D1 Enhanced Framework: Complete Guide

This document demonstrates the complete D1 Enhanced Framework with strategic callouts, verification patterns, and code scaffolding best practices.

🔄 Transforming API Data

The API returns data in its own format, but your game needs a different structure. You'll build the transformation incrementally, testing after each step to see your progress.

Step 1: Set Up Transformation Testing

Before building the transformation, add logging to see the before and after.

- **Open** `src/services/trivia.js`
- **Add** after the validation check in `fetchQuestions`:

```
if (!data.results || data.results.length === 0) {
  console.log("No questions received from API");
  return [];
}

// Add these lines ↓
const firstQuestion = data.results[0];
console.log("Before transform:", firstQuestion);

const transformed = transformQuestion(firstQuestion);
console.log("After transform:", transformed);
```

- **Verify:** Click a zone and check the console. You'll see `undefined` because `transformQuestion` isn't implemented yet.

Step 2: Extract the Raw Properties

Pull out the question and answers from the API response.

- **Find** the `transformQuestion` function
- **Add** property extraction:

```
function transformQuestion(apiQuestion) {
  // Add these lines ↓
  const question = apiQuestion.question;
  const incorrectAnswers = apiQuestion.incorrect_answers;
  const correctAnswer = apiQuestion.correct_answer;

  console.log("Extracted properties:", { question, incorrectAnswers, correctAnsw
}
```

- **Verify:** Click a zone. You should see the raw extracted data in the console.

Step 3: Decode the URL Encoding

Convert the encoded text (`What%20does%20GHz%20stand%20for%3F`) into readable format.

- **Update** `transformQuestion` to use the helper function:

```
function transformQuestion(apiQuestion) {
  // Replace the previous lines with these ↓
  const question = decodeText(apiQuestion.question);
  const incorrectAnswers = apiQuestion.incorrect_answers.map(answer => decodeTex
  const correctAnswer = decodeText(apiQuestion.correct_answer);

  console.log("Decoded data:", { question, incorrectAnswers, correctAnswer });
}
```

- **Expected output:**

```
{
  "question": "What does GHz stand for?",
  "incorrectAnswers": ["Gigahotz", "Gigahetz", "Gigahatz"],
  "correctAnswer": "Gigahertz"
}
```

> 💡 **Concept:** The `map()` method transforms each item in an array. Here, it decodes each incorrect answer.

Step 4: Shuffle Answers and Find the Correct Index

Randomize answer order so players can't memorize positions.

- **Add** the shuffling logic:

```
function transformQuestion(apiQuestion) {
  const question = decodeText(apiQuestion.question);
  const incorrectAnswers = apiQuestion.incorrect_answers.map(answer => decodeTex
  const correctAnswer = decodeText(apiQuestion.correct_answer);

  // Add these lines ↓
  const shuffledAnswers = shuffleAnswers(correctAnswer, incorrectAnswers);
  const correctIndex = shuffledAnswers.indexOf(correctAnswer);

  console.log("Shuffled answers:", shuffledAnswers);
  console.log("Correct answer is at index:", correctIndex);
}
```

- **Verify:** You should see shuffled answers and the index where the correct answer ended up.

> ⚠️ **Warning:** Don't modify the original arrays. The helper functions return new arrays to avoid side effects.

Step 5: Return the Game Object

Build the final format your game needs.

- **Add** the return statement:

```
function transformQuestion(apiQuestion) {
  const question = decodeText(apiQuestion.question);
  const incorrectAnswers = apiQuestion.incorrect_answers.map(answer => decodeTex
  const correctAnswer = decodeText(apiQuestion.correct_answer);
  const shuffledAnswers = shuffleAnswers(correctAnswer, incorrectAnswers);
  const correctIndex = shuffledAnswers.indexOf(correctAnswer);

  // Add this return statement ↓
  return {
    question: question,
    answers: shuffledAnswers,
    correct: correctIndex
  };
}
```

- **Expected output:**

```
{
  "question": "What does CPU stand for?",
  "answers": [
    "Central Process Unit",
    "Computer Personal Unit",
    "Central Processing Unit",
    "Central Processor Unit"
  ],
  "correct": 2
}
```

Step 6: Apply to All Questions

Now use your transformation function to process the entire array.

- **Replace** the test logging in `fetchQuestions`:

```
// Remove these lines:
const firstQuestion = data.results[0];
console.log("Before transform:", firstQuestion);
const transformed = transformQuestion(firstQuestion);
console.log("After transform:", transformed);

// Add these lines instead:
const questions = data.results.map(apiQuestion => transformQuestion(apiQuestion)
console.log("All transformed questions:", questions);
return questions;
```

- **Verify the complete implementation:**
  1. **Click** a zone
  2. **Open** React DevTools (`F12`)
  3. **Navigate** to Components tab
  4. **Find** GameProvider
  5. **Check** `currentQuestions` state
  6. **Confirm** it contains an array of properly formatted questions

> ✅ **Success Check:**
>
> ☐ `questions` array appears in GameProvider state
>
> ☐ Each question has `question`, `answers`, and `correct` properties
>
> ☐ Answers are shuffled (different order each time)
>
> ☐ No console errors

Data transformation is core to web development. APIs rarely return data in your exact format, so you build functions that bridge the gap.

🏆 Adding Score Tracking

Your game needs to track player performance and display it prominently.

Step 1: Add Score State

Create state to track the player's current score.

- **Open** `src/context/GameContext.jsx`
- **Add** score state after the existing state declarations:

```
export function GameProvider({ children }) {
  const [screen, setScreen] = useState(SCREENS.SPLASH);
  const [zoneProgress, setZoneProgress] = useState({});
  const [score, setScore] = useState(0);  // ← Add this line

  // (other state and functions continue below)
}
```

- **Update** the Context value to include `score`:

```
<GameContext.Provider value={{
  // GAME STATE
  screen,
  score,        // ← Add this line
  zoneProgress,

  // ACTIONS
  setScreen,
  setScore,     // ← Add this line
  // (other properties continue below)
```

- **Verify:** Open React DevTools → GameProvider → hooks → `score` should be `0`

> ℹ️ **Note:** State initialized to `0` ensures the game starts with a clean slate.

Step 2: Display Score in HUD

Show the score to players during gameplay.

- **Open** `src/components/HUD.jsx`
- **Add** the `Scoreboard` component at the top of the file:

```
// ========== ADD THIS COMPONENT ==========
function Scoreboard() {
  const { score } = useGame();
  return <div className="score-display">Score: {score}</div>;
}
// ========== END ==========

// Existing CurrentZone function below
function CurrentZone() {
  // ... existing code ...
}
```

- **Update** the HUD return statement:

```
// Replace this:
return <CurrentZone />;

// With this:
return (
  <>
    <Scoreboard />      {/* ← Add this */}
    <CurrentZone />
  </>
);
```

- **Verify:** Navigate to game screen. "Score: 0" should appear in the HUD.

> 🎯 **Pro Tip:** Use React DevTools to change the score value and watch the UI update in real-time.

Step 3: Update Score on Correct Answers

Reward players with points for correct answers.

- **Find** the `recordCorrectAnswer` function in `GameContext.jsx`
- **Add** point reward:

```
const recordCorrectAnswer = () => {
  setCorrectAnswers((prev) => prev + 1);
  setScore((prev) => prev + POINTS_PER_CORRECT);  // ← Add this line
};
```

- **Verify:** Answer questions correctly. Score should increase by `100` points each time.

> 💡 **Concept:** Updater functions like `setScore((prev) => prev + 100)` ensure accurate calculations even when React batches multiple state updates.

🎨 Code Scaffolding Pattern Examples

This section demonstrates different approaches to guiding code modifications.

Simple Addition (Inline Comment)

File: `src/context/GameContext.jsx`

```
const [zoneProgress, setZoneProgress] = useState({});
// Add score state here
```

Function Update (Context Preservation)

File: `src/context/GameContext.jsx`

```
// Find this function and add the score update:
const recordCorrectAnswer = () => {
  setCorrectAnswers((prev) => prev + 1);
  // Add score update here
};
```

New Section (Region Markers)

File: `src/components/HUD.jsx`

```
// ========== ADD THIS COMPONENT ==========
function Scoreboard() {
  const { score } = useGame();
  return <div className="score-display">Score: {score}</div>;
}
// ========== END ==========
```

Replacement (Before/After)

Before:

```
return <CurrentZone />;
```

After:

```
return (
  <>
    <Scoreboard />
    <CurrentZone />
  </>
);
```

Solo Mission (Placeholder Comments)

File: `src/hooks/useAudio.js`

```
const pause = () => {
  // TODO: Check if audio exists
  // TODO: Call pause() method
  // TODO: Update isPlaying state
};
```

JSX Comments (Special Syntax)

File: `src/components/GameBoard.jsx`

```
return (
  <div className="game-board">
    <Scoreboard />
    {/* Add CurrentZone component here */}
    <MusicToggle />
  </div>
);
```

> ⚠️ **Warning:** In JSX, use `{/* comment */}` syntax, not `// comment`. Regular JavaScript comments only work outside JSX or in JavaScript expressions.

📋 Verification Pattern Examples

Inline Verification (Simple)

- **Update** the title:

  ```
  <title>Wizcamp Realms</title>
  ```

  Verify: Browser tab should display the new title

Bullet Verification (Standard)

- **Update** the title:

  ```
  <title>Wizcamp Realms</title>
  ```

- **Verify:** Check the browser tab. It should display "Wizcamp Realms"

Dedicated Verification (Complex)

- **Update** the scoring system
- **Verify the implementation:**
  1. **Click** the **Start Game** button
  2. Answer a question correctly
  3. Check that `score` increases by `10`
  4. Answer incorrectly
  5. Verify `score` doesn't go below `0`

Expected Output (API/Data)

- **Fetch** questions from the API
- **Expected output:**

```
{
  "response_code": 0,
  "results": [...]
}
```

Success Criteria (Checklist)

- **Complete** the scoring system

> ✅ **Success Check:**
>
> ☐ Score displays “0” when game starts
>
> ☐ Score increases by 10 for correct answers
>
> ☐ Score resets to 0 when “Play Again” is clicked

🎓 Callout Type Reference

Concept Callout

> 💡 **Concept:** Use this for explaining how or why something works.

Warning Callout

> ⚠️ **Warning:** Use this to prevent common mistakes or highlight important gotchas.

Success Check Callout

> ✅ **Success Check:** Use this for verification checklists with multiple conditions.

Note Callout

> ℹ️ **Note:** Use this for additional context, tips, or side information.

Pro Tip Callout

> 🎯 **Pro Tip:** Use this for advanced techniques or shortcuts.

Key Framework Principles

1. **Step X: Descriptive Name** provides wayfinding + meaning
2. **Bullets** separate actions from context
3. **Callouts** teach concepts without blocking action
4. **Code scaffolding** matches complexity (simple → detailed)
5. **Verification** matches task complexity (inline → checklist)
6. **File paths** are mandatory for every code block
7. **Consistent patterns** reduce cognitive load

Formatting Best Practices

When to Use Bold

- **Action verbs** at the start of bullet points: **Open**, **Add**, **Update**, **Find**, **Replace**, **Verify**, **Create**, **Delete**, **Install**, **Run**, **Navigate**, **Click**, **Check**, **Confirm**
- **File paths** when mentioned inline: Open `src/components/App.jsx`
- **Important UI elements**: Click the **Start Game** button
- **Key terms** on first mention in a section: The **Context API** provides shared state

When to Use Backticks

- **Code elements**: function names (`useState`), variables (`score`), properties (`value`), constants (`SCREENS.SPLASH`)
- **File paths**: `src/context/GameContext.jsx`
- **Keyboard shortcuts**: Press `Ctrl+C` or `Cmd+C`
- **Terminal commands**: Run `npm install`

- **HTML/CSS selectors**: `.score-display` or `<div>`
- **Package names**: Install `react-router-dom`

When to Use Both Bold + Backticks

- **File paths in action bullets**: **Open** `src/components/App.jsx`
- **Important code elements in instructions**: **Find** the `recordCorrectAnswer` function
- **UI elements that are also code**: Click the `<StartButton />` component

When to Use Neither

- Regular descriptive text
- Explanations and context
- UI navigation paths in verification (React DevTools → Components → GameProvider)
- Callout body text

Consistent Patterns

Action Bullets:

- **Open** `src/context/GameContext.jsx`
- **Add** score state after the existing declarations
- **Find** the `recordCorrectAnswer` function
- **Update** the Context value to include `score`

Inline File References:

The `GameContext.jsx` file manages shared state. Open `src/components/HUD.jsx` to add the scoreboard.

Code Element References:

The `useState` hook creates state. Pass the `score` prop to the component. Update the `value` property in the object.

Verification Text:

- **Verify:** Open React DevTools → GameProvider → hooks → `score` should be `0`
- **Verify:** Navigate to game screen. "Score: 0" should appear in the HUD.
- **Verify:** Check the `currentQuestions` state in GameProvider

Common Mistakes to Avoid

- ❌ Don't bold entire sentences
- ❌ Don't use backticks for non-code terms ("the `game` needs scoring")
- ❌ Don't mix styles inconsistently (Open **src/file.js** vs **Open** `src/file.js`)
- ❌ Don't bold verification text except the word "Verify"
- ❌ Don't use backticks around quoted UI text ("Score: 0" not `"Score: 0"`)

Quick Reference

Element	Format	Example
Action verb	**Bold**	**Open** the file
File path in action	**Bold** + `backticks`	**Open** `src/App.jsx`
File path inline	`backticks`	The `App.jsx` file contains...
Function/variable	`backticks`	The `useState` hook
UI element	**Bold**	Click the **Start** button
Code + UI element	**Bold** + `backticks`	The `<Button />` component
Keyboard shortcut	`backticks`	Press `F12`
Terminal command	`backticks`	Run `npm start`
Property/constant	`backticks`	Set `score` to `0`
Quoted text	"Quotes"	Display "Score: 0"