Libraries:

- `axios` - HTTP client for API calls
- `react-router-dom` - Client-side routing

How it works:

```
User Action (click)
    ↓
Component State Updated
    ↓
API Call (axios)
    ↓
Response Received
    ↓
State Updated
    ↓
Component Re-renders (Virtual DOM)
    ↓
DOM Updated (only changed parts)
    ↓
User sees updated UI
```

BACKEND: Spring Boot 3.1.5

Why Spring Boot?

- Auto-configuration (reduce boilerplate)
- Built-in Tomcat server
- Dependency injection (loose coupling)
- Transaction management (data consistency)
- Easy API development (Spring Web MVC)

Key Components:

- `spring-boot-starter-web` - REST API support
- `spring-boot-starter-data-jpa` - Database ORM
- `h2` - In-memory database (for development)

How it works:

```
HTTP Request (GET /api/quizzes)
    ↓
DispatcherServlet routes to Controller
    ↓
Controller method called
    ↓
Service layer processes business logic
    ↓
Repository queries database
    ↓
Database returns data
    ↓
Data mapped to DTO
    ↓
JSON response sent to client
```

DATABASE: H2

Why H2?

- In-memory (no setup required)
- Perfect for development/testing
- JDBC-compliant
- Lightweight

Configuration:

```yaml
url: jdbc:h2:mem:quiz_db  # In-memory database
username: sa
password: (empty)
```

---

4. DETAILED DATA FLOW

Flow 1: Load Quiz List

```
USER CLICKS: Browser loads http://localhost:3000
    ↓
FRONTEND: useEffect() in QuizListing.jsx triggers
```

```
    ↓
AXIOS: GET http://localhost:8080/api/quizzes
    ↓
BACKEND: QuizController.getAllQuizzes()
    ↓
SERVICE: QuizService.getAllQuizzes()
    ↓
REPOSITORY: quizRepository.findAll()
    ↓
DATABASE QUERY:
    SELECT q.id, q.title, q.description, q.duration
    FROM quizzes q
    ↓
HIBERNATE: Maps ResultSet to Quiz entities
    ↓
SERVICE: Maps Quiz entities to QuizListDTO objects
    ↓
RESPONSE: JSON array with 3 quizzes
    ↓
FRONTEND: setQuizzes(response.data)
    ↓
UI: Renders 3 quiz cards
```

Flow 2: Start Quiz Attempt

```
USER CLICKS: "Start Quiz" on JavaScript Fundamentals
    ↓
FRONTEND: Navigates to /quiz/1
    ↓
QuizAttempt.jsx useEffect() triggers
    ↓
AXIOS 1: GET http://localhost:8080/api/quizzes/1
    ↓
BACKEND: QuizService.getQuizDetail(1)
    ↓
DATABASE:
    SELECT q.* FROM quizzes q WHERE q.id = 1
    SELECT qu.* FROM questions qu WHERE qu.quiz_id = 1
    SELECT o.* FROM options o WHERE o.question_id IN (1, 2)
    ↓
BACKEND: Maps data to QuizDetailDTO (with null isCorrect)
    ↓
RESPONSE: JSON with quiz and questions (no correct answers)
```

```
  ↓
FRONTEND: setQuiz(response.data)
  ↓
AXIOS 2: POST http://localhost:8080/api/quizzes/1/attempt/start?userId=user_1234567890
  ↓
BACKEND: AttemptService.startAttempt(1, "user_1234567890")
  ↓
SERVICE:
  1. Check for duplicate attempts
     SELECT * FROM user_attempts
     WHERE user_id = "user_1234567890"
     AND quiz_id = 1
     AND end_time IS NULL

  2. If no duplicate, create new UserAttempt:
     INSERT INTO user_attempts (user_id, quiz_id, start_time, score)
     VALUES ("user_1234567890", 1, NOW(), 0)

  3. Return AttemptStartDTO with attemptId
  ↓
RESPONSE: {attemptId: 5, quizId: 1, startTime: ..., endTime: ...}
  ↓
FRONTEND: setAttemptId(5)
  ↓
UI: Shows first question with radio buttons for options
```

Flow 3: Submit Answer

```
USER SELECTS: "const x = 5;" (option_id: 3)
  ↓
USER CLICKS: "Next" button
  ↓
FRONTEND: axios.post(/api/quizzes/attempt/5/response, {
  questionId: 1,
  selectedOptionId: 3
})
  ↓
BACKEND: AttemptService.submitResponse(5, request)
  ↓
SERVICE:
  1. Fetch UserAttempt with id=5
  2. Fetch Option with id=3
  3. Create Response object
```

```
   4. INSERT INTO responses:
      INSERT INTO responses (attempt_id, question_id, selected_option_id)
      VALUES (5, 1, 3)

   5. Check if option is correct:
      SELECT is_correct FROM options WHERE id = 3
      → is_correct = false

   6. Return ResponseSubmitDTO {responseId: 1, isCorrect: false, ...}
   ↓
RESPONSE: {responseId: 1, isCorrect: false, message: "Recorded"}
   ↓
FRONTEND:
   if (idx < totalQuestions - 1) {
      setIdx(idx + 1)  // Show next question
   } else {
      submit quiz  // Go to step 4
   }
   ↓
UI: Shows question 2
```

Flow 4: Submit Quiz

```
USER CLICKS: "Finish" button (after last question)
   ↓
FRONTEND: axios.post(/api/quizzes/attempt/5/submit)
   ↓
BACKEND: AttemptService.submitQuiz(5)
   ↓
SERVICE:
   1. Fetch UserAttempt with id=5
   2. Fetch all responses for attempt 5:
      SELECT r.* FROM responses r
      WHERE r.attempt_id = 5
      → Returns 2 responses

   3. Calculate score:
      For each response, check if selected option is correct
      Response 1: option_id=3, is_correct=false ❌
      Response 2: option_id=8, is_correct=true ✓

      correctAnswers = 1
      totalQuestions = 2
```

```
      score = (1 * 100) / 2 = 50%

   4. Update UserAttempt:
      UPDATE user_attempts
      SET end_time = NOW(), score = 50
      WHERE id = 5

   5. Build ResultDTO with detailed breakdown
   ↓
RESPONSE: {
   attemptId: 5,
   score: 50,
   correctAnswers: 1,
   totalQuestions: 2,
   incorrectAnswers: 1,
   details: [
      {questionId: 1, questionText: "...", userAnswer: "const x = 5", correctAnswer: "All of the above", isCorrect: false},
      {questionId: 2, questionText: "...", userAnswer: "Alphabet", correctAnswer: "Alphabet", isCorrect: true}
   ]
}
   ↓
FRONTEND: navigate(`/result/5`)
   ↓
ResultPage.jsx useEffect() triggers
   ↓
AXIOS: GET /api/quizzes/attempt/5/result
   ↓
BACKEND: AttemptService.getResult(5)
   ↓
RESPONSE: Same ResultDTO as above
   ↓
FRONTEND: setResult(response.data)
   ↓
UI:
   - Shows score: 50%
   - Shows grade: C
   - Shows breakdown: 1 correct, 1 incorrect
   - Shows button to go back home
```

5. CODE WALKTHROUGH

Frontend: QuizListing.jsx

```javascript
function QuizListing() {
  const [quizzes, setQuizzes] = useState([]);  // State to store quizzes
  const [loading, setLoading] = useState(true);
  const navigate = useNavigate();

  // Fetch quizzes when component mounts
  useEffect(() => {
    axios.get("http://localhost:8080/api/quizzes")
      .then(r => setQuizzes(r.data))      // Update state with response
      .finally(() => setLoading(false));  // Stop loading
  }, []);  // Empty dependency array = run once on mount

  // Render quiz cards
  return (
    <div>
      {quizzes.map(q => (
        <div key={q.id}>
          <h2>{q.title}</h2>
          <p>{q.description}</p>
          <button onClick={() => navigate(`/quiz/${q.id}`)}>
            Start Quiz
          </button>
        </div>
      ))}
    </div>
  );
}
```

Key Concepts:

- `useState` - React hook to manage state
- `useEffect` - Run side effects (API calls)
- `axios` - Make HTTP requests
- `.map()` - Render list of items
- `navigate` - Change URL (React Router)

Backend: QuizController.java

java

```java
@RestController
@RequestMapping("/api/quizzes")
@CrossOrigin(origins = "*")  // Allow requests from frontend
public class QuizController {

  @Autowired
  private QuizService quizService;

  @GetMapping  // Handle GET /api/quizzes
  public ResponseEntity<List<QuizListDTO>> getAllQuizzes() {
    return ResponseEntity.ok(quizService.getAllQuizzes());
    // Returns 200 OK with list of QuizListDTO
  }

  @GetMapping("/{quizId}")  // Handle GET /api/quizzes/{quizId}
  public ResponseEntity<QuizDetailDTO> getQuizDetail(@PathVariable Long quizId) {
    return ResponseEntity.ok(quizService.getQuizDetail(quizId));
    // Returns quiz detail with questions but no correct answers
  }
}
```

Key Concepts:

- `@RestController` - Creates REST API endpoint
- `@GetMapping` - Maps HTTP GET requests
- `@PathVariable` - Extract value from URL
- `ResponseEntity` - Control HTTP response (status code, body)
- `@CrossOrigin` - Allow requests from other domains

Backend: AttemptService.java

java

```
public AttemptStartDTO startAttempt(Long quizId, String userId) {
  // Check for duplicate attempts
  userAttemptRepository
    .findByUserIdAndQuizIdAndEndTimeIsNull(userId, quizId)
    .ifPresent(attempt -> {
     throw new RuntimeException("Active attempt exists");
    });

  // Fetch quiz
  Quiz quiz = quizRepository.findById(quizId)
    .orElseThrow(() -> new RuntimeException("Quiz not found"));

  // Create new attempt
  UserAttempt attempt = new UserAttempt();
  attempt.setUserId(userId);
  attempt.setQuiz(quiz);
  attempt.setStartTime(LocalDateTime.now());
  attempt.setScore(0);

  // Save to database
  UserAttempt saved = userAttemptRepository.save(attempt);

  // Return DTO
  return new AttemptStartDTO(
    saved.getId(),
    quiz.getId(),
    saved.getStartTime(),
    LocalDateTime.now().plusSeconds(quiz.getDuration() * 60)
  );
}
```

Key Concepts:

- `Optional.ifPresent()` - Check if value exists
- `orElseThrow()` - Throw exception if not found
- `LocalDateTime` - Date/time handling
- Save and return DTO - Don't expose entities to client

Backend: Score Calculation

java

```
public ResultDTO submitQuiz(Long attemptId) {
  // Fetch attempt and responses
  UserAttempt attempt = userAttemptRepository.findById(attemptId)
    .orElseThrow();
  List<Response> responses = responseRepository.findByAttempt(attempt);

  // Calculate score
  int totalQuestions = Math.toIntExact(questionRepository.countByQuiz(attempt.getQuiz()));
  int correctAnswers = (int) responses.stream()
    .filter(r -> r.getSelectedOption().isCorrect())  // Count correct ones
    .count();

  // Formula: (correct / total) * 100
  int score = (totalQuestions > 0) ? (correctAnswers * 100) / totalQuestions : 0;

  // Update attempt
  attempt.setEndTime(LocalDateTime.now());
  attempt.setScore(score);
  userAttemptRepository.save(attempt);

  // Build result with details
  return buildResult(attempt, responses, totalQuestions, correctAnswers);
}
```

Key Concepts:

- `stream().filter()` - Functional programming to filter data
- `.count()` - Count filtered items
- Integer division - (1 * 100) / 2 = 50 (not 50.0)
- Transaction safety - Update attempt and calculate score atomically

Database: JPA Entity Relationships

java

```
@Entity
public class Quiz {
  @Id
  @GeneratedValue(strategy = GenerationType.IDENTITY)
  private Long id;

  private String title;

  @OneToMany(mappedBy = "quiz")
  private List<Question> questions;  // One quiz has many questions
}

@Entity
public class Question {
  @Id
  @GeneratedValue(strategy = GenerationType.IDENTITY)
  private Long id;

  private String text;

  @ManyToOne
  @JoinColumn(name = "quiz_id")
  private Quiz quiz;  // Many questions belong to one quiz

  @OneToMany(mappedBy = "question")
  private List<Option> options;  // One question has many options
}

@Entity
public class Option {
  @Id
  @GeneratedValue(strategy = GenerationType.IDENTITY)
  private Long id;

  private String text;
  private boolean isCorrect;

  @ManyToOne
  @JoinColumn(name = "question_id")
  private Question question;
}
```

Key Concepts:

- `@OneToMany` - One entity has many of another
- `@ManyToOne` - Many entities belong to one
- `@JoinColumn` - Foreign key column
- `mappedBy` - Define relationship from other side
- `cascade` - Automatically delete related records

---

6. KEY DESIGN PATTERNS

1. MVC Pattern (Model-View-Controller)

```
Frontend (View): React components render UI
    ↓
Backend (Controller): QuizController handles HTTP requests
    ↓
Backend (Service): QuizService/AttemptService contain business logic
    ↓
Backend (Model): Quiz, Question, Option entities represent data
    ↓
Database: H2 stores persistent data
```

2. DTO (Data Transfer Object) Pattern

```java
```

```
// Entity (Database representation)
@Entity
public class Quiz {
  private Long id;
  private String title;
  private String description;
  private int duration;
  @OneToMany
  private List<Question> questions;  // Unnecessary for listing
}

// DTO (API response representation)
public class QuizListDTO {
  public Long id;
  public String title;
  public String description;
  public int duration;
  public int questionCount;  // Aggregate instead of List
}

// Why?
// - Don't expose entities directly (security)
// - Decouple API from database schema
// - Only send necessary data
// - Avoid circular references
```

3. Repository Pattern

java

```
// Instead of writing raw SQL:
// String sql = "SELECT * FROM quizzes WHERE id = ?";
// ResultSet rs = stmt.executeQuery(sql);

// Use Spring Data JPA:
public interface QuizRepository extends JpaRepository<Quiz, Long> {
  // Automatically generates:
  // - findById(Long id)
  // - findAll()
  // - save(Quiz quiz)
  // - delete(Quiz quiz)
  // etc.
}

// Benefits:
// - Less boilerplate
// - Automatic SQL generation
// - Type-safe queries
// - Easy to mock in tests
```

4. Service Layer Pattern

java

```
// DON'T do this in Controller:
@GetMapping
public Quiz getQuiz(Long id) {
  return quizRepository.findById(id).orElse(null);
}

// DO this with Service layer:
@GetMapping
public QuizDetailDTO getQuiz(Long id) {
  return quizService.getQuizDetail(id);
}

// Service contains:
public class QuizService {
  public QuizDetailDTO getQuizDetail(Long quizId) {
    // 1. Fetch quiz
    // 2. Fetch questions
    // 3. Fetch options
    // 4. Validate
    // 5. Transform to DTO
    // 6. Return
  }
}

// Benefits:
// - Business logic in one place
// - Reusable across controllers
// - Easy to unit test
// - Controller stays thin
```

5. Transaction Management

java

```
@Service
@Transactional  // All methods are transactional
public class AttemptService {

  public ResultDTO submitQuiz(Long attemptId) {
    // Step 1: Fetch attempt
    UserAttempt attempt = userAttemptRepository.findById(attemptId).orElseThrow();

    // Step 2: Calculate score
    int score = calculateScore(attempt);

    // Step 3: Update attempt
    attempt.setScore(score);
    attempt.setEndTime(LocalDateTime.now());
    userAttemptRepository.save(attempt);

    // If any exception occurs between Step 1-3,
    // ALL changes are ROLLED BACK
    // Database remains consistent
  }
}
```

---

7. DATABASE DESIGN

Schema with Relationships

```
quizzes (1)
  ├─ id (PRIMARY KEY)
  ├─ title
  ├─ description
  └─ duration
     ↓ (1:N relationship)

questions (N)
  ├─ id (PRIMARY KEY)
  ├─ quiz_id (FOREIGN KEY → quizzes.id) ON DELETE CASCADE
  └─ text
     ↓ (1:N relationship)

options (N)
  ├─ id (PRIMARY KEY)
```

```
  ├─ question_id (FOREIGN KEY → questions.id) ON DELETE CASCADE
  ├─ text
  └─ is_correct

user_attempts (N)
  ├─ id (PRIMARY KEY)
  ├─ user_id
  ├─ quiz_id (FOREIGN KEY → quizzes.id) ON DELETE CASCADE
  ├─ start_time
  ├─ end_time
  └─ score
    ↓ (1:N relationship)

responses (N)
  ├─ id (PRIMARY KEY)
  ├─ attempt_id (FOREIGN KEY → user_attempts.id) ON DELETE CASCADE
  ├─ question_id (FOREIGN KEY → questions.id) ON DELETE CASCADE
  ├─ selected_option_id (FOREIGN KEY → options.id) ON DELETE CASCADE
  └─ UNIQUE(attempt_id, question_id)  // One answer per question
```

Why CASCADE DELETE?

sql

```sql
DELETE FROM quizzes WHERE id = 1;

-- Cascades to:
DELETE FROM questions WHERE quiz_id = 1;
DELETE FROM options WHERE question_id IN (1, 2);
DELETE FROM user_attempts WHERE quiz_id = 1;
DELETE FROM responses WHERE attempt_id IN (...);

-- Without CASCADE, deletion would fail (Foreign Key Constraint)
```

Indexes for Performance

sql

```
CREATE INDEX idx_quiz_questions ON questions(quiz_id);
-- Speed up: SELECT * FROM questions WHERE quiz_id = 1

CREATE INDEX idx_user ON user_attempts(user_id);
-- Speed up: SELECT * FROM user_attempts WHERE user_id = "user_123"

CREATE INDEX idx_quiz_attempts ON user_attempts(quiz_id);
-- Speed up: SELECT * FROM user_attempts WHERE quiz_id = 1
```

---

8. COMMON INTERVIEW QUESTIONS & ANSWERS

Architecture & Design Questions

Q1: "Explain the flow from user selecting a quiz to seeing the first question"

Answer:

1. User clicks "Start Quiz" button on React frontend
2. Frontend navigates to `/quiz/{quizId}`
3. QuizAttempt.jsx component mounts
4. Two parallel API calls:
   - GET /api/quizzes/{quizId} → Fetch quiz and questions
   - POST /api/quizzes/{quizId}/attempt/start → Create attempt record
5. Backend creates UserAttempt record in database with start_time
6. Frontend receives attemptId from backend
7. Frontend displays first question with options
8. User can now select an answer

Why two API calls?

- First call gets the quiz structure
- Second call creates the attempt record (tracks when user started)
- Allows backend to enforce time limits

Q2: "Why use DTOs instead of sending entities directly?"

Answer:

java

```
// WRONG: Send entity directly
@GetMapping("/{quizId}")
public Quiz getQuiz(Long quizId) {
  return quizRepository.findById(quizId).get();
}
// Problems:
// 1. Exposes entire object structure to client
// 2. Circular references (Quiz → Questions → Quiz)
// 3. Security: What if entity had sensitive fields?
// 4. Tight coupling: API breaks if entity changes
// 5. Unnecessary data: Client gets all fields even if unused

// RIGHT: Use DTO
@GetMapping("/{quizId}")
public QuizDetailDTO getQuiz(Long quizId) {
  return quizService.getQuizDetail(quizId);
}

public class QuizDetailDTO {
  public Long id;
  public String title;
  public int duration;
  public List<QuestionDTO> questions;
  // Doesn't expose: internal IDs, relationships, etc.
}

// Benefits:
// 1. API contract is explicit
// 2. Serialize/deserialize is simple (no circular refs)
// 3. Can map multiple entities to one DTO
// 4. Easy to evolve API without breaking client
// 5. Security: Hide sensitive fields
```

Q3: "How do you ensure correct answers are not revealed during quiz?"

Answer:

java

```
// In QuizService.getQuizDetail():
List<QuestionDTO> questions = questionRepository.findByQuiz(quiz)
  .stream()
  .map(question -> new QuestionDTO(
    question.getId(),
    question.getText(),
    optionRepository.findByQuestion(question)
      .stream()
      .map(option -> new OptionDTO(
        option.getId(),
        option.getText(),
        null  // ← Set to null instead of option.isCorrect()
      ))
      .collect(Collectors.toList())
  ))
  .collect(Collectors.toList());

// Frontend receives:
{
  "id": 1,
  "text": "What is correct?",
  "options": [
    {"id": 1, "text": "Option A", "isCorrect": null},
    {"id": 2, "text": "Option B", "isCorrect": null}
  ]
}

// When user submits response:
// Backend checks: Option.isCorrect from database (client can't fake it)
// Response.isCorrect = option.isCorrect  // From DB, not client
```

Key Point: Never trust the client. Always validate on server.

Q4: "What happens if a user tries to submit the same answer twice?"

Answer:

```
java
```

```
// In AttemptService.submitResponse():
Optional<Response> existing = responseRepository
  .findByAttemptAndQuestion(attempt, questionRepository.findById(request.questionId).orElseThrow());

if (existing.isPresent()) {
  throw new RuntimeException("Response already exists");
}

// UNIQUE constraint in database:
CREATE UNIQUE INDEX unique_response ON responses(attempt_id, question_id);

// This prevents duplicate rows:
INSERT INTO responses (attempt_id, question_id, selected_option_id)
VALUES (5, 1, 3);  // Success

INSERT INTO responses (attempt_id, question_id, selected_option_id)
VALUES (5, 1, 4);  // FAILS - UNIQUE constraint violated
```

Layered Protection:

1. Service layer checks before insert
2. Database unique constraint prevents duplicates
3. Both layers ensure data consistency

Q5: "How is the score calculated?"

Answer:

```
java
```

```
public ResultDTO submitQuiz(Long attemptId) {
  // Get all responses for this attempt
  List<Response> responses = responseRepository.findByAttempt(attempt);

  // Count correct answers
  int correctAnswers = (int) responses.stream()
    .filter(r -> r.getSelectedOption().isCorrect())
    .count();
  // Example: [false, true, true] → count = 2

  // Get total questions
  int totalQuestions = Math.toIntExact(questionRepository.countByQuiz(attempt.getQuiz()));
  // Example: 3

  // Calculate percentage
  int score = (correctAnswers * 100) / totalQuestions;
  // Formula: (2 * 100) / 3 = 200 / 3 = 66 (integer division)

  attempt.setScore(score);
  return resultDTO;
}

// Note: Integer division (not float)
// 2 * 100 / 3 = 66 (not 66.67)
// This is intentional for percentage representation
```

Technical Implementation Questions

Q6: "Why use Spring @Transactional annotation?"

Answer:

```
java
```

```java
@Service
@Transactional  // ← What does this do?
public class AttemptService {

  public ResultDTO submitQuiz(Long attemptId) {
    // Step 1
    UserAttempt attempt = userAttemptRepository.findById(attemptId).orElseThrow();

    // Step 2
    List<Response> responses = responseRepository.findByAttempt(attempt);

    // Step 3
    int score = calculateScore(responses);

    // Step 4
    attempt.setScore(score);
    attempt.setEndTime(LocalDateTime.now());
    userAttemptRepository.save(attempt);

    // Step 5
    return buildResult(attempt);
  }
}

// @Transactional ensures:
// 1. All steps execute in a single transaction
// 2. If any step fails → all changes ROLLBACK
// 3. Database remains consistent

// Example: What if step 4 fails?
// Without @Transactional:
//   - Responses are saved (in previous calls)
//   - UserAttempt is not updated
//   - Database is INCONSISTENT
//
// With @Transactional:
//   - All changes are rolled back
//   - Database is CONSISTENT

// Isolation Levels:
// READ_UNCOMMITTED - Can read dirty data
// READ_COMMITTED - Only read committed data (default)
```

```
// REPEATABLE_READ - Prevents dirty reads and non-repeatable reads
// SERIALIZABLE - Highest isolation but slower
```

Q7: "What if two users try to attempt the same quiz simultaneously?"

Answer:

```
Timeline:
T1: User A starts quiz → INSERT user_attempts → attempt_id: 10
T2: User B starts quiz → INSERT user_attempts → attempt_id: 11

// No conflict! Each user has separate attempt record
// Both can answer independently

T3: User A submits response to Q1
    INSERT responses (attempt_id=10, question_id=1, selected_option_id=3)

T4: User B submits response to Q1
    INSERT responses (attempt_id=11, question_id=1, selected_option_id=4)

// No conflict! Responses are tracked by attempt_id

T5: User A submits quiz
    SELECT * FROM responses WHERE attempt_id = 10
    // Gets only User A's responses
    // Calculates User A's score independently

T6: User B submits quiz
    SELECT * FROM responses WHERE attempt_id = 11
    // Gets only User B's responses
    // Calculates User B's score independently

// Result: Both users are isolated
// No race conditions, no data corruption
```

Database Locking:

By default, H2 uses:
- Row-level locking for UPDATE/DELETE
- Table-level locking for INSERT

With @Transactional:
- Multiple transactions don't interfere
- Read-Committed isolation (default)
- Prevents dirty reads and lost updates

Q8: "How does the frontend know when to submit the quiz?"

Answer:

javascript

```
// QuizAttempt.jsx
const submit = async () => {
  if (!ans[quiz.questions[idx].id]) {
    alert("Select answer");
    return;
  }

  // Submit current answer
  await axios.post(`.../response`, {
    questionId: quiz.questions[idx].id,
    selectedOptionId: ans[quiz.questions[idx].id]
  });

  // Check if this is the last question
  if (idx < quiz.questions.length - 1) {
    setIdx(idx + 1);  // Go to next question
  } else {
    // This is the last question
    await axios.post(`.../${attemptId}/submit`);
    navigate(`/result/${attemptId}`);
  }
};

// Flow:
// Q1 → User answers → Click "Next" → idx becomes 1
// Q2 → User answers → Click "Next" → idx becomes 2
// Q3 → User answers → Click "Finish" → Submit quiz
```

Q9: "What's missing from this application?"

Answer:

High Priority:

1. Timer Implementation
   - Frontend countdown timer
   - Auto-submit when time expires
   - Server-side validation of time

2. Error Handling
   - Network error messages
   - Proper HTTP error responses
   - User-friendly error UI

3. Input Validation
   - Validate request data
   - Check quiz_id exists
   - Check user_id format

Medium Priority:

4. Authentication/Authorization
   - User login
   - User identification (not just timestamp)
   - Prevent cheating (verify user matches token)

5. Question Navigation
   - Previous button
   - Jump to specific question
   - Review before submitting

6. UI/UX
   - CSS framework (Bootstrap/Tailwind)
   - Loading spinners
   - Mobile responsiveness
   - Animations

Low Priority:

7. Analytics
   - Track user performance
   - Question difficulty analysis
   - Most missed questions

8. Admin Panel
   - Add new quizzes
   - Edit questions
   - View statistics

```
9. WebSocket Support
   - Real-time timer sync
   - Live leaderboard
   - Proctor notifications
```

Q10: "How would you scale this application?"

Answer:

Current State:
- Single process (one Spring Boot instance)
- In-memory database (data lost on restart)
- No load balancing
- Can handle ~100 concurrent users

Scaling Strategy:

LAYER 1: Database
├─ Move from H2 to PostgreSQL
├─ Implement connection pooling (HikariCP - already done)
├─ Add database replication (master-slave)
├─ Sharding by user_id for horizontal scaling
└─ Caching layer (Redis) for frequently accessed quizzes

LAYER 2: Backend
├─ Run multiple Spring Boot instances
├─ Load balancer (Nginx) to distribute requests
├─ Sticky sessions for user session affinity
├─ Async processing (Spring Async) for heavy computations
├─ Messaging queue (RabbitMQ) for score calculations
└─ Monitoring (Prometheus, Grafana)

LAYER 3: Frontend
├─ CDN for static assets
├─ Service Worker for offline support
├─ Progressive Web App (PWA)
└─ Code splitting to reduce bundle size

LAYER 4: Infrastructure
├─ Docker containers
├─ Kubernetes orchestration
├─ Auto-scaling based on CPU/Memory
├─ Multi-region deployment
└─ CI/CD pipeline (Jenkins, GitLab CI)

Example: Handle 10,000 concurrent users
- Database: 50 PostgreSQL replicas
- Application: 20 Spring Boot instances
- Load Balancer: 2x Nginx (redundancy)
- Cache: Redis cluster (3 nodes)

- Message Queue: RabbitMQ cluster
- CDN: CloudFlare

Q11: "What's the difference between @Service and regular class?"

Answer:

java

```
// Without @Service
public class QuizService {
  private QuizRepository quizRepository;

  public List<QuizListDTO> getAllQuizzes() {
    return quizRepository.findAll();
  }
}

// Problem: Need to manually create instance
QuizService service = new QuizService();
service.setQuizRepository(new QuizRepository());

// With @Service
@Service
public class QuizService {
  @Autowired
  private QuizRepository quizRepository;

  public List<QuizListDTO> getAllQuizzes() {
    return quizRepository.findAll();
  }
}

// Spring automatically:
// 1. Creates instance (singleton)
// 2. Injects dependencies (@Autowired)
// 3. Registers in ApplicationContext
// 4. Manages lifecycle

// Benefits:
// 1. Dependency Injection (loose coupling)
// 2. Singleton pattern (one instance shared)
// 3. Easy to mock for testing
// 4. Lifecycle hooks (@PostConstruct, @PreDestroy)
```

Q12: "Why is the userId generated as a timestamp?"

Answer:

```
javascript
```

```
// Current implementation:
const uid = "user_" + Date.now();

// Problems:
// 1. Not persistent
//    - Same user gets different ID each attempt
//    - Can't track user history
//    - Timestamps can collide (multiple users simultaneously)

// 2. Security issue
//    - Guessing user IDs is easy
//    - Anyone can claim to be "user_123456"

// 3. Data integrity
//    - Can't prevent duplicate attempts per user per quiz

// Better implementation:
// 1. Database-driven user management
@Entity
public class User {
  @Id
  private String userId;  // UUID or email
  private String name;
  private String password;  // hashed
  private LocalDateTime createdAt;
}

// 2. Authentication
// POST /login with email + password
// Returns JWT token
String token = jwtTokenProvider.generateToken(user);

// 3. Use token in requests
axios.post("/api/quizzes/1/attempt/start", {}, {
  headers: {
    Authorization: `Bearer ${token}`
  }
});

// 4. Extract user from token
@PostMapping("/{quizId}/attempt/start")
public ResponseEntity<AttemptStartDTO> startAttempt(
  @PathVariable Long quizId,
```

```
  @RequestHeader("Authorization") String token
) {
  String userId = jwtTokenProvider.getUserIdFromToken(token);
  return ResponseEntity.ok(attemptService.startAttempt(quizId, userId));
}
```

Q13: "How would you prevent answer tampering?"

Answer:

java

```
// User could intercept request and change selected_option_id
// Example:
POST /api/quizzes/attempt/5/response
{
  questionId: 1,
  selectedOptionId: 3  // User changes to correct answer
}

// How to prevent:

// Method 1: Server-side validation
@PostMapping("/attempt/{attemptId}/response")
public ResponseEntity<?> submitResponse(
  @PathVariable Long attemptId,
  @RequestBody ResponseSubmitRequestDTO request
) {
  // Verify:
  // 1. Option exists
  Option option = optionRepository.findById(request.selectedOptionId)
    .orElseThrow(() -> new IllegalArgumentException("Invalid option"));

  // 2. Option belongs to the question
  Question question = questionRepository.findById(request.questionId)
    .orElseThrow(() -> new IllegalArgumentException("Invalid question"));

  if (!option.getQuestion().getId().equals(question.getId())) {
    throw new IllegalArgumentException("Option doesn't belong to question");
  }

  // 3. Question belongs to the quiz
  UserAttempt attempt = userAttemptRepository.findById(attemptId)
    .orElseThrow();

  if (!question.getQuiz().getId().equals(attempt.getQuiz().getId())) {
    throw new IllegalArgumentException("Question doesn't belong to quiz");
  }

  // 4. Verify user is authenticated (JWT token)
  // String userId = extractUserFromToken();
  // if (!userId.equals(attempt.getUserId())) ...

  // If all validations pass, save response
  return attemptService.submitResponse(attemptId, request);
```

```
}

// Method 2: Checksum verification
// Frontend sends checksum of correct option
POST /api/quizzes/attempt/5/response
{
  questionId: 1,
  selectedOptionId: 3,
  checksum: "abc123def456"  // Hash of option data + timestamp
}

// Backend verifies checksum matches option data

// Method 3: Time-based tokens
// Option IDs are encrypted with time-based token
// Token expires after quiz duration
// Prevents reuse of tokens from old quizzes
```

Q14: "What's the difference between @OneToMany and @ManyToOne?"

Answer:

```
java
```

```
// @OneToMany: One parent has many children
@Entity
public class Quiz {
  @Id
  private Long id;

  @OneToMany(mappedBy = "quiz")  // Questions reference back to quiz
  private List<Question> questions;  // One quiz → Many questions
}

// @ManyToOne: Many children belong to one parent
@Entity
public class Question {
  @Id
  private Long id;

  @ManyToOne
  @JoinColumn(name = "quiz_id")
  private Quiz quiz;  // Many questions → One quiz
}

// Database:
questions table:
  id | quiz_id | text
  1  | 1       | "What is JS?"
  2  | 1       | "Is JS typed?"
  3  | 2       | "What is React?"

// Retrieving:
// Using @OneToMany:
Quiz quiz = quizRepository.findById(1);
List<Question> questions = quiz.getQuestions();  // [Q1, Q2]

// Using @ManyToOne:
Question question = questionRepository.findById(1);
Quiz quiz = question.getQuiz();  // Quiz 1

// Best Practice:
// - Use @ManyToOne when querying children
// - Use @OneToMany when querying parent
// - Define relationship from both sides
// - Use mappedBy on @OneToMany (avoids foreign key duplication)
```

Q15: "Explain the Repository Pattern with a real example"

Answer:

java

```
// Without Repository Pattern (Manual SQL):
public class QuizDAO {
  private DataSource dataSource;

  public Quiz findById(Long id) throws SQLException {
    Connection conn = dataSource.getConnection();
    String sql = "SELECT * FROM quizzes WHERE id = ?";
    PreparedStatement stmt = conn.prepareStatement(sql);
    stmt.setLong(1, id);
    ResultSet rs = stmt.executeQuery();

    if (rs.next()) {
      Quiz quiz = new Quiz();
      quiz.setId(rs.getLong("id"));
      quiz.setTitle(rs.getString("title"));
      quiz.setDuration(rs.getInt("duration"));
      return quiz;
    }

    stmt.close();
    conn.close();
    return null;
  }

  public void save(Quiz quiz) throws SQLException {
    // More boilerplate...
  }

  public void delete(Long id) throws SQLException {
    // More boilerplate...
  }
}

// Problem:
// - Lots of boilerplate
// - Error handling complex
// - Hard to test
// - Coupling to SQL

// With Repository Pattern:
public interface QuizRepository extends JpaRepository<Quiz, Long> {
  List<Quiz> findByTitle(String title);
  List<Quiz> findByDuration(int duration);
```

```
}

// Spring automatically generates:
quizRepository.findById(1);          // SELECT * FROM quizzes WHERE id = 1
quizRepository.findAll();            // SELECT * FROM quizzes
quizRepository.save(quiz);           // INSERT or UPDATE
quizRepository.delete(quiz);         // DELETE
quizRepository.findByTitle("JS");    // SELECT * FROM quizzes WHERE title = 'JS'

// Benefits:
// 1. Type-safe (compiler checks)
// 2. Less code (90% less boilerplate)
// 3. Automatic SQL generation
// 4. Easy to mock for testing
// 5. Consistency across application
```

Behavioral & Problem-Solving Questions

Q16: "Describe a challenging part of this project and how you solved it"

Answer:

Challenge: "How to prevent duplicate answers and ensure data consistency?"

Problem Analysis:
- User could submit the same answer twice
- Could corrupt the scoring logic
- No database constraint to prevent it

Initial Approach (Wrong):
```
// Just check in service
if (responseRepository.findByAttemptAndQuestion(...).isPresent()) {
  throw RuntimeException;
}
```

Problem:
- Race condition: Two requests arrive simultaneously
- Both check: "No response exists" (True)
- Both insert: Response saved twice
- Data inconsistency!

Solution Implemented (Right):

1. Service-level check:
```
@Service
public void submitResponse(Long attemptId, ResponseSubmitRequestDTO request) {
  Optional<Response> existing = responseRepository
    .findByAttemptAndQuestion(attempt, question);

  if (existing.isPresent()) {
    throw new RuntimeException("Response already exists");
  }
  // Save response
}
```

2. Database-level constraint:
```
CREATE UNIQUE INDEX unique_response ON responses(attempt_id, question_id);

// Now even if two requests pass service check,
// database will reject duplicate insert with constraint violation
```

3. Transaction safety:
```
@Transactional(isolation = REPEATABLE_READ)
public ResultDTO submitQuiz(...) {
  // Locks rows during transaction
```

```
  // Prevents race conditions
}
```

Result:
✓ Service layer catches most duplicates (fail fast)
✓ Database constraint catches edge cases (data integrity)
✓ Transaction isolation prevents race conditions
✓ System is robust and fault-tolerant

This is defense-in-depth: Multiple layers of protection.

Q17: "If the database went down, what would break?"

Answer:

What would break immediately:
1. GET /api/quizzes → Can't fetch quiz list → UI shows "Loading..."
2. POST /api/quizzes/{id}/attempt/start → Can't create attempt → Can't start quiz
3. POST /api/quizzes/attempt/{id}/response → Can't save answer → Quiz unusable
4. GET /api/quizzes/attempt/{id}/result → Can't fetch results → UI shows "Loading..."

User Experience:
- App loads (static assets cached)
- Quiz list doesn't load (API error)
- "Start Quiz" button leads to blank page
- User confused, thinks app is broken

How to handle:

1. Error Response from Backend:
```
@ExceptionHandler(DataAccessException.class)
public ResponseEntity<ErrorResponse> handleDbError(DataAccessException ex) {
  return ResponseEntity
    .status(HttpStatus.SERVICE_UNAVAILABLE)
    .body(new ErrorResponse("Database unavailable", "DB_ERROR"));
}
```

2. Frontend Error Handling:
```
axios.get("/api/quizzes")
  .then(r => setQuizzes(r.data))
  .catch(err => {
    if (err.response?.status === 503) {
      setError("Database is temporarily down. Try again later.");
    } else {
      setError("Network error. Check your connection.");
    }
  });
```

3. Graceful Degradation:
```
// Show cached quiz list from localStorage
const cachedQuizzes = localStorage.getItem("quizzes");
if (cachedQuizzes) {
  setQuizzes(JSON.parse(cachedQuizzes));
  setWarning("Using offline data");
}
```

4. Health Check Endpoint:
```
@GetMapping("/health")
```

```
public ResponseEntity<HealthStatus> health() {
  try {
    quizRepository.count();  // Test DB connection
    return ResponseEntity.ok(new HealthStatus("UP"));
  } catch (Exception e) {
    return ResponseEntity
      .status(503)
      .body(new HealthStatus("DOWN"));
  }
}

Frontend periodically calls:
setInterval(() => {
  axios.get("/health").catch(() => {
    setDatabaseDown(true);
  });
}, 10000);
```

Q18: "How would you approach adding a 'timer' feature?"

Answer:

```
javascript
```

```
// Current state: No timer enforcement

// Step 1: Store deadline on frontend
useEffect(() => {
  axios.post(`.../attempt/start`, ..., {params: {userId}})
    .then(r => {
      setAttemptId(r.data.attemptId);
      setDeadline(new Date(r.data.endTime));  // Store end time
    });
}, [quizId]);

// Step 2: Countdown timer
useEffect(() => {
  if (!deadline) return;

  const timer = setInterval(() => {
    const remaining = Math.max(0, deadline - new Date());
    setTimeLeft(Math.floor(remaining / 1000));  // In seconds

    if (remaining <= 0) {
      clearInterval(timer);
      submitQuiz();  // Auto-submit
    }
  }, 1000);

  return () => clearInterval(timer);
}, [deadline]);

// Step 3: Display timer
<div>
  <h2>Time Remaining: {Math.floor(timeLeft / 60)}:{(timeLeft % 60).toString().padStart(2, '0')}</h2>
  {timeLeft < 60 && <div style={{color: 'red'}}>Less than 1 minute!</div>}
</div>

// Step 4: Backend validation
@PostMapping("/attempt/{attemptId}/submit")
public ResponseEntity<ResultDTO> submitQuiz(@PathVariable Long attemptId) {
  UserAttempt attempt = userAttemptRepository.findById(attemptId).orElseThrow();

  // Calculate elapsed time
  long elapsedSeconds = ChronoUnit.SECONDS.between(
    attempt.getStartTime(),
    LocalDateTime.now()
```

```
    );

    long allowedSeconds = attempt.getQuiz().getDuration() * 60;

    // If submitted after time, still mark as submitted
    // But could add flag: wasLate = elapsedSeconds > allowedSeconds

    attempt.setEndTime(LocalDateTime.now());
    attempt.setScore(calculateScore(attempt));
    userAttemptRepository.save(attempt);

    return ResponseEntity.ok(getResult(attemptId));
}

// Step 5: Handle edge cases
// What if clock skew between frontend and backend?
// What if user pauses window?

// Solution: Trust backend time
// Periodically sync with server:
useEffect(() => {
  setInterval(() => {
    axios.get("/api/sync-time")
      .then(r => {
        const serverTime = new Date(r.data.currentTime);
        const clientTime = new Date();
        setTimeDrift(serverTime - clientTime);
      });
  }, 10000);  // Every 10 seconds
}, []);
```

SUMMARY TABLE

Aspect	Technology	Why?
Frontend Framework	React	Component-based, Virtual DOM, easy state management
Frontend Routing	React Router	SPA routing without page reloads
HTTP Client	Axios	Promise-based, interceptors, timeout
Backend Framework	Spring Boot	Auto-config, embedded Tomcat, dependency injection

Aspect	Technology	Why?
Database ORM	JPA/Hibernate	Automatic SQL generation, type-safe queries
Database	H2	In-memory, zero setup, perfect for dev
API Communication	REST/JSON	Stateless, cacheable, standard
Auth	Session/JWT	Identify users, prevent unauthorized access
Data Validation	Service + DB	Defense-in-depth, server validates all inputs
Error Handling	@ExceptionHandler	Centralized error handling, consistent responses

---

INTERVIEW TIPS

1. **Understand the flow end-to-end -** You should be able to trace from UI click to database and back
2. **Know your tech stack -** Be ready to explain why each technology was chosen
3. **Discuss trade-offs -** Every choice has pros and cons, be honest about them
4. **Think about scale -** How would this work with 1M users?
5. **Security mindset -** Always validate inputs, never trust the client
6. **Testing -** Be able to talk about unit tests, integration tests, mocking
7. **Monitoring -** How would you know if something is wrong in production?
8. **Documentation -** Can you explain architecture to a non-technical person?

FOLLOW-UP IMPROVEMENTS TO MENTION

"If I had more time, I would add:"

1. Comprehensive error handling with user-friendly messages
2. Input validation using @Valid annotations
3. Unit tests using JUnit 5 and Mockito
4. Integration tests using @SpringBootTest
5. Timer implementation with server-side validation
6. JWT authentication for proper user identification
7. Proper logging using SLF4J
8. API documentation using Swagger/OpenAPI

9. Performance testing to identify bottlenecks
10. Caching layer for frequently accessed quizzes