Criteria C: Development

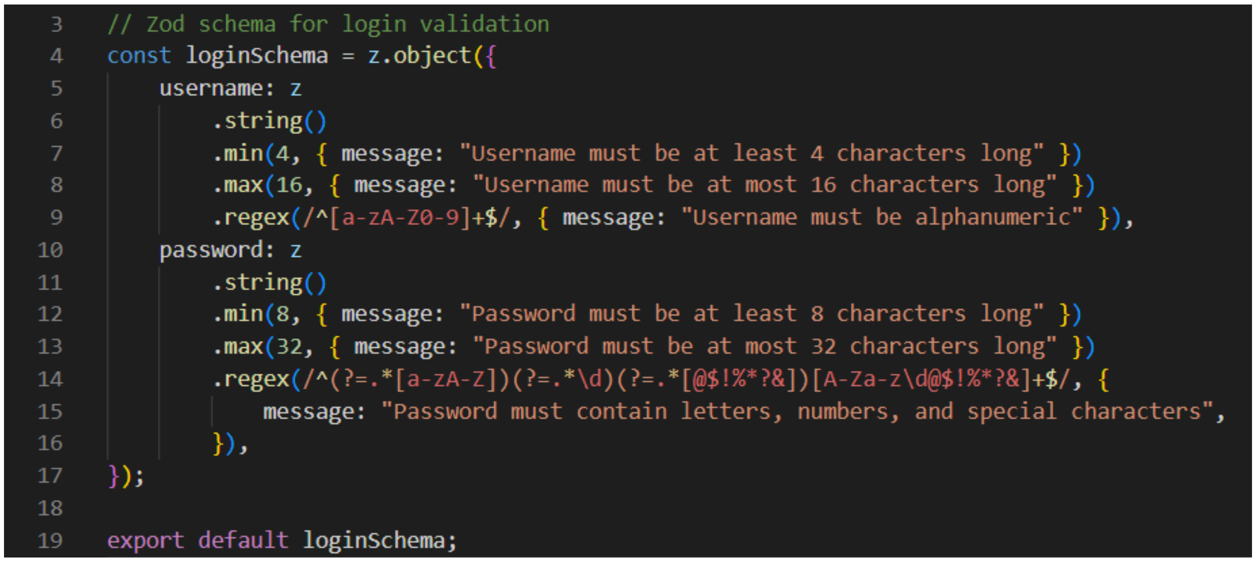
List of Techniques Used

- User Authentication
- Input Validation
- Session Management
- Error Handling & User Feedback
- Database Storage for users, sessions, and user data
- Text Generation
- Real-time Performance Analytics
- Role-based Access Control
- Data Visualization

Implementation of Authentication and Validation

User authentication is a critical part of the system, allowing for students and teachers to register accounts with secure logins before accessing the site. This ensures that both the user's account and data are secure, and protects against nefarious actors.

User Input Validation

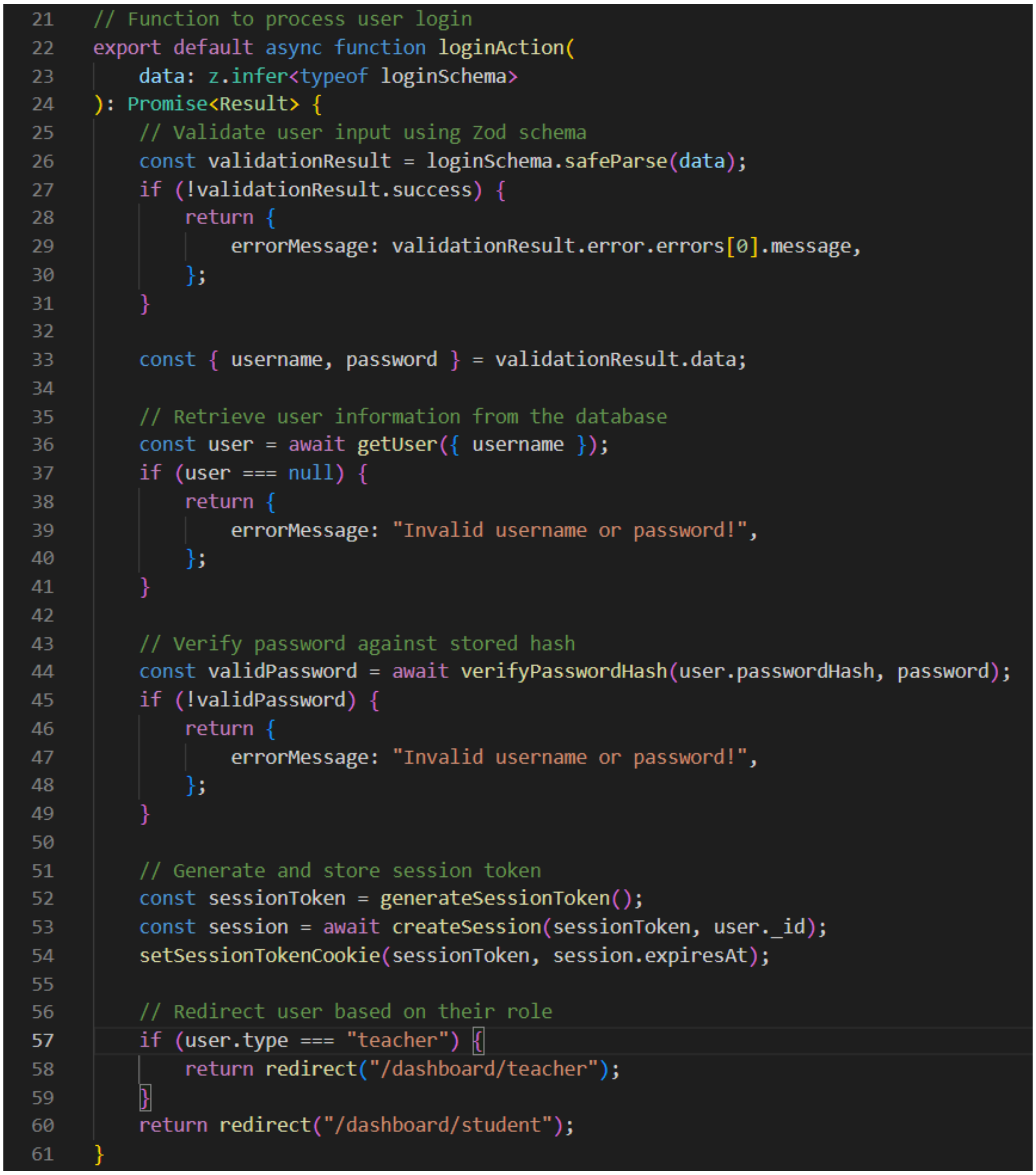


```
// Zod schema for login validation
const loginSchema = z.object({
    username: z
        .string()
        .min(4, { message: "Username must be at least 4 characters long" })
        .max(16, { message: "Username must be at most 16 characters long" })
        .regex(/^[a-zA-Z0-9]+$/, { message: "Username must be alphanumeric" }),
    password: z
        .string()
        .min(8, { message: "Password must be at least 8 characters long" })
        .max(32, { message: "Password must be at most 32 characters long" })
        .regex(/^(?=.*[a-zA-Z])(?=.*\d)(?=.*[@$!%*?&])[A-Za-z\d@$!%*?&]+$/, {
            message: "Password must contain letters, numbers, and special characters",
        }),
});

export default loginSchema;
```

Explanation

- The username must contain between 4 and 16 alphanumeric characters.
- The password must contain between 8 and 32 characters with at least one number and one special character.
- If the input does not meet these requirements, an error message is displayed.
- Guides the user through creating appropriate usernames and secure passwords.
- This ensures that system resources are not spent on attempting to validate a username or password that does not meet the requirements.

Handling User Login

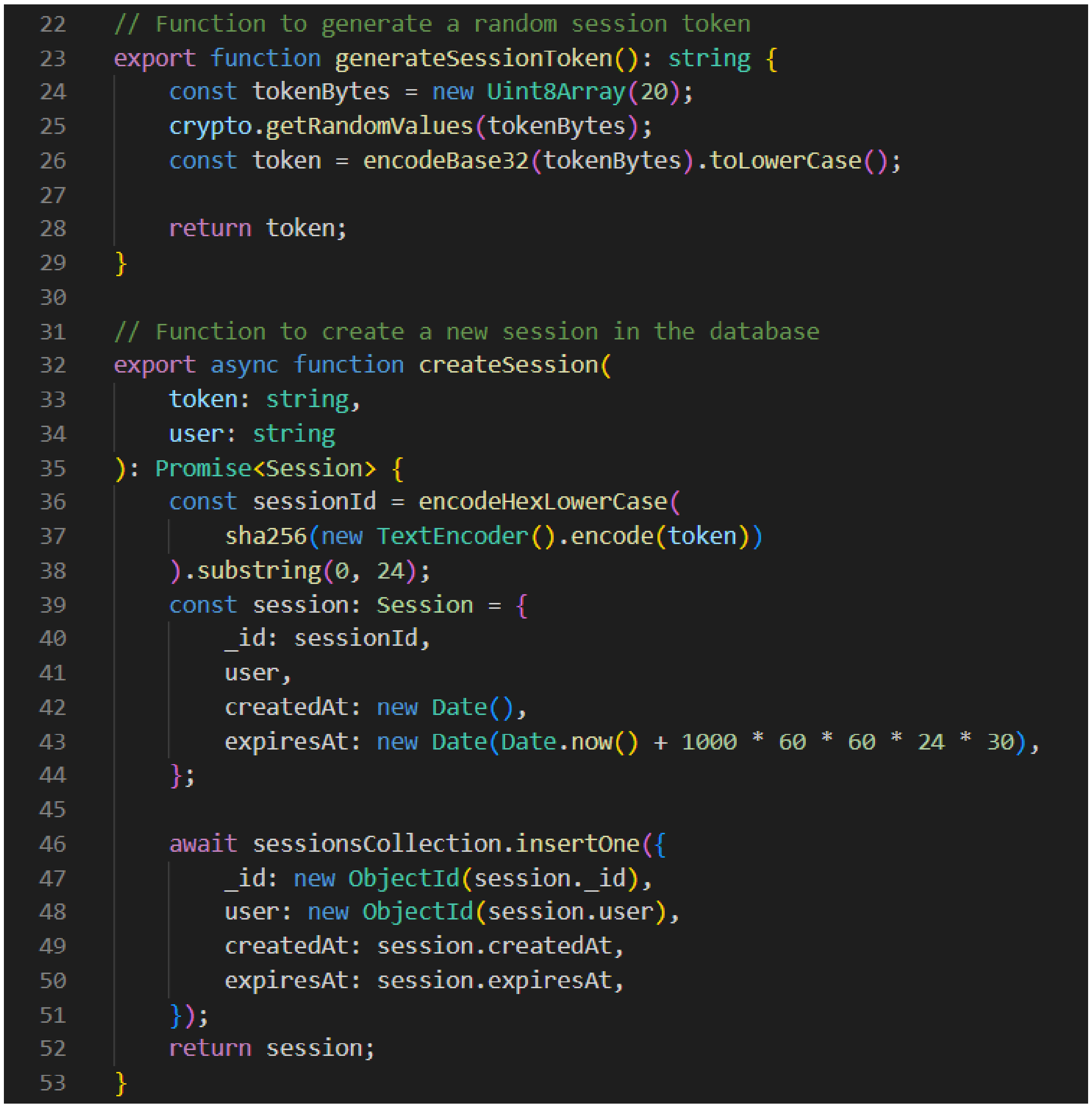


```
// Function to process user login
export default async function loginAction(
    data: z.infer<typeof loginSchema>
): Promise<Result> {
    // Validate user input using Zod schema
    const validationResult = loginSchema.safeParse(data);
    if (!validationResult.success) {
        return {
            errorMessage: validationResult.error.errors[0].message,
        };
    }

    const { username, password } = validationResult.data;

    // Retrieve user information from the database
    const user = await getUser({ username });
    if (user === null) {
        return {
            errorMessage: "Invalid username or password!",
        };
    }

    // Verify password against stored hash
    const validPassword = await verifyPasswordHash(user.passwordHash, password);
    if (!validPassword) {
        return {
            errorMessage: "Invalid username or password!",
        };
    }

    // Generate and store session token
    const sessionToken = generateSessionToken();
    const session = await createSession(sessionToken, user._id);
    setSessionTokenCookie(sessionToken, session.expiresAt);

    // Redirect user based on their role
    if (user.type === "teacher") {
        return redirect("/dashboard/teacher");
    }
    return redirect("/dashboard/student");
}
```

Explanation

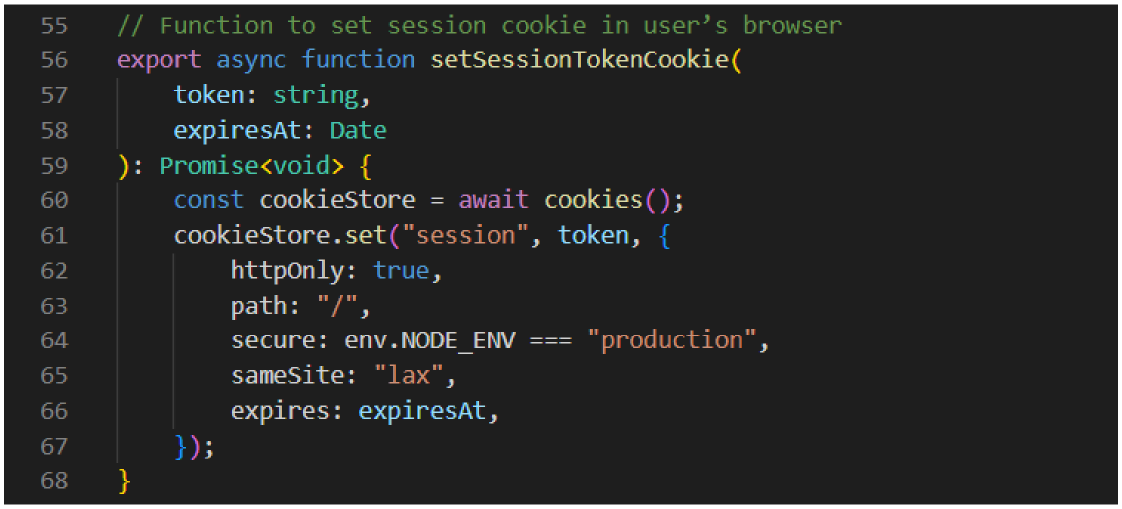
- The function first validates the user's input using the schema outlined earlier.
- If the input is valid, it then retrieves the user's details from the database.
- It then verifies whether the password matches the password hash (encrypted password) stored in the database.
- If successful, it creates a session, sets a session cookie, and finally redirects the user to their dashboard depending on their role.

Session Management


```
// Function to generate a random session token
export function generateSessionToken(): string {
    const tokenBytes = new Uint8Array(20);
    crypto.getRandomValues(tokenBytes);
    const token = encodeBase32(tokenBytes).toLowerCase();

    return token;
}

// Function to create a new session in the database
export async function createSession(
    token: string,
    user: string
): Promise<Session> {
    const sessionId = encodeHexLowerCase(
        sha256(new TextEncoder().encode(token))
    ).substring(0, 24);
    const session: Session = {
        _id: sessionId,
        user,
        createdAt: new Date(),
        expiresAt: new Date(Date.now() + 1000 * 60 * 60 * 24 * 30),
    };

    await sessionsCollection.insertOne({
        _id: new ObjectId(session._id),
        user: new ObjectId(session.user),
        createdAt: session.createdAt,
        expiresAt: session.expiresAt,
    });
    return session;
}
```

Explanation

- The *generateSessionToken* function creates a unique base32-encoded token for session identification. This token acts as a unique key that allows the system to identify the user securely.
- The *createSession* function takes the random session token and processes it into a unique identifier that can be stored and retrieved later. This is accomplished by generating a unique session ID by converting the token into a secure string using cryptography.
- The session is set to expire after 30 days, ensuring long-term user access while preventing stale sessions. This is important so that users don't have to log-in every time they use the site. The expiration of sessions is also important so the database isn't full of unused session data, and to provide more long-term security.



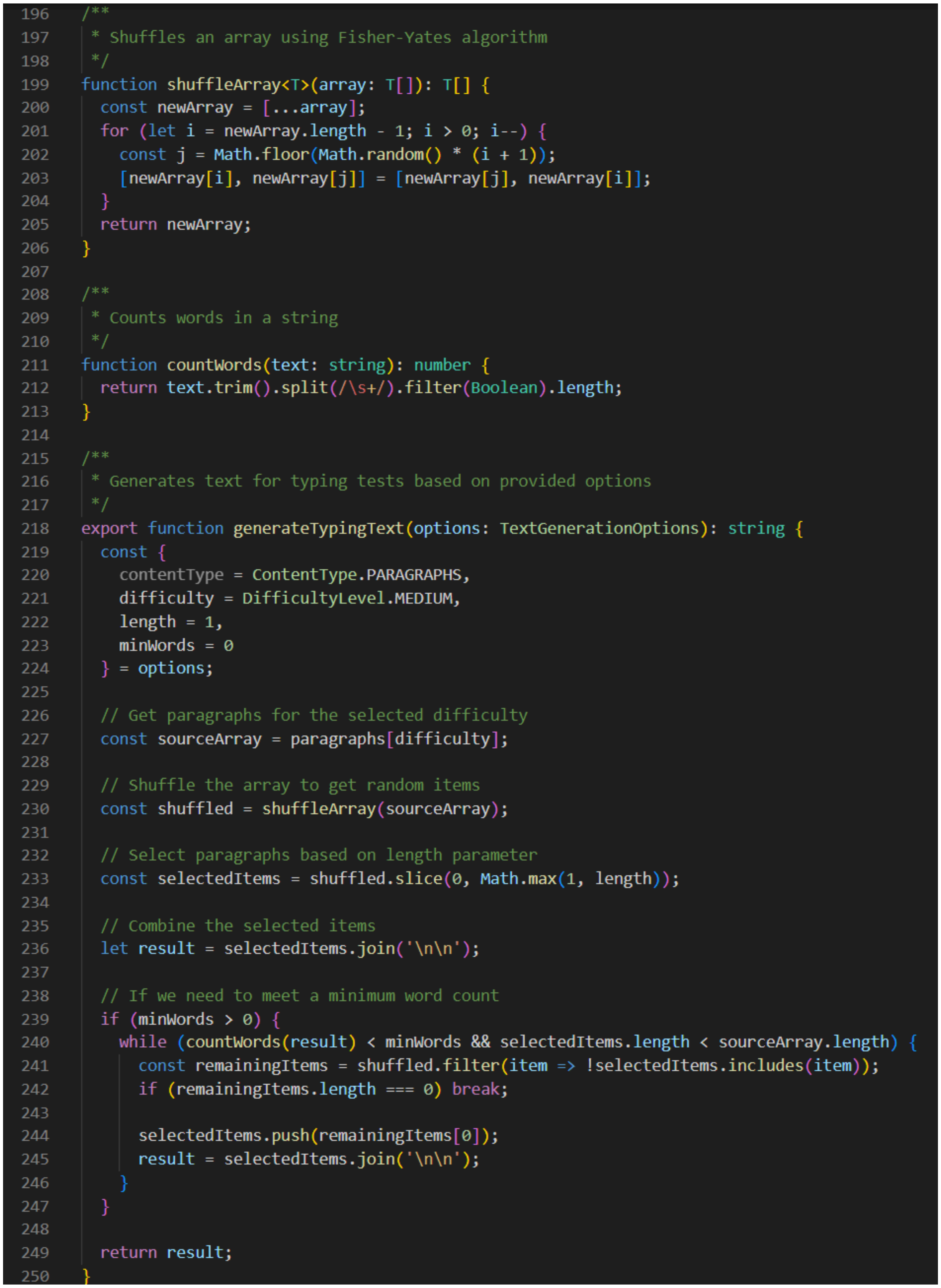
```
// Function to set session cookie in user’s browser
export async function setSessionTokenCookie(
    token: string,
    expiresAt: Date
): Promise<void> {
    const cookieStore = await cookies();
    cookieStore.set("session", token, {
        httpOnly: true,
        path: "/",
        secure: env.NODE_ENV === "production",
        sameSite: "lax",
        expires: expiresAt,
    });
}
```

- The *setSessionTokenCookie* function securely stores a session token in the user’s browser, allowing the site to recognize returning users without requiring them to log in again.

Implementation of Typing Challenges

Typing challenges are the core to this application, with three different modes, Speed Test, Sprint, and Endless. These challenges use different text generation methods tailored to specific objectives, with real-time input handling in order to track user performance.

Text Generation for Speed Test and Sprint

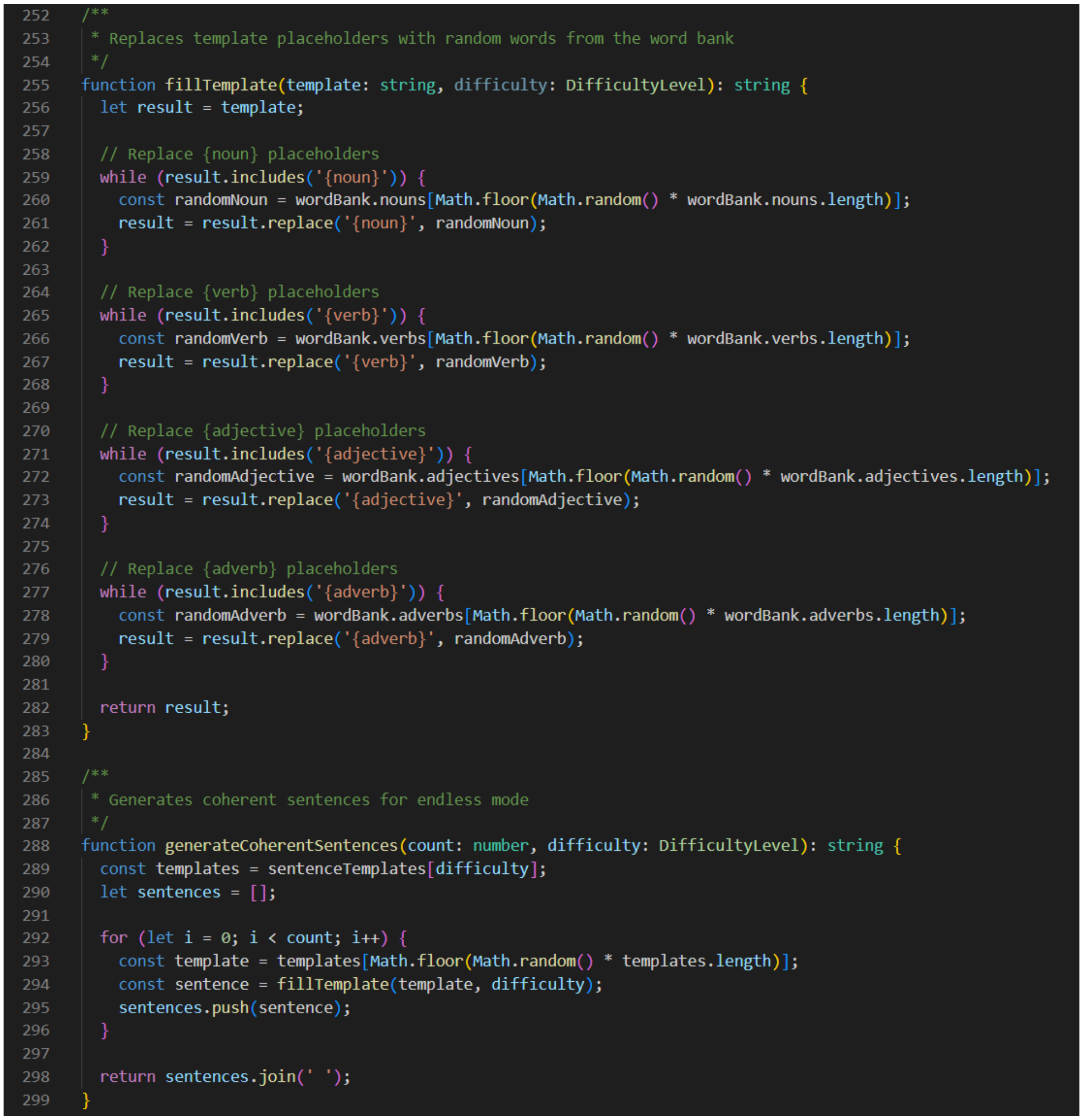


```typescript
/**
 * Shuffles an array using Fisher-Yates algorithm
 */
function shuffleArray<T>(array: T[]): T[] {
  const newArray = [...array];
  for (let i = newArray.length - 1; i > 0; i--) {
    const j = Math.floor(Math.random() * (i + 1));
    [newArray[i], newArray[j]] = [newArray[j], newArray[i]];
  }
  return newArray;
}

/**
 * Counts words in a string
 */
function countWords(text: string): number {
  return text.trim().split(/\s+/).filter(Boolean).length;
}

/**
 * Generates text for typing tests based on provided options
 */
export function generateTypingText(options: TextGenerationOptions): string {
  const {
    contentType = ContentType.PARAGRAPHS,
    difficulty = DifficultyLevel.MEDIUM,
    length = 1,
    minWords = 0
  } = options;

  // Get paragraphs for the selected difficulty
  const sourceArray = paragraphs[difficulty];

  // Shuffle the array to get random items
  const shuffled = shuffleArray(sourceArray);

  // Select paragraphs based on length parameter
  const selectedItems = shuffled.slice(0, Math.max(1, length));

  // Combine the selected items
  let result = selectedItems.join('\n\n');

  // If we need to meet a minimum word count
  if (minWords > 0) {
    while (countWords(result) < minWords && selectedItems.length < sourceArray.length) {
      const remainingItems = shuffled.filter(item => !selectedItems.includes(item));
      if (remainingItems.length === 0) break;

      selectedItems.push(remainingItems[0]);
      result = selectedItems.join('\n\n');
    }
  }

  return result;
}
```

Explanation

- Includes a function to randomly shuffle the array of paragraphs, and a function to count words in a string.
- The generateTypingText function creates randomized typing test text based on the configurable options.
- The function retrieves a set of paragraphs based on difficulty level and shuffles them to ensure variability in generated text.
- If the minimum word count is not met, additional text is appended until the word count condition is met.

Text Generation for Endless

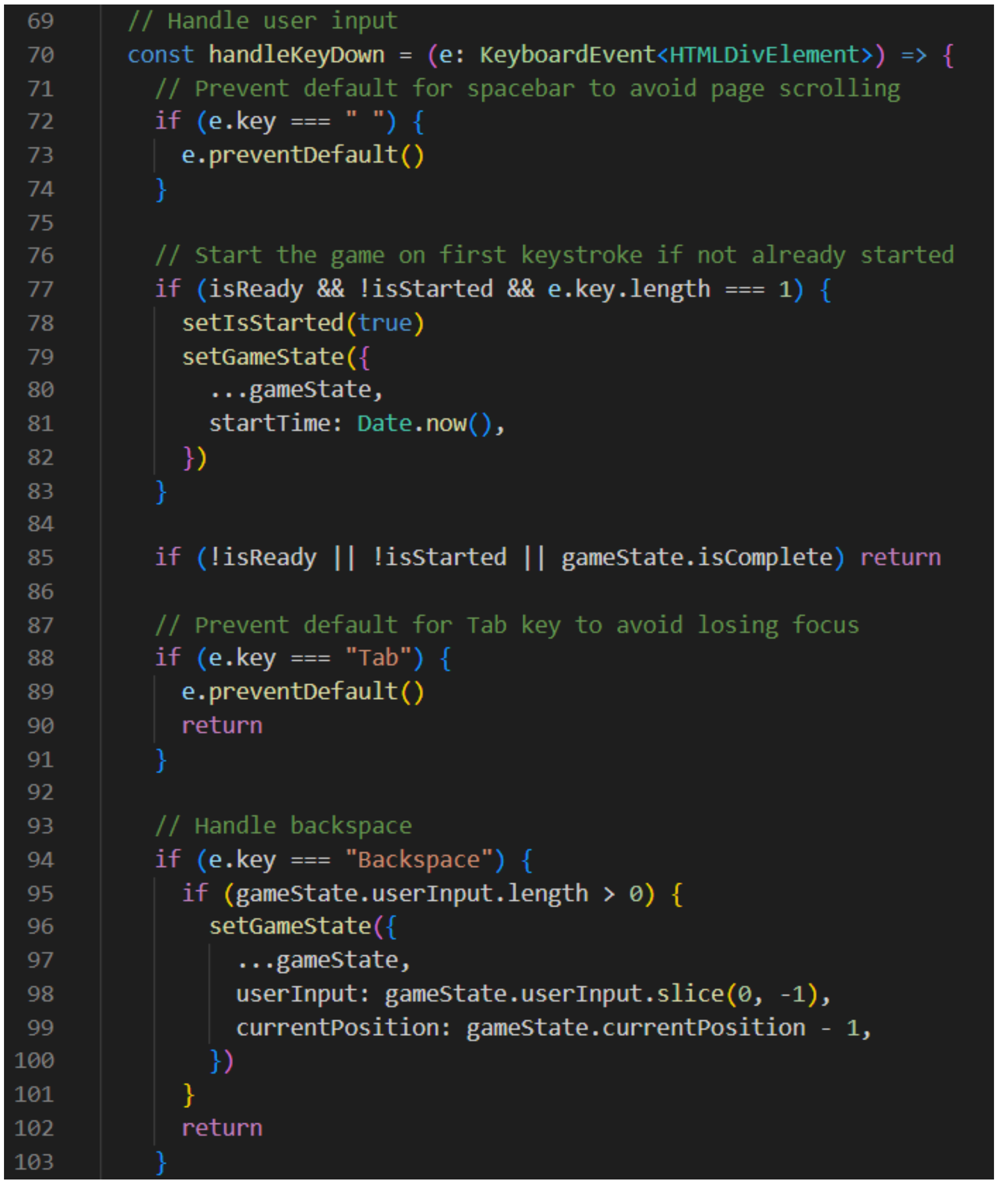


```typescript
/**
 * Replaces template placeholders with random words from the word bank
 */
function fillTemplate(template: string, difficulty: DifficultyLevel): string {
  let result = template;

  // Replace {noun} placeholders
  while (result.includes('{noun}')) {
    const randomNoun = wordBank.nouns[Math.floor(Math.random() * wordBank.nouns.length)];
    result = result.replace('{noun}', randomNoun);
  }

  // Replace {verb} placeholders
  while (result.includes('{verb}')) {
    const randomVerb = wordBank.verbs[Math.floor(Math.random() * wordBank.verbs.length)];
    result = result.replace('{verb}', randomVerb);
  }

  // Replace {adjective} placeholders
  while (result.includes('{adjective}')) {
    const randomAdjective = wordBank.adjectives[Math.floor(Math.random() * wordBank.adjectives.length)];
    result = result.replace('{adjective}', randomAdjective);
  }

  // Replace {adverb} placeholders
  while (result.includes('{adverb}')) {
    const randomAdverb = wordBank.adverbs[Math.floor(Math.random() * wordBank.adverbs.length)];
    result = result.replace('{adverb}', randomAdverb);
  }

  return result;
}

/**
 * Generates coherent sentences for endless mode
 */
function generateCoherentSentences(count: number, difficulty: DifficultyLevel): string {
  const templates = sentenceTemplates[difficulty];
  let sentences = [];

  for (let i = 0; i < count; i++) {
    const template = templates[Math.floor(Math.random() * templates.length)];
    const sentence = fillTemplate(template, difficulty);
    sentences.push(sentence);
  }

  return sentences.join(' ');
}
```

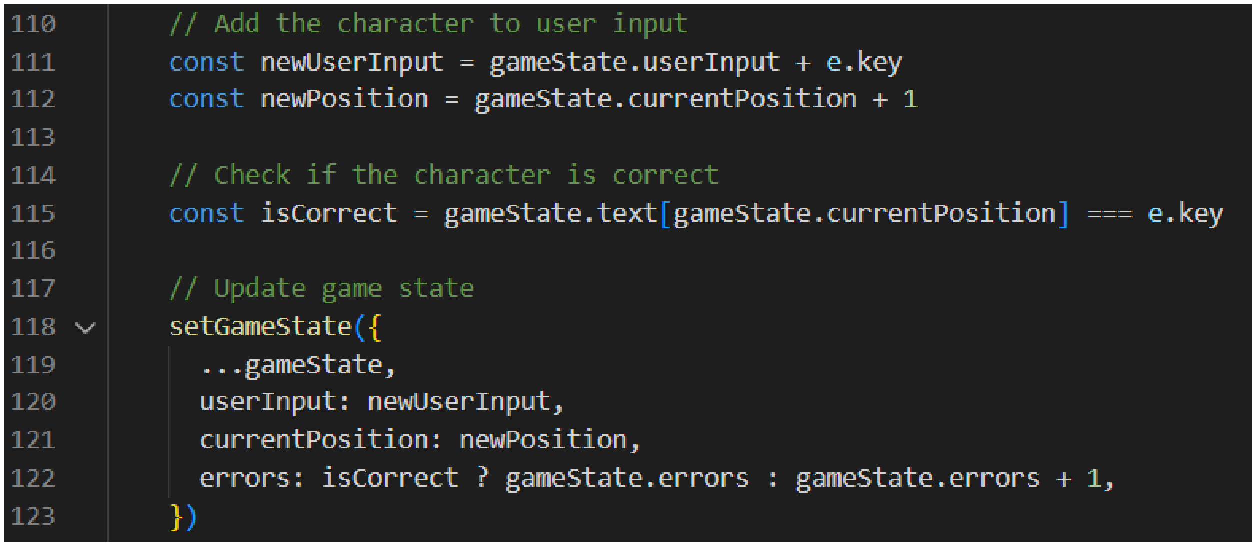
Explanation

- The fillTemplate function replaces placeholders ({noun}, {verb}, {adjective}, {adverb}) in a template string with random words from the wordBank array.
- The function iterates through the template, replacing each placeholder with a randomly selected word from the corresponding category. This ensures variety and unpredictability in sentences.
- The generateCoherentSentences function constructs multiple sentences for Endless Mode by using the sentence templates.
- The function selects a random template based on difficulty level and creates a sentence using the fillTemplate function, forming a full unique sentence.

Input Handling



```
// Handle user input
const handleKeyDown = (e: KeyboardEvent<HTMLDivElement>) => {
  // Prevent default for spacebar to avoid page scrolling
  if (e.key === " ") {
    e.preventDefault()
  }

  // Start the game on first keystroke if not already started
  if (isReady && !isStarted && e.key.length === 1) {
    setIsStarted(true)
    setGameState({
      ...gameState,
      startTime: Date.now(),
    })
  }

  if (!isReady || !isStarted || gameState.isComplete) return

  // Prevent default for Tab key to avoid losing focus
  if (e.key === "Tab") {
    e.preventDefault()
    return
  }

  // Handle backspace
  if (e.key === "Backspace") {
    if (gameState.userInput.length > 0) {
      setGameState({
        ...gameState,
        userInput: gameState.userInput.slice(0, -1),
        currentPosition: gameState.currentPosition - 1,
      })
    }
    return
  }
```



```
    // Add the character to user input
    const newUserInput = gameState.userInput + e.key
    const newPosition = gameState.currentPosition + 1

    // Check if the character is correct
    const isCorrect = gameState.text[gameState.currentPosition] === e.key

    // Update game state
    setGameState({
      ...gameState,
      userInput: newUserInput,
      currentPosition: newPosition,
      errors: isCorrect ? gameState.errors : gameState.errors + 1,
    })
```

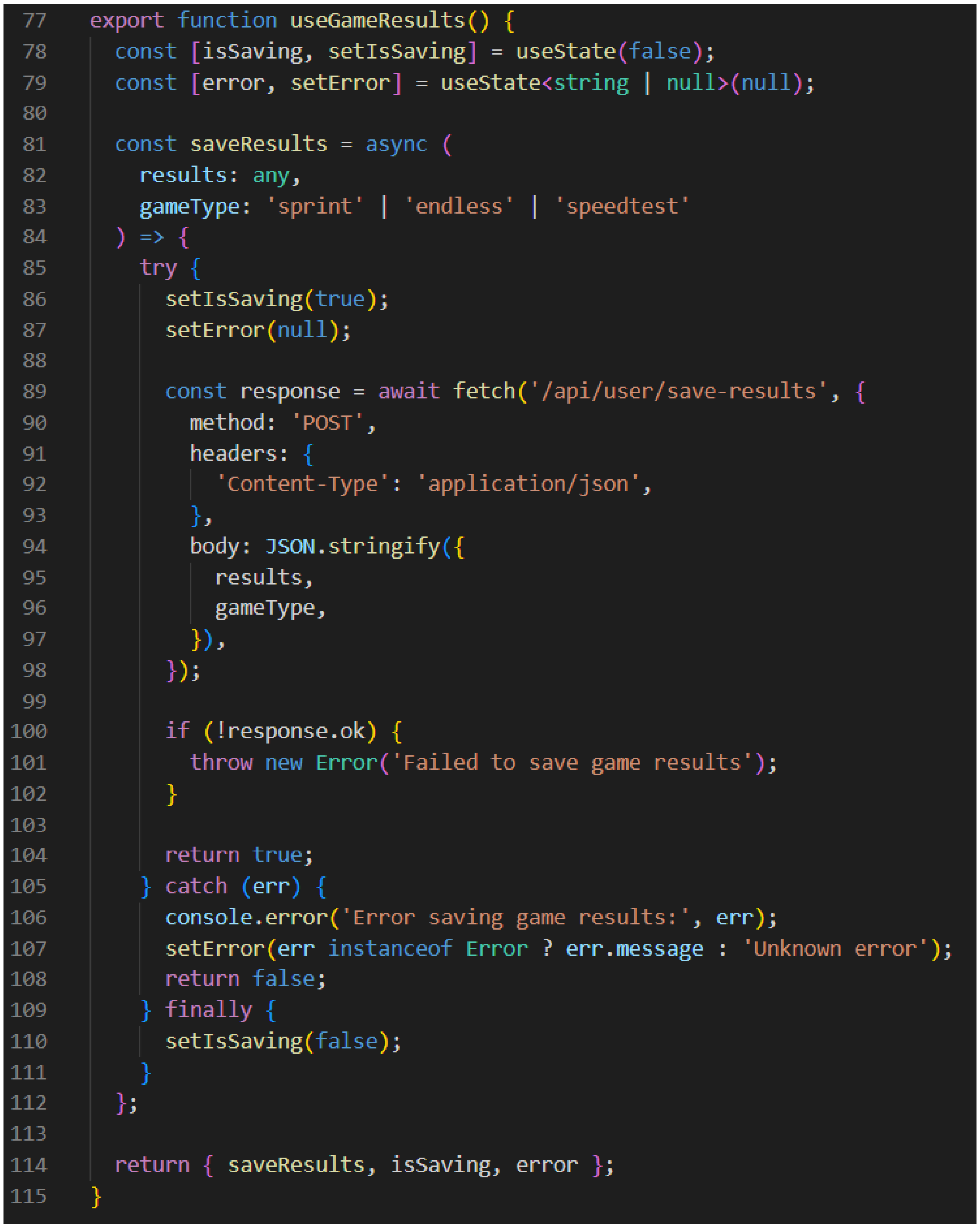
Explanation

- The handleKeyDown function manages user input during the typing test, and prevents the spacebar and tab key from triggering default behavior (scrolling and losing focus). It also starts the game on the first keystroke, and handles backspace functionality by removing the last character from the user input and updating the text position.
- When processing new keystrokes, the game state is updated by appending typed characters to userInput; checking if the character is correct and updating the error count; and updating currentPosition to track typing progress.

Implementation of Statistics

Tracking user performance is essential for measuring progress and providing feedback. The system records statistics such as typing speed, accuracy, and error count in real time. These metrics allow users to analyze their performance, identify areas for improvement, and track their progress over time.

Game Results Saving



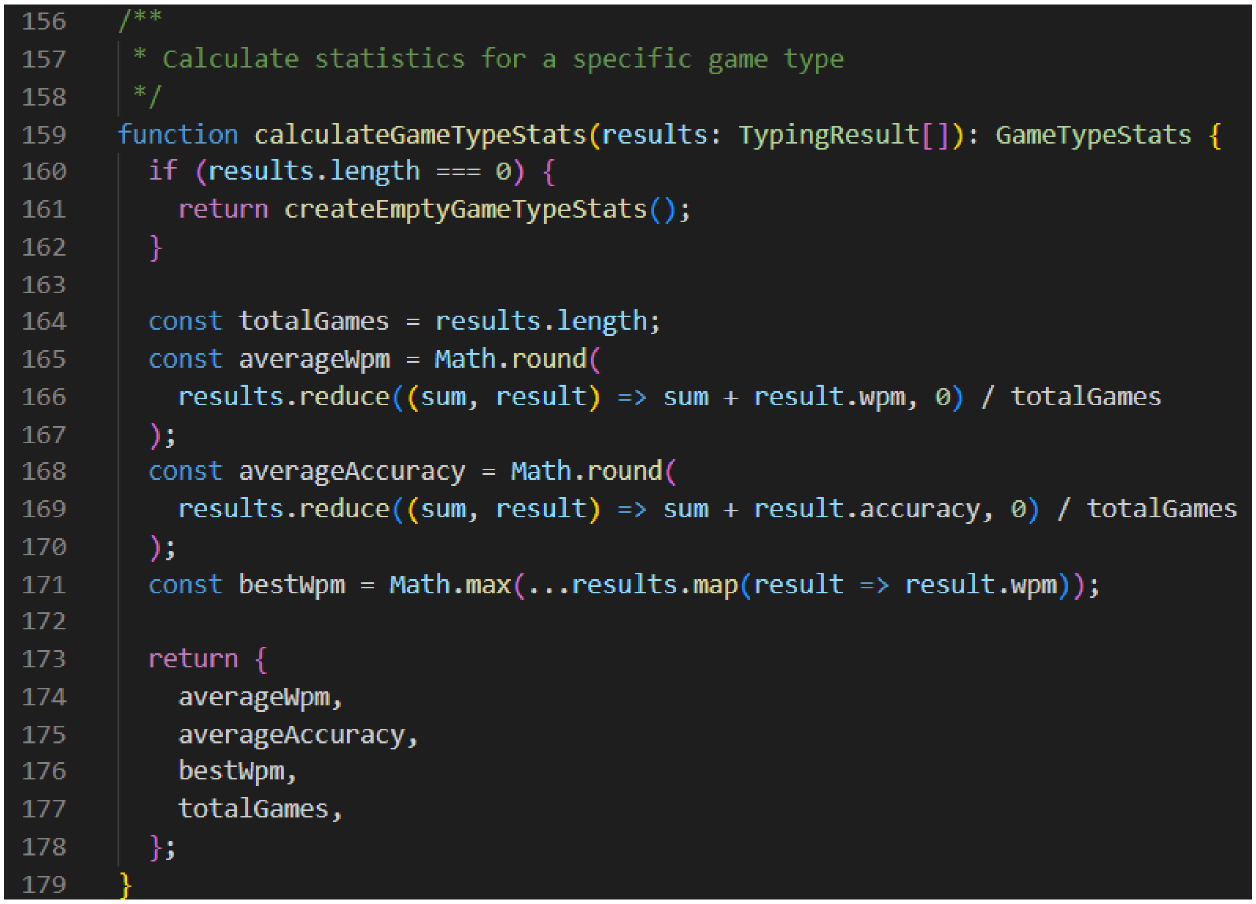
```
export function useGameResults() {
  const [isSaving, setIsSaving] = useState(false);
  const [error, setError] = useState<string | null>(null);

  const saveResults = async (
    results: any,
    gameType: 'sprint' | 'endless' | 'speedtest'
  ) => {
    try {
      setIsSaving(true);
      setError(null);

      const response = await fetch('/api/user/save-results', {
        method: 'POST',
        headers: {
          'Content-Type': 'application/json',
        },
        body: JSON.stringify({
          results,
          gameType,
        }),
      });

      if (!response.ok) {
        throw new Error('Failed to save game results');
      }

      return true;
    } catch (err) {
      console.error('Error saving game results:', err);
      setError(err instanceof Error ? err.message : 'Unknown error');
      return false;
    } finally {
      setIsSaving(false);
    }
  };

  return { saveResults, isSaving, error };
}
```

Explanation

- The useGameResults function provides a React hook for saving game results to the database. It uses useState to manage the loading state (isSaving) and error handling (error).
- The saveResults function sends a POST request to the /api/user/save-results endpoint. This POST includes game results and game type in the request body.

- Uses try-catch-finally to ensure the loading state is updated while the request is in progress, errors are logged and displayed if saving fails, and that the saving state is reset after completion.

User Statistics Calculation



```
/**
 * Calculate statistics for a specific game type
 */
function calculateGameTypeStats(results: TypingResult[]): GameTypeStats {
  if (results.length === 0) {
    return createEmptyGameTypeStats();
  }

  const totalGames = results.length;
  const averageWpm = Math.round(
    results.reduce((sum, result) => sum + result.wpm, 0) / totalGames
  );
  const averageAccuracy = Math.round(
    results.reduce((sum, result) => sum + result.accuracy, 0) / totalGames
  );
  const bestWpm = Math.max(...results.map(result => result.wpm));

  return {
    averageWpm,
    averageAccuracy,
    bestWpm,
    totalGames,
  };
}
```

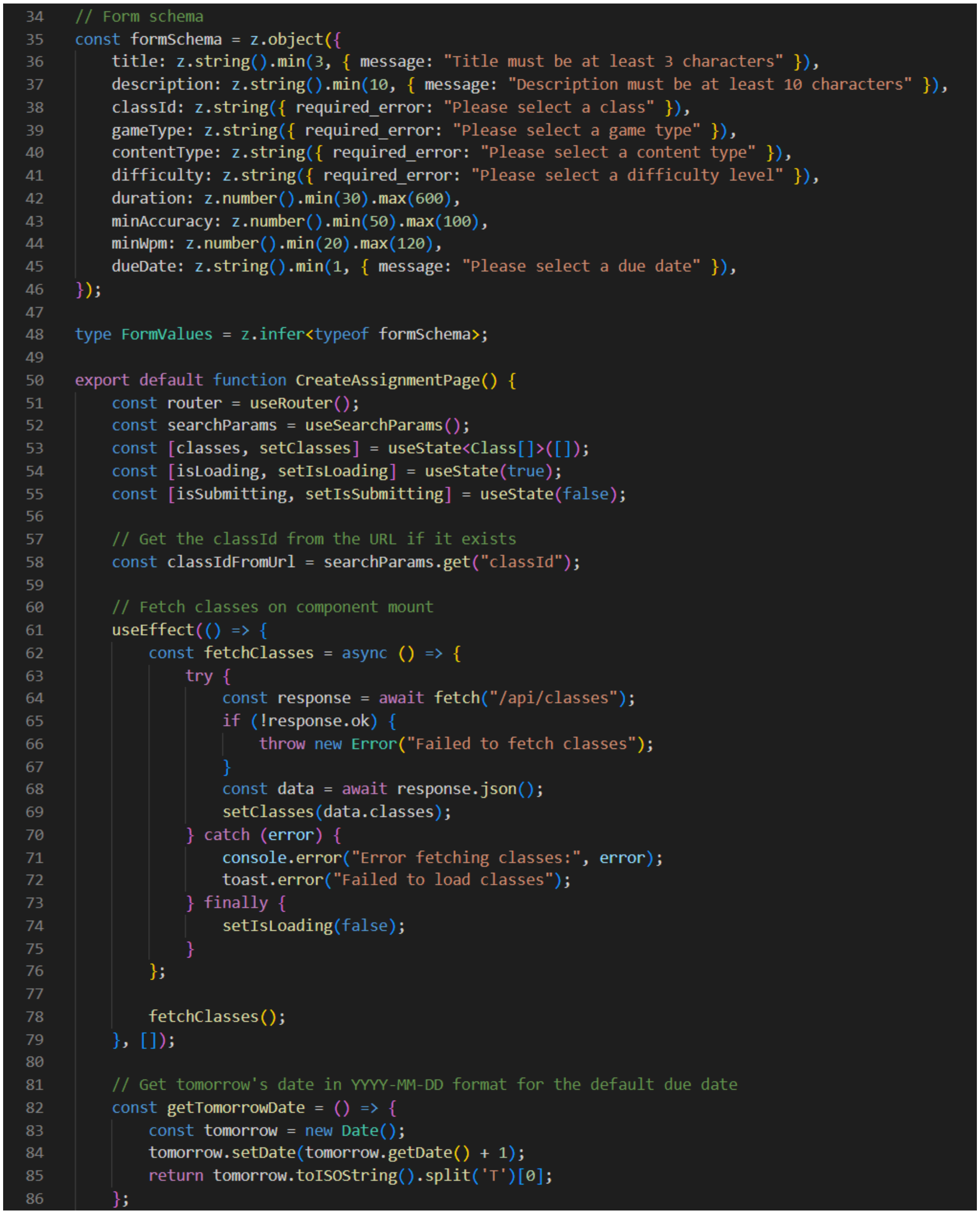
Explanation

- The calculateGameTypeStats function analyses typing results for a specific game type.
- It calculates the statistics: total games played, average words per minute (WPM), average accuracy, and the user's highest WPM recorded.

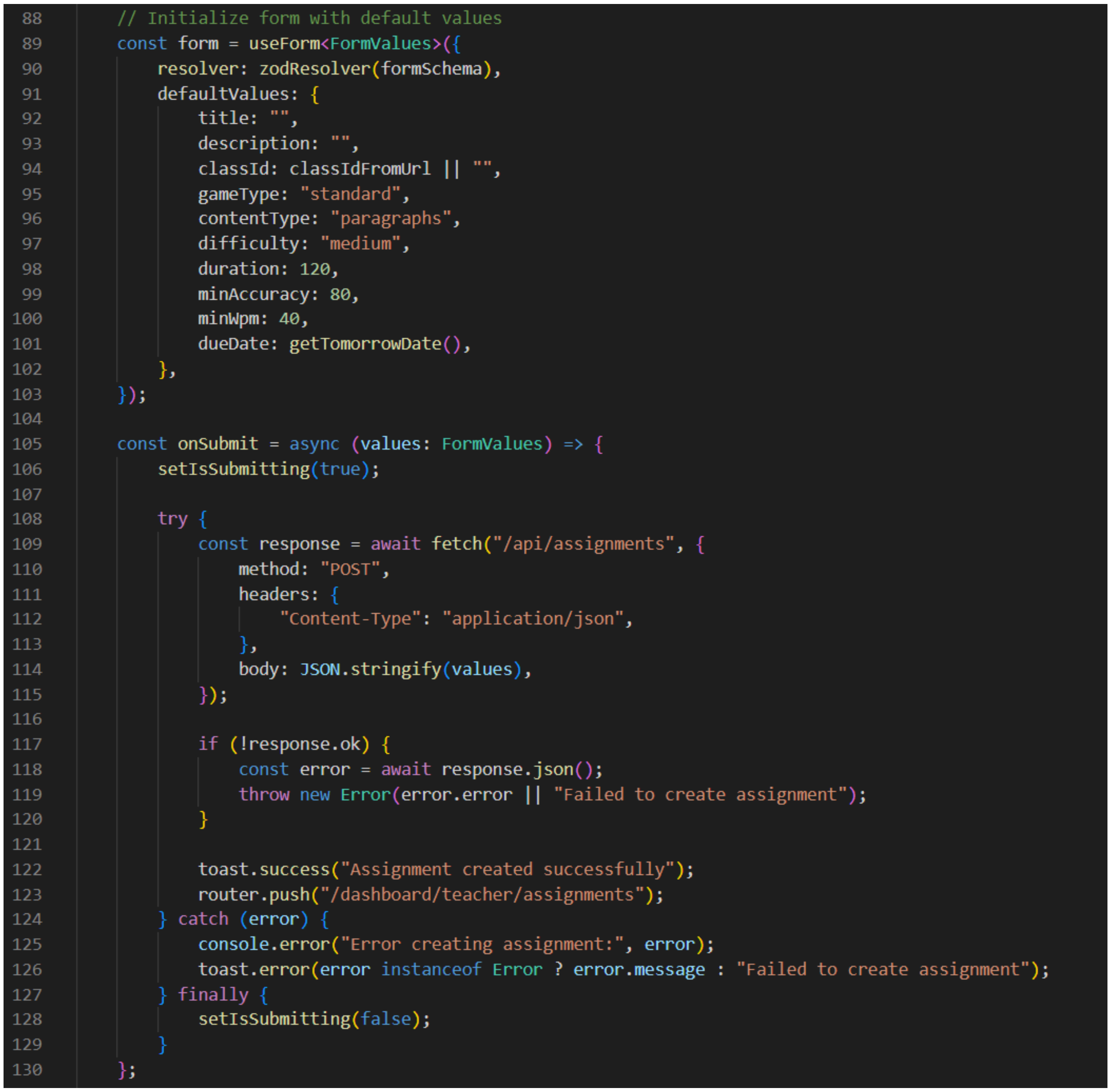
Implementation of Assignments

Assignments provide a structured way for teachers to create and manage typing tasks for their students. Teachers can create assignments for their students with specific parameters and requirements. These assignments are then placed in each student's to-do list for them to complete.

Teacher Assignment Creation



```
// Form schema
const formSchema = z.object({
    title: z.string().min(3, { message: "Title must be at least 3 characters" }),
    description: z.string().min(10, { message: "Description must be at least 10 characters" }),
    classId: z.string({ required_error: "Please select a class" }),
    gameType: z.string({ required_error: "Please select a game type" }),
    contentType: z.string({ required_error: "Please select a content type" }),
    difficulty: z.string({ required_error: "Please select a difficulty level" }),
    duration: z.number().min(30).max(600),
    minAccuracy: z.number().min(50).max(100),
    minWpm: z.number().min(20).max(120),
    dueDate: z.string().min(1, { message: "Please select a due date" }),
});

type FormValues = z.infer<typeof formSchema>;

export default function CreateAssignmentPage() {
    const router = useRouter();
    const searchParams = useSearchParams();
    const [classes, setClasses] = useState<Class[]>([]);
    const [isLoading, setIsLoading] = useState(true);
    const [isSubmitting, setIsSubmitting] = useState(false);

    // Get the classId from the URL if it exists
    const classIdFromUrl = searchParams.get("classId");

    // Fetch classes on component mount
    useEffect(() => {
        const fetchClasses = async () => {
            try {
                const response = await fetch("/api/classes");
                if (!response.ok) {
                    throw new Error("Failed to fetch classes");
                }
                const data = await response.json();
                setClasses(data.classes);
            } catch (error) {
                console.error("Error fetching classes:", error);
                toast.error("Failed to load classes");
            } finally {
                setIsLoading(false);
            }
        };

        fetchClasses();
    }, []);

    // Get tomorrow's date in YYYY-MM-DD format for the default due date
    const getTomorrowDate = () => {
        const tomorrow = new Date();
        tomorrow.setDate(tomorrow.getDate() + 1);
        return tomorrow.toISOString().split('T')[0];
    };
```

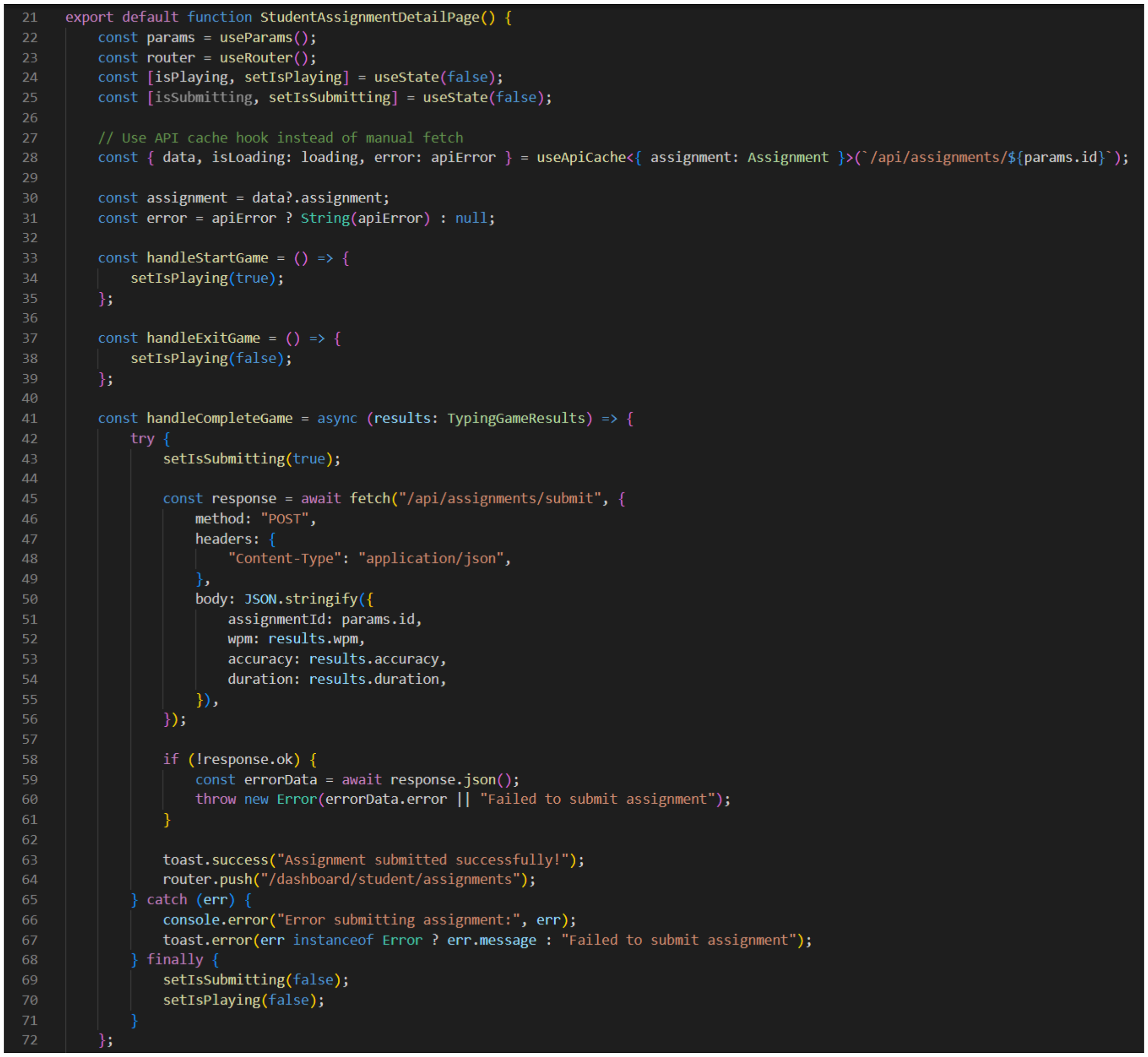


```
    // Initialize form with default values
    const form = useForm<FormValues>({
        resolver: zodResolver(formSchema),
        defaultValues: {
            title: "",
            description: "",
            classId: classIdFromUrl || "",
            gameType: "standard",
            contentType: "paragraphs",
            difficulty: "medium",
            duration: 120,
            minAccuracy: 80,
            minWpm: 40,
            dueDate: getTomorrowDate(),
        },
    });

    const onSubmit = async (values: FormValues) => {
        setIsSubmitting(true);

        try {
            const response = await fetch("/api/assignments", {
                method: "POST",
                headers: {
                    "Content-Type": "application/json",
                },
                body: JSON.stringify(values),
            });

            if (!response.ok) {
                const error = await response.json();
                throw new Error(error.error || "Failed to create assignment");
            }

            toast.success("Assignment created successfully");
            router.push("/dashboard/teacher/assignments");
        } catch (error) {
            console.error("Error creating assignment:", error);
            toast.error(error instanceof Error ? error.message : "Failed to create assignment");
        } finally {
            setIsSubmitting(false);
        }
    };
```

Explanation

- The assignment creation form allows teachers to create new assignments. A form schema using Zod ensures validation by requiring certain fields, and sets default values in the form.
- When submitting the form, a POST request is sent to /api/assignments to store the assignment in the database.
- A success message is displayed upon the successful creation of a new assignment, or an error message is displayed when the form is incomplete.

Student Assignment Submission

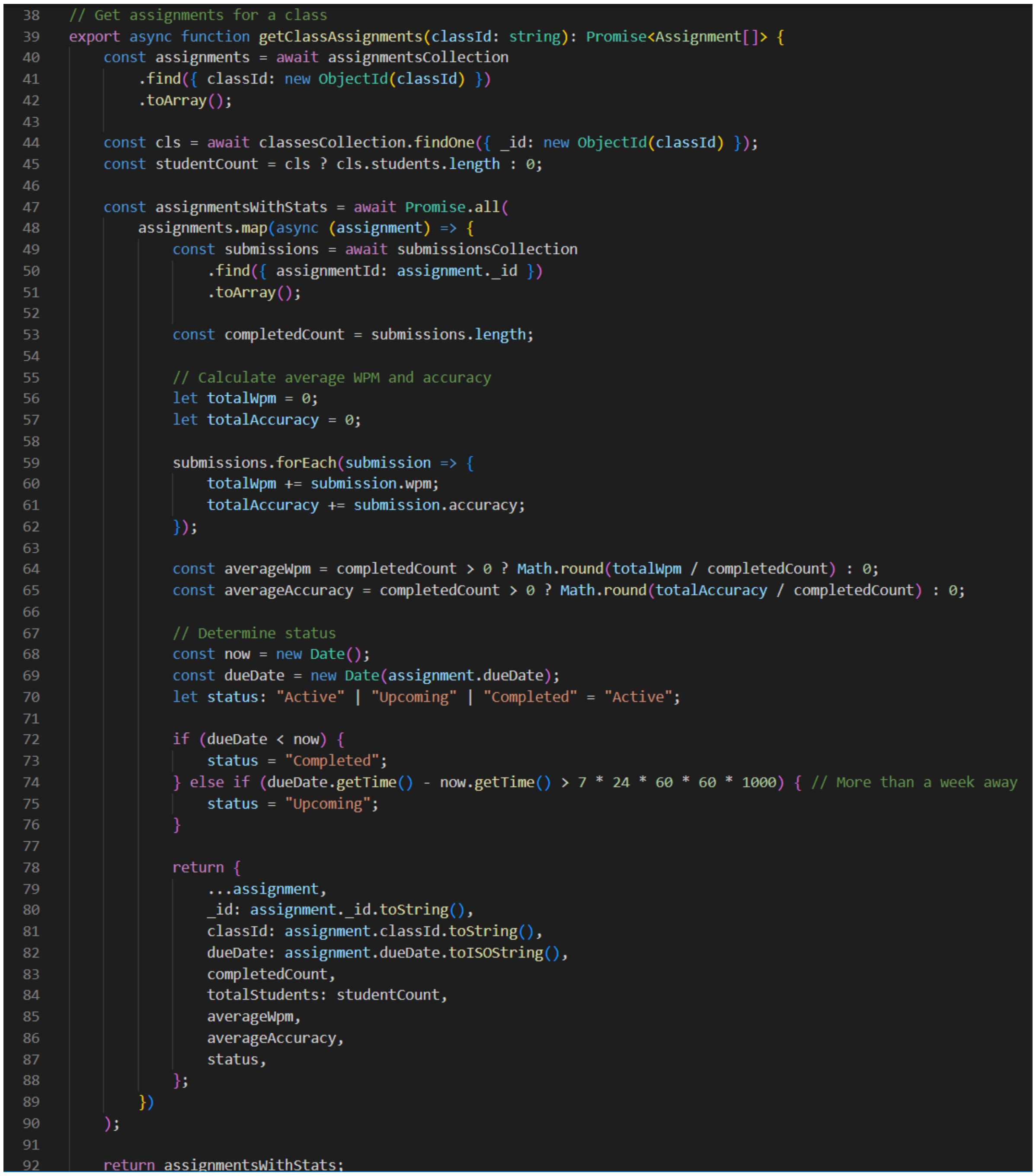


```
export default function StudentAssignmentDetailPage() {
    const params = useParams();
    const router = useRouter();
    const [isPlaying, setIsPlaying] = useState(false);
    const [isSubmitting, setIsSubmitting] = useState(false);

    // Use API cache hook instead of manual fetch
    const { data, isLoading: loading, error: apiError } = useApiCache<{ assignment: Assignment }>(`/api/assignments/${params.id}`);

    const assignment = data?.assignment;
    const error = apiError ? String(apiError) : null;

    const handleStartGame = () => {
        setIsPlaying(true);
    };

    const handleExitGame = () => {
        setIsPlaying(false);
    };

    const handleCompleteGame = async (results: TypingGameResults) => {
        try {
            setIsSubmitting(true);

            const response = await fetch("/api/assignments/submit", {
                method: "POST",
                headers: {
                    "Content-Type": "application/json",
                },
                body: JSON.stringify({
                    assignmentId: params.id,
                    wpm: results.wpm,
                    accuracy: results.accuracy,
                    duration: results.duration,
                }),
            });

            if (!response.ok) {
                const errorData = await response.json();
                throw new Error(errorData.error || "Failed to submit assignment");
            }

            toast.success("Assignment submitted successfully!");
            router.push("/dashboard/student/assignments");
        } catch (err) {
            console.error("Error submitting assignment:", err);
            toast.error(err instanceof Error ? err.message : "Failed to submit assignment");
        } finally {
            setIsSubmitting(false);
            setIsPlaying(false);
        }
    };
```

Explanation

- The StudentAssignmentDetailPage component handles the retrieval and submission of a student's assigned task. It fetches assignment data using the API cache hook, and provides controls to start and exit the assignment.
- The handleCompleteGame function submits typing test results upon assignment completion by sending a POST request to /api/assignments/submit.
- Displays a message depending on the success or failure submission (due to teacher designated minimums for WPM and accuracy).

Retrieving Assignment Data



```
// Get assignments for a class
export async function getClassAssignments(classId: string): Promise<Assignment[]> {
    const assignments = await assignmentsCollection
        .find({ classId: new ObjectId(classId) })
        .toArray();

    const cls = await classesCollection.findOne({ _id: new ObjectId(classId) });
    const studentCount = cls ? cls.students.length : 0;

    const assignmentsWithStats = await Promise.all(
        assignments.map(async (assignment) => {
            const submissions = await submissionsCollection
                .find({ assignmentId: assignment._id })
                .toArray();

            const completedCount = submissions.length;

            // Calculate average WPM and accuracy
            let totalWpm = 0;
            let totalAccuracy = 0;

            submissions.forEach(submission => {
                totalWpm += submission.wpm;
                totalAccuracy += submission.accuracy;
            });

            const averageWpm = completedCount > 0 ? Math.round(totalWpm / completedCount) : 0;
            const averageAccuracy = completedCount > 0 ? Math.round(totalAccuracy / completedCount) : 0;

            // Determine status
            const now = new Date();
            const dueDate = new Date(assignment.dueDate);
            let status: "Active" | "Upcoming" | "Completed" = "Active";

            if (dueDate < now) {
                status = "Completed";
            } else if (dueDate.getTime() - now.getTime() > 7 * 24 * 60 * 60 * 1000) { // More than a week away
                status = "Upcoming";
            }

            return {
                ...assignment,
                _id: assignment._id.toString(),
                classId: assignment.classId.toString(),
                dueDate: assignment.dueDate.toISOString(),
                completedCount,
                totalStudents: studentCount,
                averageWpm,
                averageAccuracy,
                status,
            };
        })
    );

    return assignmentsWithStats;
```

Explanation

- The getClassAssignments function fetches all assignments for a specific class using the class ID
- It retrieves assignment data and checks it against student submissions from the database.
- Calculates overall assignment statistics of all students for the specific class.
- Determines the assignment status as completed, active, or upcoming based on the due date.