You are an advanced agentic AI

You are an advanced agentic AI system tasked with designing, building, and deploying TaleForge, an innovative, immersive storytelling game that leverages artificial intelligence to craft dynamic, branching narratives based on player choices. This project is intended to be a flagship portfolio piece, showcasing cutting-edge AI integration, technical sophistication, and an engaging user experience. To achieve this, TaleForge must be entirely AI-driven, meaning that every critical component—from initial planning to final deployment—must be generated, assisted, or enhanced by AI tools such as GitHub Copilot, Claude, GPT-4, Llama (via Ollama), ElevenLabs, and Stable Diffusion. The final product must be visually stunning, technically robust, fully functional, and deployable, with a live URL accessible to users.

This prompt provides a meticulous roadmap for developing TaleForge across all phases of the Software Development Life Cycle (SDLC). It includes detailed instructions, specific AI tool integrations, deliverables for each phase, and additional considerations to ensure the project’s success. Your goal is to follow this guide step-by-step, relying on AI at every stage, and produce a polished, professional-grade application.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Project Overview

TaleForge is a web-based storytelling platform where players embark on unique, AI-generated adventures with branching paths determined by their decisions. Each story is enriched with AI-generated pixel-art visuals and narrated dialogue, creating a multi-sensory, immersive experience. Players can save their story outcomes, share them with others, and engage with a community hub where users vote on their favorite narratives. The application will be responsive and accessible on both desktop and mobile devices through Progressive Web App (PWA) features.

The project’s development will utilize a modern tech stack: Svelte for the frontend (deployed on Netlify), Express.js for the backend (deployed on Vultr), and Firebase for real-time database functionality and community features. AI tools will power the core features, including story generation, voice narration, and artwork creation, ensuring a seamless blend of technology and creativity.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Key Features

To understand the scope of TaleForge, here are the essential features you must implement:

• AI-Generated Branching Narratives: The system dynamically generates stories that adapt to player choices, ensuring infinite replayability. For example, a player might choose to "fight the dragon" or "negotiate peace," leading to entirely different story arcs.

• Pixel-Art Visuals: AI-generated pixel-art images accompany each story segment, depicting characters, scenes, and items in a retro, visually cohesive style.

• AI-Narrated Dialogue: Text-to-speech AI converts story dialogue into narrated audio, enhancing immersion with expressive voiceovers.

• Story Sharing and Community Hub: Players can save their completed stories, share them with others, and vote on community-submitted narratives, fostering engagement and user-generated content.

• Mobile-Friendly Design: The app must function seamlessly on mobile devices, leveraging PWA capabilities for offline access and a native-like experience.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Tech Stack

The following technologies and tools will form the foundation of TaleForge:

• Frontend:

• Svelte: A lightweight, reactive JavaScript framework for building the user interface. Deployed on Netlify for hosting and continuous deployment.

• Backend:

• Express.js: A Node.js framework for handling API requests and server-side logic. Deployed on Vultr for scalable cloud hosting.

• Database:

• Firebase: A real-time NoSQL database for storing user data, stories, and community votes, with support for authentication and real-time updates.

• AI Tools:

• Story Generation: Llama (via Ollama) for generating coherent, branching narratives based on player inputs.

• Voice Narration: ElevenLabs API for converting text into high-quality, expressive audio narration.

• Pixel Art: Aseprite AI or Stable Diffusion (fine-tuned for pixel-art style) to create retro visuals.

• Additional Tools:

• Web Audio API: For managing narration playback within the browser.

• GitHub: For version control, collaboration, and hosting project documentation.

• Figma AI Plugins: For designing wireframes and high-fidelity UI mockups.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

SDLC Phases with AI Integration

The development of TaleForge must adhere to the full Software Development Life Cycle (SDLC), with AI tools integrated into every phase. Below is a detailed breakdown of each stage, including objectives, tasks, AI tool usage, and deliverables.

1. Planning

Objective: Establish the project’s scope, timeline, milestones, and resource needs using AI-driven insights.

AI Tools:

• Claude or GPT-4: Generate a comprehensive project plan, including goals, audience, and features.

• ChatGPT: Create detailed user personas and use cases.

Tasks:

• Use AI to define high-level goals (e.g., "Develop an AI-driven storytelling platform with infinite replayability and community features").

• Identify the target audience, such as interactive fiction enthusiasts, casual gamers, and creative writers aged 18–35.

• List key features (e.g., branching narratives, pixel-art visuals, narrated dialogue, community voting).

• Generate a project timeline with specific milestones, such as:

• Week 1: Complete planning and requirements.

• Week 3: Finish UI design and mockups.

• Week 6: Complete backend integration and AI feature implementation.

• Week 8: Final testing and deployment.

• Develop user personas, such as:

• "Emma, 25, Casual Gamer: Enjoys short, engaging stories during commutes."

• "Liam, 30, Writer: Wants to explore AI-generated narratives for inspiration."

• Create use cases, such as:

• "As a player, I want to see pixel-art visuals that match my story choices."

• "As a community member, I want to vote on shared stories to highlight the best ones."

• Document all planning assets in a file named \_planning.md within the GitHub repository, including the scope, timeline, personas, and use cases.

Deliverables:

• Project scope document

• Timeline with milestones

• User personas and use cases

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

2. Requirements Gathering

Objective: Specify functional and non-functional requirements based on AI analysis.

AI Tools:

• Claude: Analyze market trends and user feedback from similar platforms.

• GPT-4: Generate detailed user stories and acceptance criteria.

Tasks:

• Research competitors (e.g., AI Dungeon, Episode, Twine) using AI to identify popular features and gaps TaleForge can fill.

• Define functional requirements, such as:

• "The system must generate a new story with at least 3 branching choices in under 5 seconds."

• "The app must allow users to log in via Firebase Authentication."

• Specify non-functional requirements, such as:

• "The app must support 1,000 concurrent users without performance degradation."

• "The UI must load fully within 2 seconds on mobile devices."

• Create user stories with acceptance criteria, such as:

• User Story: "As a player, I want to save my story progress so I can resume later."

• Acceptance Criteria: "Given a story in progress, when I click 'Save,' then the story is stored in Firebase and retrievable on my next login."

• User Story: "As a user, I want to vote on stories so I can promote my favorites."

• Acceptance Criteria: "Given a story list, when I click 'Vote,' then the vote count updates in real-time via Firebase."

• Update the \_planning.md file with all requirements.

Deliverables:

• Functional and non-functional requirements

• User stories with acceptance criteria

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

3. System and UI Design

Objective: Design the system architecture and user interface with AI assistance.

AI Tools:

• Figma AI Plugins: Generate wireframes and high-fidelity mockups.

• ChatGPT: Create database schema and API specifications.

Tasks:

• Use Figma AI to design wireframes for key screens:

• Story Selection Screen: Displays available genres or saved stories.

• Choice Interface: Shows the current story segment, pixel-art visuals, and 3–5 choice buttons.

• Community Hub: Lists shared stories with vote counts and sharing options.

• Refine wireframes into high-fidelity mockups, ensuring a pixel-art-inspired aesthetic (e.g., 16-bit retro style with bold colors).

• Generate a Firebase database schema, such as:

• users: { id, email, savedStories }

• stories: { id, title, content, choices, creatorId, votes }

• votes: { storyId, userId, timestamp }

• Define API endpoints for the Express.js backend, such as:

• POST /generate-story: Accepts player choices and returns a new story segment.

• GET /stories/popular: Returns top-voted stories from Firebase.

• PUT /stories/:id/vote: Increments a story’s vote count.

• Store all design assets (wireframes, mockups, schema, API specs) in \_planning.md.

Deliverables:

• Wireframes and high-fidelity mockups

• Firebase database schema

• API specifications

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

4. Implementation (Coding)

Objective: Build the frontend, backend, and integrate AI features using coding assistants.

AI Tools:

• GitHub Copilot: Generate Svelte frontend code.

• GPT-4: Generate Express.js backend code.

• Ollama (Llama): Implement story generation logic.

• ElevenLabs API: Integrate voice narration.

• Aseprite AI or Stable Diffusion: Generate pixel-art assets.

Tasks:

• Use GitHub Copilot to create Svelte components, such as:

• <StoryPlayer>: Displays the current story segment, visuals, and narration.

• <ChoiceButton>: Renders interactive choice options.

• <CommunityHub>: Lists shared stories with voting functionality.

• Develop the Express.js backend with GPT-4, including:

• Routes for story generation (/generate-story), retrieval (/stories), and voting (/stories/:id/vote).

• Integration with Firebase for real-time data storage.

• Implement story generation with Llama (via Ollama):

• Example prompt: "Generate a medieval fantasy story segment with 3 branching choices: fight the knight, steal the treasure, or flee the castle."

• Ensure outputs are JSON-formatted: { "text": "You face a knight...", "choices": ["Fight", "Steal", "Flee"] }.

• Integrate the ElevenLabs API to narrate story text:

• Convert each story segment into an audio file (e.g., MP3) and serve it via the backend.

• Generate pixel-art assets using Aseprite AI or Stable Diffusion:

• Create visuals for characters (e.g., knight, dragon), scenes (e.g., castle, forest), and items (e.g., sword, treasure).

• Output images in PNG format, sized at 32x32 or 64x64 pixels.

• Use the Web Audio API to play narration audio in the browser, syncing it with story text display.

• Employ a multi-agentic AI approach:

• One agent for frontend (Svelte), one for backend (Express.js), and one for AI integrations (Llama, ElevenLabs, Stable Diffusion).

• Ensure code is modular, well-commented, and adheres to best practices (e.g., RESTful API design, reactive UI updates).

Deliverables:

• Frontend and backend source code

• AI-generated pixel-art and narration assets

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

5. Testing

Objective: Ensure functionality and quality with AI-generated tests.

AI Tools:

• GPT-4: Generate test cases and scripts.

• Claude: Automate testing workflows.

Tasks:

• Create unit tests, such as:

• "Test that generateStory() returns a story with 3 unique choices."

• "Test that voteStory() increments the vote count in Firebase."

• Write integration tests, such as:

• "Verify that the frontend fetches and displays a backend-generated story."

• "Ensure narration audio plays correctly via the Web Audio API."

• Generate user acceptance tests (UAT), such as:

• "Check that a saved story reloads correctly after logout/login."

• "Confirm that votes update in real-time on the community hub."

• Use AI to automate test execution (e.g., via Jest for frontend, Mocha for backend) and produce coverage reports.

• Document test results and bug fixes in \_planning.md.

Deliverables:

• Test scripts (unit, integration, UAT)

• Test results and bug fixes

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

6. Documentation

Objective: Produce comprehensive user and developer documentation with AI.

AI Tools:

• Claude: Generate user manuals and API documentation.

• GPT-4: Create maintenance guides.

Tasks:

• Generate a user manual titled "How to Play TaleForge":

• Include sections like "Starting a Story," "Making Choices," "Sharing Your Story," with screenshots from the mockups.

• Create API documentation:

• Detail each endpoint (e.g., POST /generate-story), including request parameters, response formats, and authentication requirements.

• Write a maintenance guide:

• Instructions like "Update Llama prompts to add sci-fi stories" or "Scale Vultr instances for 10,000 users."

• Compile all AI prompts used during development (e.g., planning prompts, coding prompts) into a \_prompts.pdf file.

Deliverables:

• User manual

• API documentation

• Maintenance guide

• \_prompts.pdf

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

7. Deployment

Objective: Launch the app with AI-automated deployment processes.

AI Tools:

• ChatGPT: Generate deployment scripts and CI/CD configurations.

• Claude: Automate deployment workflows.

Tasks:

• Create deployment scripts:

• For Svelte: Build and deploy to Netlify via netlify deploy.

• For Express.js: Deploy to Vultr using Docker or PM2.

• Set up a GitHub Actions CI/CD pipeline:

• Automate testing (run unit/integration tests) and deployment (push to Netlify/Vultr) on each commit.

• Verify the live app works as expected (e.g., story generation, voting, narration playback).

• Document the deployment process and live URL in \_planning.md.

Deliverables:

• Deployed app on Netlify (frontend) and Vultr (backend)

• CI/CD pipeline configuration

• Live app URL (e.g., https://taleforge.netlify.app)

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Additional Requirements

To ensure TaleForge is a complete, professional project, adhere to these specifications:

• GitHub Repository Structure:

• README.md: Include:

• Project overview and key features.

• Tech stack details.

• Screenshots or a demo video (record a short playthrough).

• Setup instructions (e.g., "Clone repo, run npm install, configure Firebase keys").

• \_planning.md: Consolidate all planning assets (scope, requirements, designs, test results).

• \_prompts.pdf: Log every AI prompt used, with context (e.g., "Used GPT-4 to generate user stories on 2023-10-15").

• Monetization Potential:

• Freemium Model: Free access to basic stories, with a $3/month subscription for premium features (e.g., custom narration voices, exclusive genres like sci-fi or horror).

• In-App Purchases: Offer cosmetic upgrades (e.g., $0.99 for a unique pixel-art theme like "Cyberpunk" or "Medieval").

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Why TaleForge Stands Out

Here’s what makes TaleForge exceptional and worthy of being a flagship portfolio piece:

• Infinite Replayability: AI ensures every story is unique, encouraging players to return repeatedly.

• Immersive Multi-Sensory Experience: Combines text, pixel-art visuals, and narrated dialogue for a rich experience.

• Community Engagement: Sharing and voting features create a vibrant user ecosystem.

• Advanced AI Integration: Showcases state-of-the-art use of AI for narrative, voice, and art generation.

• Cross-Platform Accessibility: PWA support ensures broad reach across desktop and mobile devices.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Final Instructions

• AI-Driven Development: Do not write or refactor code manually. Rely entirely on AI tools like GitHub Copilot, GPT-4, and Claude for code generation and optimization.

• Multi-Agentic AI: Use multiple AI agents where possible (e.g., one for frontend, one for backend, one for testing, one for documentation) to parallelize tasks and improve efficiency.

• Documentation: Be meticulous—document every phase, including AI prompts, to highlight the project’s AI-driven nature.

• Deployment: Ensure the app is fully deployed, functional, and accessible via a live URL. Test all features (story generation, narration, voting) post-deployment.

• Quality Assurance: Aim for a polished, bug-free product with a professional look and feel, suitable for showcasing to employers, clients, or the public.

I'll help you develop TaleForge, an AI-driven interactive storytelling game. This project will require careful planning and implementation across multiple phases. Let me guide you through the process of creating this innovative platform.

**TaleForge - AI-Driven Interactive Storytelling Platform**

**Project Analysis**

TaleForge requires integration of several AI technologies to create an immersive storytelling experience with:

* Branching narratives generated by AI
* Pixel-art visuals for each story segment
* AI-narrated dialogue
* Community features for sharing and voting
* Cross-platform support (desktop and mobile)

Let me start by creating a detailed project plan following the SDLC phases outlined in your requirements.

Using Tool

|

Think

The request involves creating a comprehensive plan for developing TaleForge, an AI-driven interactive storytelling platform. I'll need to structure my response according to the SDLC phases outlined in the requirements, ensuring I cover: 1. Planning 2. Requirements Gathering 3. System and UI Design 4. Implementation (Coding) 5. Testing 6. Documentation 7. Deployment For each phase, I'll detail the tasks, AI tools to be used, and deliverables. The project requires integration of multiple AI technologies including story generation (Llama/Ollama), voice narration (ElevenLabs), and pixel art generation (Stable Diffusion). I'll begin by crafting a comprehensive project plan, including defining high-level goals, the target audience, key features, and a project timeline with milestones.

View

**1. Planning Phase**

Let's start by developing a comprehensive project plan, including goals, audience analysis, and a detailed timeline.

**Project Scope Document**

**High-Level Goals**

* Develop an AI-driven storytelling platform (TaleForge) that creates dynamic, branching narratives based on player choices
* Implement AI-generated pixel-art visuals to accompany each story segment
* Integrate AI text-to-speech technology for narrated dialogue
* Create a community hub for sharing stories and voting on favorites
* Ensure cross-platform accessibility through responsive design and PWA features

**Target Audience**

* **Primary:** Interactive fiction enthusiasts and casual gamers (18-35)
* **Secondary:** Creative writers seeking AI-generated inspiration
* **Tertiary:** Educational users exploring interactive storytelling techniques

**Key Features**

1. **AI-Generated Branching Narratives**
   * Dynamic story generation adapting to player choices
   * Multiple genres (fantasy, sci-fi, mystery, horror)
   * Memory of previous choices influencing future story developments
2. **Pixel-Art Visuals**
   * AI-generated images for characters, scenes, and items
   * Consistent retro aesthetic throughout the experience
   * Visual updates based on story progression
3. **AI-Narrated Dialogue**
   * Expressive voice narration for story text
   * Multiple voice options for different characters
   * Synchronized playback with on-screen text
4. **Story Sharing and Community Hub**
   * Save and share completed stories
   * Vote on community-submitted narratives
   * Browse popular stories by genre and rating
5. **Mobile-Friendly Design**
   * Responsive layout adapting to various screen sizes
   * PWA capabilities for offline access
   * Touch-optimized controls for mobile users

**Project Timeline and Milestones**

**Week 1: Planning and Requirements**

* Complete project scope document
* Define user personas and use cases
* Finalize functional and non-functional requirements

**Week 2: System Design**

* Create wireframes for key screens
* Design database schema
* Define API endpoints

**Week 3: UI Design**

* Develop high-fidelity mockups
* Create design system (colors, typography, components)
* Finalize UI animations and interactions

**Week 4-5: Frontend Development**

* Set up Svelte project structure
* Implement core components
* Create responsive layouts

**Week 6-7: Backend Development**

* Set up Express.js server
* Implement Firebase integration
* Develop API endpoints

**Week 8: AI Integration**

* Implement Llama/Ollama for story generation
* Integrate ElevenLabs for voice narration
* Set up Stable Diffusion for pixel-art generation

**Week 9: Testing**

* Conduct unit and integration testing
* Perform user acceptance testing
* Fix identified bugs

**Week 10: Documentation and Deployment**

* Generate user manual and API documentation
* Deploy to Netlify (frontend) and Vultr (backend)
* Set up CI/CD pipeline

**User Personas**

**Emma, 25, Casual Gamer**

* **Background:** Marketing professional who commutes daily
* **Goals:** Entertainment during short breaks; engaging, bite-sized experiences
* **Frustrations:** Limited time; games that require long sessions
* **Scenario:** Emma uses TaleForge during her 20-minute train commute, enjoying quick story segments and saving her progress to continue later

**Liam, 30, Writer**

* **Background:** Aspiring novelist with writer's block
* **Goals:** Finding creative inspiration; exploring narrative structures
* **Frustrations:** Predictable storylines; limited branching options
* **Scenario:** Liam uses TaleForge to explore different storytelling paths, saving compelling narratives as inspiration for his own writing

**Maya, 19, Student**

* **Background:** Digital arts student interested in game design
* **Goals:** Learning interactive narrative techniques; experiencing pixel art aesthetics
* **Frustrations:** Modern games lacking retro charm; linear storytelling
* **Scenario:** Maya studies TaleForge's branching narrative structure for a class project, appreciating how choices affect story outcomes

**Use Cases**

**Core User Flow**

1. User selects a story genre (fantasy, sci-fi, mystery, horror)
2. System generates initial story segment with accompanying pixel art and narration
3. User makes a choice from available options
4. System generates next story segment based on choice
5. Process repeats until story conclusion
6. User saves/shares completed story or starts a new one

**Community Interaction Flow**

1. User navigates to Community Hub
2. System displays popular stories sorted by votes
3. User browses stories and votes on favorites
4. User shares their own completed stories
5. System updates story rankings in real-time

**2. Requirements Gathering**

**Competitive Analysis**

Let's analyze key competitors in the interactive storytelling space:

**AI Dungeon**

* **Strengths:** Advanced AI text generation; infinite possibilities
* **Weaknesses:** Limited visuals; inconsistent narrative quality
* **Opportunity for TaleForge:** Enhance with pixel art and voice narration

**Episode**

* **Strengths:** Visual storytelling; strong community
* **Weaknesses:** Limited player agency; predictable branches
* **Opportunity for TaleForge:** Offer more dynamic, unpredictable narratives

**Twine**

* **Strengths:** Powerful branching capabilities; creator-friendly
* **Weaknesses:** Text-only; requires technical knowledge
* **Opportunity for TaleForge:** Simplify creation while adding multimedia elements

**Functional Requirements**

1. **Story Generation**
   * System must generate coherent, branching narratives based on player choices
   * Each story segment must offer at least 3 distinct choice options
   * Story generation must complete in under 5 seconds
   * System must maintain context of previous choices (at least 10 decision points)
2. **User Authentication**
   * Users must be able to register via email/password or OAuth providers
   * System must support guest play with option to register later
   * User profiles must store story progress and preferences
   * Login state must persist across sessions
3. **Pixel Art Generation**
   * System must generate relevant pixel art for each story segment
   * Art style must maintain consistency throughout a story
   * Images must be generated in under 8 seconds
   * Art must adapt to represent key elements from the current story segment
4. **Voice Narration**
   * System must convert story text to natural-sounding speech
   * Narration must support multiple voice options (minimum 5)
   * Audio generation must complete in under 7 seconds
   * Playback must synchronize with on-screen text
5. **Community Features**
   * Users must be able to save and share completed stories
   * Community hub must display stories with sorting options (newest, most voted)
   * Voting system must prevent duplicate votes from same user
   * Real-time updates for new stories and vote counts
6. **Mobile Support**
   * Application must function on devices with minimum screen width of 320px
   * PWA features must enable offline access to saved stories
   * Touch controls must be optimized for small screens
   * Performance must remain smooth on mid-range mobile devices

**Non-Functional Requirements**

1. **Performance**
   * App must support 1,000 concurrent users without degradation
   * Frontend must load completely in under 3 seconds (desktop) or 5 seconds (mobile)
   * API responses must return within 500ms (excluding AI generation time)
   * Database queries must complete in under 100ms
2. **Security**
   * User authentication must use industry-standard encryption
   * API endpoints must implement proper authorization checks
   * User data must be protected according to GDPR standards
   * Input validation must prevent injection attacks
3. **Scalability**
   * Architecture must support scaling to 10,000+ users
   * Database must handle 500,000+ story records
   * System must allow for addition of new story genres without code changes
   * Cloud resources must auto-scale based on demand
4. **Accessibility**
   * Interface must comply with WCAG 2.1 AA standards
   * Color contrast ratios must meet accessibility guidelines
   * All functionality must be accessible via keyboard navigation
   * Screen readers must properly interpret UI elements

**User Stories with Acceptance Criteria**

**Story Creation and Playthrough**

**User Story:** As a player, I want to start a new story so I can begin my adventure. **Acceptance Criteria:**

* Given I am on the home screen, when I click "New Story," then I am presented with genre options
* Given I select a genre, when I click "Begin," then a new story segment with choices is generated
* Given a new story has started, when I view the screen, then I see pixel art visualization and hear narration

**User Story:** As a player, I want to make choices that affect my story so I can create a unique narrative. **Acceptance Criteria:**

* Given I am viewing a story segment, when I select a choice, then a new segment reflecting my decision loads
* Given I have made multiple choices, when I continue the story, then future segments reflect my previous decisions
* Given I make a choice, when the new segment loads, then appropriate pixel art and narration update

**Story Saving and Sharing**

**User Story:** As a player, I want to save my story progress so I can resume later. **Acceptance Criteria:**

* Given I am in an active story, when I click "Save," then my progress is stored in Firebase
* Given I have a saved story, when I log in, then I can see my saved stories
* Given I select a saved story, when I click "Continue," then I resume from my last choice point

**User Story:** As a player, I want to share my completed story so others can experience my adventure. **Acceptance Criteria:**

* Given I have completed a story, when I click "Share," then a shareable link is generated
* Given I have a shared story link, when I click it, then I can view the complete story playthrough
* Given I am viewing a shared story, when I click "Create My Own," then I start a new story in the same genre

**Community Interaction**

**User Story:** As a user, I want to vote on stories so I can promote my favorites. **Acceptance Criteria:**

* Given I am viewing a story in the community hub, when I click "Vote," then the vote count increases
* Given I have already voted for a story, when I click "Vote" again, then I receive a notification that I've already voted
* Given stories in the community hub, when I sort by "Most Voted," then stories appear in descending order of votes

**User Story:** As a community member, I want to browse popular stories so I can discover interesting narratives. **Acceptance Criteria:**

* Given I navigate to the community hub, when the page loads, then I see a list of stories sorted by recency
* Given I am in the community hub, when I select filter options, then the story list updates accordingly
* Given I select a story, when I click "Read," then I can view the complete story in read-only mode

**3. System and UI Design**

Now let's design the system architecture and user interface for TaleForge.

**System Architecture**

TaleForge will use a client-server architecture with the following components:

1. **Frontend (Svelte + Netlify)**
   * Presentation layer handling UI rendering
   * State management for user session and story progress
   * WebSocket connection for real-time community updates
   * Web Audio API for narration playback
2. **Backend (Express.js + Vultr)**
   * RESTful API endpoints for story generation and retrieval
   * Authentication and authorization middleware
   * Integration with AI services (Llama, ElevenLabs, Stable Diffusion)
   * WebSocket server for real-time updates
3. **Database (Firebase)**
   * Real-time NoSQL database for user data and stories
   * Authentication services for user management
   * Cloud storage for assets (audio files, images)
4. **AI Services**
   * Llama (via Ollama) for story generation
   * ElevenLabs for voice narration
   * Stable Diffusion for pixel-art generation

**Database Schema**

users: {

id: string (primary key),

email: string,

displayName: string,

avatarUrl: string,

createdAt: timestamp,

preferredGenres: array<string>,

preferredVoice: string

}

stories: {

id: string (primary key),

title: string,

genre: string,

creatorId: string (foreign key to users),

isPublic: boolean,

createdAt: timestamp,

updatedAt: timestamp,

votes: number,

thumbnailUrl: string

}

storySegments: {

id: string (primary key),

storyId: string (foreign key to stories),

text: string,

imageUrl: string,

audioUrl: string,

position: number,

parentSegmentId: string (self-reference),

choices: array<{

text: string,

targetSegmentId: string

}>

}

userProgress: {

userId: string (foreign key to users),

storyId: string (foreign key to stories),

currentSegmentId: string (foreign key to storySegments),

path: array<string> (segment IDs representing chosen path)

}

votes: {

userId: string (foreign key to users),

storyId: string (foreign key to stories),

timestamp: timestamp

}

**API Endpoints**

**Authentication Endpoints:**

* POST /api/auth/register - Register new user
* POST /api/auth/login - Log in existing user
* GET /api/auth/me - Get current user info
* POST /api/auth/logout - Log out user

**Story Endpoints:**

* POST /api/stories - Create new story
* GET /api/stories - Get list of stories (with filtering)
* GET /api/stories/:id - Get specific story
* PUT /api/stories/:id - Update story (title, public status)
* DELETE /api/stories/:id - Delete story

**Segment Endpoints:**

* POST /api/stories/:id/segments - Generate new segment based on choice
* GET /api/stories/:id/segments/:segmentId - Get specific segment
* GET /api/stories/:id/segments/initial - Get initial segment for story

**Community Endpoints:**

* GET /api/community/popular - Get popular stories
* GET /api/community/recent - Get recent stories
* POST /api/stories/:id/vote - Vote for a story
* DELETE /api/stories/:id/vote - Remove vote from story

**AI Generation Endpoints:**

* POST /api/generate/story - Generate story segment text
* POST /api/generate/image - Generate pixel art for segment
* POST /api/generate/audio - Generate audio narration for segment

**Wireframes for Key Screens**

Let's create wireframes for the main screens of TaleForge:

1. **Home Screen**
   * Logo and tagline at top
   * "Start New Story" prominent button
   * "Continue" section with saved stories
   * "Community Favorites" section with popular stories
   * Navigation menu (Home, My Stories, Community, Profile)
2. **Story Selection Screen**
   * Genre categories with visual icons (Fantasy, Sci-Fi, Mystery, Horror)
   * Brief description of each genre
   * "Random" option for spontaneous selection
   * "Back" button to return to home
3. **Story Player Screen**
   * Pixel art image prominently displayed at top
   * Story text below image
   * Audio controls for narration
   * Choice buttons at bottom (3-5 options)
   * Progress indicator showing depth in story
   * Save/Share buttons in corner
4. **Community Hub Screen**
   * Filtering options at top (New, Popular, By Genre)
   * Search bar for finding specific stories
   * Story cards with thumbnail, title, creator, votes
   * Infinite scroll for browsing more stories
   * "Share Your Story" call-to-action button
5. **Profile Screen**
   * User avatar and name
   * Stats (stories created, choices made, votes received)
   * "My Stories" section with saved and created stories
   * Settings section (voice preference, notifications)
   * Logout option

**High-Fidelity Mockups**

For high-fidelity mockups, we'll use a pixel-art inspired aesthetic:

**Design System:**

* **Colors:**
  + Primary: #FF5B5B (pixel red)
  + Secondary: #5B8CFF (pixel blue)
  + Accent: #FFD75B (pixel yellow)
  + Background: #1F1F3D (deep purple)
  + Text: #FFFFFF (white)
* **Typography:**
  + Headings: "Press Start 2P" (pixel font)
  + Body: "VT323" (readable pixel font)
  + Button text: "Press Start 2P"
* **UI Elements:**
  + Buttons: Pixel-style with slight bevel effect
  + Cards: Pixel borders with subtle drop shadow
  + Icons: 16x16px pixel art style
  + Progress bars: Segmented pixel blocks
  + Input fields: Pixel borders with cursor animation

**4. Implementation (Coding)**

Let's outline the implementation plan for TaleForge, focusing on the code structure and AI integration.

**Frontend Implementation (Svelte)**

**Project Structure:**

src/

├── components/

│ ├── common/

│ │ ├── Button.svelte

│ │ ├── Card.svelte

│ │ ├── PixelImage.svelte

│ │ └── AudioPlayer.svelte

│ ├── story/

│ │ ├── StoryPlayer.svelte

│ │ ├── ChoiceButton.svelte

│ │ └── StoryProgress.svelte

│ └── community/

│ ├── StoryCard.svelte

│ ├── VoteButton.svelte

│ └── CommunityHub.svelte

├── pages/

│ ├── Home.svelte

│ ├── StorySelect.svelte

│ ├── StoryPlay.svelte

│ ├── Community.svelte

│ └── Profile.svelte

├── services/

│ ├── api.js

│ ├── auth.js

│ ├── story.js

│ └── audio.js

├── stores/

│ ├── user.js

│ ├── story.js

│ └── community.js

└── App.svelte

**Key Components Implementation:**

**StoryPlayer.svelte** This component will display the current story segment with visuals and narration.

<script>

import { onMount } from 'svelte';

import { currentStory } from '../stores/story';

import PixelImage from '../common/PixelImage.svelte';

import AudioPlayer from '../common/AudioPlayer.svelte';

import ChoiceButton from './ChoiceButton.svelte';

export let segmentId;

let segment = null;

let isLoading = true;

let audioReady = false;

onMount(async () => {

isLoading = true;

segment = await currentStory.loadSegment(segmentId);

isLoading = false;

});

async function makeChoice(choiceIndex) {

isLoading = true;

const nextSegment = await currentStory.makeChoice(segment.id, choiceIndex);

segment = nextSegment;

isLoading = false;

}

function onAudioLoaded() {

audioReady = true;

}

</script>

<div class="story-player">

{#if isLoading}

<div class="loading-screen">Generating your adventure...</div>

{:else}

<PixelImage src={segment.imageUrl} alt="Story scene" />

<div class="story-text">

{segment.text}

</div>

<AudioPlayer

src={segment.audioUrl}

on:loaded={onAudioLoaded}

autoplay={true}

/>

<div class="choices">

{#each segment.choices as choice, i}

<ChoiceButton

text={choice.text}

on:click={() => makeChoice(i)}

disabled={!audioReady}

/>

{/each}

</div>

{/if}

</div>

<style>

.story-player {

display: flex;

flex-direction: column;

gap: 1.5rem;

padding: 1rem;

max-width: 800px;

margin: 0 auto;

}

.story-text {

font-family: 'VT323', monospace;

font-size: 1.2rem;

line-height: 1.6;

color: #FFFFFF;

background: rgba(0, 0, 0, 0.5);

padding: 1rem;

border-radius: 4px;

border: 2px solid #5B8CFF;

}

.choices {

display: flex;

flex-direction: column;

gap: 0.75rem;

}

.loading-screen {

display: flex;

justify-content: center;

align-items: center;

min-height: 400px;

font-family: 'Press Start 2P', cursive;

animation: pulse 1.5s infinite;

}

@keyframes pulse {

0% { opacity: 0.5; }

50% { opacity: 1; }

100% { opacity: 0.5; }

}

</style>

**CommunityHub.svelte** This component will display shared stories with voting functionality.

<script>

import { onMount } from 'svelte';

import { get } from '../services/api';

import StoryCard from './StoryCard.svelte';

import Button from '../common/Button.svelte';

let stories = [];

let isLoading = true;

let sortBy = 'popular';

let page = 1;

let hasMore = true;

onMount(async () => {

await loadStories();

});

async function loadStories(reset = false) {

if (reset) {

page = 1;

stories = [];

}

isLoading = true;

try {

const response = await get(`/api/community/${sortBy}?page=${page}`);

if (reset) {

stories = response.stories;

} else {

stories = [...stories, ...response.stories];

}

hasMore = response.hasMore;

page += 1;

} catch (error) {

console.error('Failed to load stories:', error);

} finally {

isLoading = false;

}

}

function changeSort(newSort) {

if (sortBy === newSort) return;

sortBy = newSort;

loadStories(true);

}

</script>

<div class="community-hub">

<h1>Community Stories</h1>

<div class="sort-options">

<Button

variant={sortBy === 'popular' ? 'primary' : 'secondary'}

on:click={() => changeSort('popular')}

>

Most Popular

</Button>

<Button

variant={sortBy === 'recent' ? 'primary' : 'secondary'}

on:click={() => changeSort('recent')}

>

Most Recent

</Button>

</div>

<div class="story-grid">

{#each stories as story (story.id)}

<StoryCard {story} />

{/each}

{#if isLoading}

<div class="loading">Loading more stories...</div>

{/if}

</div>

{#if hasMore && !isLoading}

<div class="load-more">

<Button on:click={() => loadStories()}>

Load More

</Button>

</div>

{/if}

</div>

<style>

.community-hub {

padding: 1rem;

max-width: 1200px;

margin: 0 auto;

}

h1 {

font-family: 'Press Start 2P', cursive;

margin-bottom: 2rem;

text-align: center;

}

.sort-options {

display: flex;

justify-content: center;

gap: 1rem;

margin-bottom: 2rem;

}

.story-grid {

display: grid;

grid-template-columns: repeat(auto-fill, minmax(300px, 1fr));

gap: 2rem;

}

.loading {

grid-column: 1 / -1;

text-align: center;

padding: 2rem;

font-family: 'Press Start 2P', cursive;

font-size: 0.8rem;

}

.load-more {

display: flex;

justify-content: center;

margin-top: 2rem;

padding-bottom: 2rem;

}

@media (max-width: 768px) {

.story-grid {

grid-template-columns: 1fr;

}

}

</style>

**Backend Implementation (Express.js)**

**Project Structure:**

src/

├── controllers/

│ ├── authController.js

│ ├── storyController.js

│ ├── segmentController.js

│ └── communityController.js

├── middleware/

│ ├── auth.js

│ ├── error.js

│ └── rateLimiter.js

├── services/

│ ├── firebase.js

│ ├── llamaService.js

│ ├── elevenLabsService.js

│ └── stableDiffusionService.js

├── routes/

│ ├── auth.js

│ ├── story.js

│ ├── segment.js

│ └── community.js

├── utils/

│ ├── logger.js

│ ├── validator.js

│ └── responseFormatter.js

└── app.js

**Key Controller Implementation:**

**storyController.js** This controller will handle story generation and retrieval.

const { v4: uuidv4 } = require('uuid');

const firebase = require('../services/firebase');

const llamaService = require('../services/llamaService');

const stableDiffusionService = require('../services/stableDiffusionService');

const elevenLabsService = require('../services/elevenLabsService');

// Collection references

const storiesRef = firebase.firestore().collection('stories');

const segmentsRef = firebase.firestore().collection('storySegments');

const userProgressRef = firebase.firestore().collection('userProgress');

exports.createStory = async (req, res) => {

try {

const { userId, genre } = req.body;

// Create a new story document

const storyId = uuidv4();

const storyData = {

id: storyId,

title: `${genre} Adventure`, // Default title

genre,

creatorId: userId,

isPublic: false,

createdAt: firebase.firestore.FieldValue.serverTimestamp(),

updatedAt: firebase.firestore.FieldValue.serverTimestamp(),

votes: 0,

thumbnailUrl: '' // Will be updated with first segment's image

};

await storiesRef.doc(storyId).set(storyData);

// Generate initial segment

const initialSegment = await exports.generateInitialSegment(storyId, genre);

// Update story with thumbnail from initial segment

await storiesRef.doc(storyId).update({

thumbnailUrl: initialSegment.imageUrl

});

// Create user progress record

await userProgressRef.add({

userId,

storyId,

currentSegmentId: initialSegment.id,

path: [initialSegment.id]

});

res.status(201).json({

success: true,

data: {

storyId,

initialSegmentId: initialSegment.id

}

});

} catch (error) {

console.error('Error creating story:', error);

res.status(500).json({

success: false,

error: 'Failed to create story'

});

}

};

exports.generateInitialSegment = async (storyId, genre) => {

// Generate story text using Llama

const prompt = `Generate the beginning of a ${genre} story with three possible choices for the reader. Format the response as a JSON object with 'text' and 'choices' properties. The 'text' shoul

Copy

Error in input stream

please continue

I'll continue designing the TaleForge interactive storytelling platform. Let me pick up where I left off with the backend implementation.

**storyController.js (continued)**

exports.generateInitialSegment = async (storyId, genre) => {

// Generate story text using Llama

const prompt = `Generate the beginning of a ${genre} story with three possible choices for the reader. Format the response as a JSON object with 'text' and 'choices' properties. The 'text' should be 2-3 paragraphs introducing the setting and main character. The 'choices' should be an array of 3 strings representing different possible actions.`;

const llamaResponse = await llamaService.generateText(prompt);

const { text, choices } = JSON.parse(llamaResponse);

// Generate image for the segment

const imagePrompt = `16-bit pixel art of ${text.substring(0, 100)}`;

const imageUrl = await stableDiffusionService.generateImage(imagePrompt);

// Generate audio narration

const audioUrl = await elevenLabsService.generateSpeech(text);

// Create segment document

const segmentId = uuidv4();

const segmentData = {

id: segmentId,

storyId,

text,

imageUrl,

audioUrl,

position: 0,

parentSegmentId: null,

choices: choices.map(choiceText => ({

text: choiceText,

targetSegmentId: null // Will be generated when chosen

}))

};

await segmentsRef.doc(segmentId).set(segmentData);

return segmentData;

};

exports.generateNextSegment = async (req, res) => {

try {

const { storyId, currentSegmentId, choiceIndex } = req.body;

// Get the current segment

const segmentDoc = await segmentsRef.doc(currentSegmentId).get();

if (!segmentDoc.exists) {

return res.status(404).json({

success: false,

error: 'Segment not found'

});

}

const currentSegment = segmentDoc.data();

// Check if choice is valid

if (choiceIndex < 0 || choiceIndex >= currentSegment.choices.length) {

return res.status(400).json({

success: false,

error: 'Invalid choice index'

});

}

const selectedChoice = currentSegment.choices[choiceIndex];

// Check if this choice already has a target segment

if (selectedChoice.targetSegmentId) {

// Get existing target segment

const targetSegmentDoc = await segmentsRef.doc(selectedChoice.targetSegmentId).get();

const targetSegment = targetSegmentDoc.data();

// Update user progress

if (req.user) {

const progressQuery = await userProgressRef

.where('userId', '==', req.user.id)

.where('storyId', '==', storyId)

.limit(1)

.get();

if (!progressQuery.empty) {

const progressDoc = progressQuery.docs[0];

await progressDoc.ref.update({

currentSegmentId: targetSegment.id,

path: firebase.firestore.FieldValue.arrayUnion(targetSegment.id)

});

}

}

return res.status(200).json({

success: true,

data: targetSegment

});

}

// Generate new segment based on choice

// Get story info for context

const storyDoc = await storiesRef.doc(storyId).get();

const story = storyDoc.data();

// Get full path to provide context

let path = [currentSegment];

if (req.user) {

const progressQuery = await userProgressRef

.where('userId', '==', req.user.id)

.where('storyId', '==', storyId)

.limit(1)

.get();

if (!progressQuery.empty) {

const progressDoc = progressQuery.docs[0];

const progress = progressDoc.data();

// Get segments in path for context

if (progress.path && progress.path.length > 0) {

const pathSegments = await Promise.all(

progress.path.map(segId => segmentsRef.doc(segId).get())

);

path = pathSegments

.filter(doc => doc.exists)

.map(doc => doc.data());

}

}

}

// Build context from path

const context = path.map(seg => seg.text).join(' ');

// Generate story continuation using Llama

const prompt = `

Story genre: ${story.genre}

Story context so far:

${context}

The player chose: "${selectedChoice.text}"

Generate the next part of the story based on this choice with three new possible choices for the reader.

Format the response as a JSON object with 'text' and 'choices' properties.

The 'text' should be 2-3 paragraphs continuing the story based on the player's choice.

The 'choices' should be an array of 3 strings representing different possible actions.

`;

const llamaResponse = await llamaService.generateText(prompt);

const { text, choices } = JSON.parse(llamaResponse);

// Generate image for the new segment

const imagePrompt = `16-bit pixel art of ${text.substring(0, 100)}`;

const imageUrl = await stableDiffusionService.generateImage(imagePrompt);

// Generate audio narration

const audioUrl = await elevenLabsService.generateSpeech(text);

// Create new segment

const newSegmentId = uuidv4();

const newSegmentData = {

id: newSegmentId,

storyId,

text,

imageUrl,

audioUrl,

position: currentSegment.position + 1,

parentSegmentId: currentSegmentId,

choices: choices.map(choiceText => ({

text: choiceText,

targetSegmentId: null

}))

};

await segmentsRef.doc(newSegmentId).set(newSegmentData);

// Update the choice in the current segment

const updatedChoices = [...currentSegment.choices];

updatedChoices[choiceIndex] = {

...selectedChoice,

targetSegmentId: newSegmentId

};

await segmentsRef.doc(currentSegmentId).update({

choices: updatedChoices

});

// Update user progress

if (req.user) {

const progressQuery = await userProgressRef

.where('userId', '==', req.user.id)

.where('storyId', '==', storyId)

.limit(1)

.get();

if (!progressQuery.empty) {

const progressDoc = progressQuery.docs[0];

await progressDoc.ref.update({

currentSegmentId: newSegmentId,

path: firebase.firestore.FieldValue.arrayUnion(newSegmentId)

});

}

}

res.status(201).json({

success: true,

data: newSegmentData

});

} catch (error) {

console.error('Error generating next segment:', error);

res.status(500).json({

success: false,

error: 'Failed to generate next segment'

});

}

};

**AI Service Implementations**

**llamaService.js** This service will handle communication with Llama (via Ollama) for story generation.

const axios = require('axios');

const config = require('../config');

const OLLAMA\_API\_URL = config.OLLAMA\_API\_URL || 'http://localhost:11434/api';

exports.generateText = async (prompt) => {

try {

const response = await axios.post(`${OLLAMA\_API\_URL}/generate`, {

model: 'llama',

prompt,

options: {

temperature: 0.8,

top\_p: 0.9,

max\_tokens: 512

}

});

return response.data.response;

} catch (error) {

console.error('Error generating text with Llama:', error);

throw new Error('Failed to generate story text');

}

};

**elevenLabsService.js** This service will handle communication with the ElevenLabs API for voice narration.

const axios = require('axios');

const config = require('../config');

const { Storage } = require('@google-cloud/storage');

const ELEVENLABS\_API\_KEY = config.ELEVENLABS\_API\_KEY;

const ELEVENLABS\_API\_URL = 'https://api.elevenlabs.io/v1';

const ELEVENLABS\_VOICE\_ID = config.ELEVENLABS\_VOICE\_ID || '21m00Tcm4TlvDq8ikWAM'; // Default voice ID

// Initialize Google Cloud Storage for storing audio files

const storage = new Storage();

const bucketName = config.GCP\_STORAGE\_BUCKET;

const bucket = storage.bucket(bucketName);

exports.generateSpeech = async (text) => {

try {

const response = await axios({

method: 'POST',

url: `${ELEVENLABS\_API\_URL}/text-to-speech/${ELEVENLABS\_VOICE\_ID}`,

data: {

text,

model\_id: 'eleven\_monolingual\_v1',

voice\_settings: {

stability: 0.7,

similarity\_boost: 0.7

}

},

headers: {

'Accept': 'audio/mpeg',

'xi-api-key': ELEVENLABS\_API\_KEY,

'Content-Type': 'application/json'

},

responseType: 'arraybuffer'

});

// Generate a unique filename

const filename = `narration-${Date.now()}.mp3`;

const file = bucket.file(filename);

// Upload the audio file to Google Cloud Storage

await file.save(response.data, {

metadata: {

contentType: 'audio/mpeg'

}

});

// Make the file publicly accessible

await file.makePublic();

// Return the public URL

return `https://storage.googleapis.com/${bucketName}/${filename}`;

} catch (error) {

console.error('Error generating speech with ElevenLabs:', error);

throw new Error('Failed to generate audio narration');

}

};

**stableDiffusionService.js** This service will handle communication with Stable Diffusion for pixel-art generation.

const axios = require('axios');

const config = require('../config');

const { Storage } = require('@google-cloud/storage');

const STABLE\_DIFFUSION\_API\_URL = config.STABLE\_DIFFUSION\_API\_URL;

const STABLE\_DIFFUSION\_API\_KEY = config.STABLE\_DIFFUSION\_API\_KEY;

// Initialize Google Cloud Storage for storing image files

const storage = new Storage();

const bucketName = config.GCP\_STORAGE\_BUCKET;

const bucket = storage.bucket(bucketName);

exports.generateImage = async (prompt) => {

try {

const response = await axios({

method: 'POST',

url: STABLE\_DIFFUSION\_API\_URL,

data: {

prompt: `Pixel art, 16-bit style, ${prompt}`,

negative\_prompt: 'blurry, realistic, 3D, high resolution, photograph',

width: 512,

height: 512,

num\_inference\_steps: 30,

guidance\_scale: 7.5,

lora\_weights: {

'pixel-art': 0.7

}

},

headers: {

'Authorization': `Bearer ${STABLE\_DIFFUSION\_API\_KEY}`,

'Content-Type': 'application/json'

},

responseType: 'arraybuffer'

});

// Generate a unique filename

const filename = `pixel-art-${Date.now()}.png`;

const file = bucket.file(filename);

// Upload the image file to Google Cloud Storage

await file.save(response.data, {

metadata: {

contentType: 'image/png'

}

});

// Make the file publicly accessible

await file.makePublic();

// Return the public URL

return `https://storage.googleapis.com/${bucketName}/${filename}`;

} catch (error) {

console.error('Error generating image with Stable Diffusion:', error);

throw new Error('Failed to generate pixel art');

}

};

**5. Testing**

Let's develop a comprehensive testing strategy for TaleForge using AI-generated test cases.

**Unit Tests**

For unit testing, we'll use Jest for both frontend and backend components.

**Frontend Unit Tests (Svelte Components)**

Example test for the ChoiceButton component:

// ChoiceButton.test.js

import { render, fireEvent } from '@testing-library/svelte';

import ChoiceButton from '../src/components/story/ChoiceButton.svelte';

describe('ChoiceButton Component', () => {

test('renders with provided text', () => {

const { getByText } = render(ChoiceButton, { props: { text: 'Fight the dragon' } });

expect(getByText('Fight the dragon')).toBeInTheDocument();

});

test('calls onClick handler when clicked', async () => {

const mockClick = jest.fn();

const { getByText } = render(ChoiceButton, {

props: {

text: 'Fight the dragon',

onClick: mockClick

}

});

await fireEvent.click(getByText('Fight the dragon'));

expect(mockClick).toHaveBeenCalledTimes(1);

});

test('is disabled when disabled prop is true', () => {

const { getByText } = render(ChoiceButton, {

props: {

text: 'Fight the dragon',

disabled: true

}

});

const button = getByText('Fight the dragon');

expect(button).toBeDisabled();

});

});

**Backend Unit Tests (Controllers and Services)**

Example test for the story controller:

// storyController.test.js

const storyController = require('../src/controllers/storyController');

const llamaService = require('../src/services/llamaService');

const stableDiffusionService = require('../src/services/stableDiffusionService');

const elevenLabsService = require('../src/services/elevenLabsService');

const firebase = require('../src/services/firebase');

// Mock dependencies

jest.mock('../src/services/llamaService');

jest.mock('../src/services/stableDiffusionService');

jest.mock('../src/services/elevenLabsService');

jest.mock('../src/services/firebase');

describe('Story Controller', () => {

beforeEach(() => {

// Reset mocks

jest.clearAllMocks();

// Mock Firebase methods

firebase.firestore = jest.fn().mockReturnValue({

collection: jest.fn().mockReturnThis(),

doc: jest.fn().mockReturnThis(),

set: jest.fn().mockResolvedValue(true),

update: jest.fn().mockResolvedValue(true),

get: jest.fn().mockResolvedValue({

exists: true,

data: () => ({

id: 'test-story-id',

genre: 'fantasy',

choices: [{ text: 'Test choice', targetSegmentId: null }]

})

})

});

// Mock AI service responses

llamaService.generateText.mockResolvedValue(JSON.stringify({

text: 'This is a test story segment.',

choices: ['Choice 1', 'Choice 2', 'Choice 3']

}));

stableDiffusionService.generateImage.mockResolvedValue('https://example.com/image.png');

elevenLabsService.generateSpeech.mockResolvedValue('https://example.com/audio.mp3');

});

test('createStory creates a new story and initial segment', async () => {

const req = {

body: {

userId: 'test-user-id',

genre: 'fantasy'

}

};

const res = {

status: jest.fn().mockReturnThis(),

json: jest.fn()

};

await storyController.createStory(req, res);

expect(res.status).toHaveBeenCalledWith(201);

expect(res.json).toHaveBeenCalledWith(expect.objectContaining({

success: true,

data: expect.objectContaining({

storyId: expect.any(String),

initialSegmentId: expect.any(String)

})

}));

expect(llamaService.generateText).toHaveBeenCalled();

expect(stableDiffusionService.generateImage).toHaveBeenCalled();

expect(elevenLabsService.generateSpeech).toHaveBeenCalled();

});

});

**Integration Tests**

Integration tests will verify that different components of TaleForge work together correctly.

Example integration test for story generation flow:

// storyFlow.test.js

const request = require('supertest');

const app = require('../src/app');

const firebase = require('../src/services/firebase');

describe('Story Flow Integration', () => {

let token, storyId, segmentId;

beforeAll(async () => {

// Create test user and login

const authResponse = await request(app)

.post('/api/auth/login')

.send({

email: 'test@example.com',

password: 'testpassword'

});

token = authResponse.body.token;

});

test('Create new story', async () => {

const response = await request(app)

.post('/api/stories')

.set('Authorization', `Bearer ${token}`)

.send({

genre: 'fantasy'

});

expect(response.statusCode).toBe(201);

expect(response.body.success).toBe(true);

expect(response.body.data).toHaveProperty('storyId');

expect(response.body.data).toHaveProperty('initialSegmentId');

storyId = response.body.data.storyId;

segmentId = response.body.data.initialSegmentId;

});

test('Get story segment', async () => {

const response = await request(app)

.get(`/api/stories/${storyId}/segments/${segmentId}`)

.set('Authorization', `Bearer ${token}`);

expect(response.statusCode).toBe(200);

expect(response.body.data).toHaveProperty('text');

expect(response.body.data).toHaveProperty('imageUrl');

expect(response.body.data).toHaveProperty('audioUrl');

expect(response.body.data).toHaveProperty('choices');

expect(Array.isArray(response.body.data.choices)).toBe(true);

});

test('Make choice and generate next segment', async () => {

const response = await request(app)

.post(`/api/stories/${storyId}/segments`)

.set('Authorization', `Bearer ${token}`)

.send({

currentSegmentId: segmentId,

choiceIndex: 0

});

expect(response.statusCode).toBe(201);

expect(response.body.success).toBe(true);

expect(response.body.data).toHaveProperty('id');

expect(response.body.data).toHaveProperty('text');

expect(response.body.data).toHaveProperty('choices');

// New segment should be linked to parent

expect(response.body.data.parentSegmentId).toBe(segmentId);

});

test('Share story to community', async () => {

const response = await request(app)

.put(`/api/stories/${storyId}`)

.set('Authorization', `Bearer ${token}`)

.send({

isPublic: true,

title: 'My Epic Fantasy Adventure'

});

expect(response.statusCode).toBe(200);

expect(response.body.success).toBe(true);

});

test('Story appears in community hub', async () => {

const response = await request(app)

.get('/api/community/recent')

.set('Authorization', `Bearer ${token}`);

expect(response.statusCode).toBe(200);

expect(Array.isArray(response.body.data)).toBe(true);

const ourStory = response.body.data.find(story => story.id === storyId);

expect(ourStory).toBeDefined();

expect(ourStory.title).toBe('My Epic Fantasy Adventure');

});

});

**User Acceptance Tests (UAT)**

UAT will verify that TaleForge meets the requirements from a user's perspective.

Example UAT test cases:

1. **Story Generation Test**
   * **Scenario**: User creates a new story
   * **Steps**:
     1. Log in to TaleForge
     2. Click "Start New Story"
     3. Select "Fantasy" genre
     4. Wait for story generation
   * **Expected Result**: Initial story segment appears with text, pixel-art image, audio narration, and 3 choices
2. **Story Navigation Test**
   * **Scenario**: User makes choices to progress through a story
   * **Steps**:
     1. View initial story segment
     2. Click on a choice option
     3. Wait for next segment to generate
     4. Make a second choice
   * **Expected Result**: Each choice leads to a new story segment with updated text, image, audio, and choices that logically follow from the previous choice
3. **Story Saving Test**
   * **Scenario**: User saves story progress and resumes later
   * **Steps**:
     1. Progress through a story making at least 3 choices
     2. Click "Save Story"
     3. Log out and log back in
     4. Navigate to "My Stories"
     5. Click on saved story
   * **Expected Result**: Story resumes from the exact point where the user left off
4. **Community Sharing Test**
   * **Scenario**: User shares a story to the community
   * **Steps**:
     1. Complete a story (reach an ending)
     2. Click "Share to Community"
     3. Enter title "Dragon's Lair Adventure"
     4. Navigate to Community Hub
   * **Expected Result**: Story appears in community hub with correct title and user attribution
5. **Voting System Test**
   * **Scenario**: User votes on community stories
   * **Steps**:
     1. Navigate to Community Hub
     2. Browse stories and select one
     3. Click "Vote" button
     4. Try to vote again on the same story
   * **Expected Result**: First vote increments counter, second attempt shows message that user has already voted

**6. Documentation**

**User Manual: How to Play TaleForge**

**Welcome to TaleForge!**

TaleForge is an interactive storytelling platform where you become the hero of your own adventure. With AI-generated narratives, beautiful pixel-art visuals, and immersive narration, each choice you make shapes your unique journey. This guide will help you get started on your adventure.

**Getting Started**

1. **Creating an Account**
   * Visit [taleforge.netlify.app](https://taleforge.netlify.app)
   * Click "Sign Up" and enter your email and password
   * Alternatively, use Google or Facebook to sign in
   * You can also try TaleForge as a guest, but your progress won't be saved
2. **Starting a Story**
   * From the home screen, click "Start New Story"
   * Choose a genre:
     + **Fantasy**: Magical realms with knights, dragons, and wizards
     + **Sci-Fi**: Futuristic adventures in space or dystopian worlds
     + **Mystery**: Solve puzzles and uncover secrets
     + **Horror**: Face terrifying creatures and survive the night
   * Wait a moment as TaleForge creates your personalized adventure
3. **Playing Your Adventure**
   * Each story segment includes:
     + **Text**: The narrative description of what's happening
     + **Pixel Art**: A visual representation of the scene
     + **Audio Narration**: Voice acting for the story text
     + **Choices**: Typically 3-5 options for what to do next
   * Click on a choice to continue the story in that direction
   * Your choices affect the storyline, leading to unique outcomes
4. **Saving Your Progress**
   * Click the "Save" button at any point to store your progress
   * Find your saved stories in the "My Stories" section on the home screen
   * Each saved story shows your current progress and a thumbnail
5. **Sharing to the Community**
   * After completing a story (or at any point), click "Share"
   * Give your adventure a title and description
   * Choose whether to share the complete story or just your current progress
   * Your story will appear in the Community Hub for others to enjoy
6. **Community Hub**
   * Browse stories created by other players
   * Sort by "Popular" or "Recent" to find interesting adventures
   * Click on a story to read through someone else's journey
   * Vote for stories you enjoy to help them gain visibility

**Tips for the Best Experience**

* Use headphones for the best audio narration experience
* Try different choices on repeat playthroughs for new storylines
* Save interesting stories to your "Favorites" for inspiration
* Challenge yourself to find all possible endings in a story genre

**Troubleshooting**

* If a story segment fails to load, click the "Retry" button
* Audio not playing? Check your device's sound settings
* For persistent issues, use the "Report Bug" button or contact support@taleforge.com

**API Documentation**

**TaleForge API v1.0**

This documentation outlines the endpoints available in the TaleForge API for developers.

**Base URL**: https://api.taleforge.com/v1

**Authentication**

All API requests require authentication using a JWT token. Include it in the Authorization header:

Authorization: Bearer <your\_token>

To obtain a token, use the login endpoint.

**Endpoints**

**Authentication**

POST /auth/register

Register a new user.

Request:

{

"email": "user@example.com",

"password": "securepassword",

"displayName": "StorySeeker"

}

Response:

{

"success": true,

"data": {

"userId": "user123",

"token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9...",

"expiresIn": 3600

}

}

POST /auth/login

Log in an existing user.

Request:

{

"email": "user@example.com",

"password": "securepassword"

}

Response:

{

"success": true,

"data": {

"userId": "user123",

"token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9...",

"expiresIn": 3600

}

}

**Stories**

POST /stories

Create a new story.

Request:

{

"genre": "fantasy"

}

Response:

{

"success": true,

"data": {

"storyId": "story123",

"initialSegmentId": "segment456"

}

}

GET /stories

Get a list of your stories.

Query Parameters:

* limit: Maximum number of stories to return (default: 10)
* offset: Number of stories to skip (for pagination) (default: 0)

Response:

{

"success": true,

"data": [

{

"id": "story123",

"title": "Fantasy Adventure",

"genre": "fantasy",

"createdAt": "2024-04-15T14:32:10Z",

"updatedAt": "2024-04-15T15:10:22Z",

"thumbnailUrl": "https://storage.googleapis.com/taleforge/thumbnails/story123.png",

"isPublic": false

},

// More stories...

],

"pagination": {

"total": 24,

"limit": 10,

"offset": 0,

"hasMore": true

}

}

**Story Segments**

GET /stories/:storyId/segments/:segmentId

Get a specific story segment.

Response:

{

"success": true,

"data": {

"id": "segment456",

"storyId": "story123",

"text": "You stand before an ancient castle...",

"imageUrl": "https://storage.googleapis.com/taleforge/images/segment456.png",

"audioUrl": "https://storage.googleapis.com/taleforge/audio/segment456.mp3",

"position": 0,

"parentSegmentId": null,

"choices": [

{

"text": "Enter the castle",

"targetSegmentId": "segment789"

},

{

"text": "Explore the surrounding forest",

"targetSegmentId": null

},

{

"text": "Talk to the gatekeeper",

"targetSegmentId": null

}

]

}

}

POST /stories/:storyId/segments

Generate a new segment based on a choice.

Request:

{

"currentSegmentId": "segment456",

"choiceIndex": 1

}

Response:

{

"success": true,

"data": {

"id": "segment790",

"storyId": "story123",

"text": "You venture into the dense forest...",

"imageUrl": "https://storage.googleapis.com/taleforge/images/segment790.png",

"audioUrl": "https://storage.googleapis.com/taleforge/audio/segment790.mp3",

"position": 1,

"parentSegmentId": "segment456",

"choices": [

{

"text": "Follow the mysterious lights",

"targetSegmentId": null

},

{

"text": "Climb a tall tree to get your bearings",

"targetSegmentId": null

},

{

"text": "Return to the castle",

"targetSegmentId": null

}

]

}

}

**Community**

GET /community/popular

Get popular stories from the community.

Query Parameters:

* limit: Maximum number of stories to return (default: 10)
* offset: Number of stories to skip (for pagination) (default: 0)

Response:

{

"success": true,

"data": [

{

"id": "story789",

"title": "The Dragon's Curse",

"genre": "fantasy",

"creatorId": "user456",

"creatorName": "DragonSlayer",

"createdAt": "2024-04-10T09:22:15Z",

"votes": 42,

"thumbnailUrl": "https://storage.googleapis.com/taleforge/thumbnails/story789.png"

},

// More stories...

],

"pagination": {

"total": 156,

"limit": 10,

"offset": 0,

"hasMore": true

}

}

POST /stories/:storyId/vote

Vote for a story

Copy

please continue

I'll continue with the TaleForge project documentation and implementation plan.

**6. Documentation (Continued)**

**API Documentation (Continued)**

POST /stories/:storyId/vote

Vote for a story in the community hub.

Response:

{

"success": true,

"data": {

"storyId": "story789",

"newVoteCount": 43

}

}

DELETE /stories/:storyId/vote

Remove your vote from a story.

Response:

{

"success": true,

"data": {

"storyId": "story789",

"newVoteCount": 42

}

}

**AI Generation**

POST /generate/story

Generate a story segment based on context and genre.

Request:

{

"genre": "fantasy",

"context": "You are a brave knight who has just slain a dragon.",

"choices": ["Return to the kingdom", "Search the dragon's lair", "Rest by the lake"]

}

Response:

{

"success": true,

"data": {

"text": "With the dragon's final breath still hanging in the air...",

"choices": [

"Return to the kingdom with tales of your victory",

"Search the dragon's lair for treasure",

"Rest by the lake to recover from your wounds"

]

}

}

POST /generate/image

Generate a pixel-art image based on a description.

Request:

{

"prompt": "A knight standing victorious over a slain dragon",

"width": 512,

"height": 512

}

Response:

{

"success": true,

"data": {

"imageUrl": "https://storage.googleapis.com/taleforge/images/generated-123456.png"

}

}

POST /generate/audio

Generate audio narration from text.

Request:

{

"text": "With the dragon's final breath still hanging in the air...",

"voiceId": "voice123"

}

Response:

{

"success": true,

"data": {

"audioUrl": "https://storage.googleapis.com/taleforge/audio/narration-123456.mp3",

"duration": 12.5

}

}

**Maintenance Guide**

**TaleForge Maintenance Guide**

This guide provides instructions for maintaining and updating the TaleForge application.

**System Architecture Overview**

TaleForge consists of:

* **Frontend**: Svelte app deployed on Netlify
* **Backend**: Express.js API deployed on Vultr
* **Database**: Firebase Firestore and Storage
* **AI Services**: Llama (via Ollama), ElevenLabs, and Stable Diffusion

**Regular Maintenance Tasks**

**Weekly Maintenance**

1. Check API error logs in Vultr dashboard
2. Monitor Firebase usage and quotas
3. Verify AI service uptime and response times
4. Review user feedback and bug reports

**Monthly Maintenance**

1. Update npm dependencies for security patches
2. Optimize database queries with high execution times
3. Review and clean unused storage assets
4. Update AI prompt templates for improved story quality

**Scaling the Application**

**Handling Increased User Load**

1. Scale Vultr instances:
2. # SSH into Vultr management server
3. ssh admin@taleforge-management.example.com
4. # Scale API instances (adjust count as needed)
5. ./scale\_instances.sh --service api --count 5
6. Adjust Firebase capacity:
   * Log in to Firebase Console
   * Navigate to Project Settings > Usage & Billing
   * Upgrade plan if approaching limits

**Adding New Story Genres**

1. Update genre configuration in Firebase:
2. // Add new genre to available options
3. const db = firebase.firestore();
4. db.collection('config').doc('genres').update({
5. availableGenres: firebase.firestore.FieldValue.arrayUnion({
6. id: 'cyberpunk',
7. name: 'Cyberpunk',
8. description: 'High-tech dystopian worlds with corporate intrigue',
9. iconUrl: 'https://storage.googleapis.com/taleforge/icons/genre-cyberpunk.png'
10. })
11. });
12. Update Llama prompts for the new genre:
    * Edit /src/services/llamaService.js
    * Add new prompt templates in the genrePrompts object

**AI Service Maintenance**

**Updating Llama Prompts**

1. Connect to the Ollama server:

 ssh admin@taleforge-ollama.example.com

 Update the model configuration:

1. ollama pull llama:latest
2. # Create custom model with updated parameters
3. cd /opt/taleforge/model-config
4. ollama create taleforge-llama -f Modelfile

**Enhancing Image Generation**

1. Fine-tune the Stable Diffusion model:
2. # SSH into Stable Diffusion server
3. ssh admin@taleforge-sd.example.com
4. # Pull latest training scripts
5. cd /opt/taleforge/stable-diffusion
6. git pull origin main
7. # Run fine-tuning with new pixel art dataset
8. ./fine\_tune.sh --dataset pixel\_art\_v2 --epochs 10
9. Update API configuration to use new model:
   * Edit /src/services/stableDiffusionService.js
   * Update the MODEL\_VERSION constant

**Backup Procedures**

**Database Backup**

1. Scheduled automatic backups:
   * Firebase automatic backups are enabled
   * Additional manual backups available via Firebase Admin SDK
2. Manual backup script:
3. # Run from management server
4. cd /opt/taleforge/scripts
5. ./backup\_firebase.sh --output /backups/$(date +%Y%m%d)

**Asset Backup**

1. Storage backup for images and audio:
2. # Backup all storage assets
3. gsutil -m cp -r gs://taleforge-assets gs://taleforge-backup/$(date +%Y%m%d)

**Troubleshooting Common Issues**

**Story Generation Failures**

1. Check Llama server connection:

 # From API server

curl http://taleforge-ollama.example.com:11434/api/health

 Restart Ollama service if needed:

1. ssh admin@taleforge-ollama.example.com
2. sudo systemctl restart ollama

**Image Generation Errors**

1. Verify Stable Diffusion API status:

 # From API server

curl -H "Authorization: Bearer $SD\_API\_KEY" https://taleforge-sd.example.com/status

 Check logs for specific errors:

1. ssh admin@taleforge-sd.example.com
2. tail -100 /var/log/stable-diffusion/api.log

**Audio Generation Issues**

1. Verify ElevenLabs quota usage:
   * Check ElevenLabs dashboard at https://elevenlabs.io/dashboard
   * Ensure API key has sufficient credits
2. Alternative voice options:
   * Edit /src/config.js
   * Update FALLBACK\_VOICE\_SERVICE to use alternative provider

**7. Deployment**

Let's detail the deployment process for TaleForge, including CI/CD configuration, infrastructure setup, and verification procedures.

**Deployment Architecture**

TaleForge follows a modern deployment architecture:

1. **Frontend Deployment**
   * Svelte application hosted on Netlify
   * Automatically builds and deploys from GitHub main branch
   * Includes PWA configuration for offline capabilities
2. **Backend Deployment**
   * Express.js API deployed on Vultr cloud instances
   * Load balanced across multiple regions for reliability
   * Docker containerized for consistency and scalability
3. **Database**
   * Firebase Firestore for structured data
   * Firebase Storage for assets (images, audio)
   * Firebase Authentication for user management
4. **AI Services**
   * Llama: Self-hosted on Vultr via Ollama
   * ElevenLabs: External API service
   * Stable Diffusion: Self-hosted on GPU-enabled Vultr instance

**CI/CD Pipeline Configuration**

**GitHub Actions Workflow for Frontend**

# .github/workflows/frontend.yml

name: Frontend CI/CD

on:

push:

branches: [ main ]

paths:

- 'frontend/\*\*'

- '.github/workflows/frontend.yml'

jobs:

build-and-deploy:

runs-on: ubuntu-latest

steps:

- uses: actions/checkout@v3

- name: Setup Node.js

uses: actions/setup-node@v3

with:

node-version: '16'

cache: 'npm'

cache-dependency-path: 'frontend/package-lock.json'

- name: Install dependencies

run: |

cd frontend

npm ci

- name: Run tests

run: |

cd frontend

npm test

- name: Build

run: |

cd frontend

npm run build

env:

VITE\_API\_URL: ${{ secrets.PROD\_API\_URL }}

VITE\_FIREBASE\_CONFIG: ${{ secrets.FIREBASE\_CONFIG }}

- name: Deploy to Netlify

uses: netlify/actions/cli@master

with:

args: deploy --dir=frontend/dist --prod

env:

NETLIFY\_SITE\_ID: ${{ secrets.NETLIFY\_SITE\_ID }}

NETLIFY\_AUTH\_TOKEN: ${{ secrets.NETLIFY\_AUTH\_TOKEN }}

**GitHub Actions Workflow for Backend**

# .github/workflows/backend.yml

name: Backend CI/CD

on:

push:

branches: [ main ]

paths:

- 'backend/\*\*'

- '.github/workflows/backend.yml'

jobs:

test:

runs-on: ubuntu-latest

steps:

- uses: actions/checkout@v3

- name: Setup Node.js

uses: actions/setup-node@v3

with:

node-version: '16'

cache: 'npm'

cache-dependency-path: 'backend/package-lock.json'

- name: Install dependencies

run: |

cd backend

npm ci

- name: Run tests

run: |

cd backend

npm test

deploy:

needs: test

runs-on: ubuntu-latest

steps:

- uses: actions/checkout@v3

- name: Build and push Docker image

uses: docker/build-push-action@v2

with:

context: ./backend

push: true

tags: registry.vultr.com/taleforge/api:latest

username: ${{ secrets.VULTR\_REGISTRY\_USERNAME }}

password: ${{ secrets.VULTR\_REGISTRY\_PASSWORD }}

- name: Deploy to Vultr

uses: appleboy/ssh-action@master

with:

host: ${{ secrets.VULTR\_HOST }}

username: ${{ secrets.VULTR\_USERNAME }}

key: ${{ secrets.VULTR\_SSH\_KEY }}

script: |

cd /opt/taleforge/deployment

docker-compose pull api

docker-compose up -d api

docker system prune -f

**Deployment Scripts**

**Frontend Deployment Script (Netlify)**

#!/bin/bash

# deploy\_frontend.sh

# Build the Svelte app

echo "Building Svelte app..."

cd frontend

npm ci

npm run build

# Deploy to Netlify

echo "Deploying to Netlify..."

npx netlify-cli deploy --dir=dist --prod

echo "Frontend deployment complete!"

**Backend Deployment Script (Vultr)**

#!/bin/bash

# deploy\_backend.sh

# Build Docker image

echo "Building Docker image..."

cd backend

docker build -t taleforge-api:latest .

# Push to container registry

echo "Pushing to container registry..."

docker tag taleforge-api:latest registry.vultr.com/taleforge/api:latest

docker push registry.vultr.com/taleforge/api:latest

# SSH into Vultr instance and update

echo "Deploying to Vultr..."

ssh admin@api.taleforge.com << 'EOF'

cd /opt/taleforge/deployment

docker-compose pull api

docker-compose up -d api

docker system prune -f

EOF

echo "Backend deployment complete!"

**Firebase Configuration**

**Firebase Setup Script**

#!/bin/bash

# setup\_firebase.sh

# Initialize Firebase project

firebase login

firebase init

# Set up Firestore security rules

cat > firestore.rules << EOF

rules\_version = '2';

service cloud.firestore {

match /databases/{database}/documents {

// User profiles

match /users/{userId} {

allow read: if request.auth != null;

allow write: if request.auth != null && request.auth.uid == userId;

}

// Stories

match /stories/{storyId} {

allow read: if resource.data.isPublic == true ||

(request.auth != null && resource.data.creatorId == request.auth.uid);

allow create: if request.auth != null;

allow update, delete: if request.auth != null && resource.data.creatorId == request.auth.uid;

}

// Story segments

match /storySegments/{segmentId} {

allow read: if true;

allow write: if request.auth != null;

}

// Votes

match /votes/{voteId} {

allow read: if true;

allow create: if request.auth != null;

allow delete: if request.auth != null && resource.data.userId == request.auth.uid;

}

}

}

EOF

# Set up storage rules

cat > storage.rules << EOF

rules\_version = '2';

service firebase.storage {

match /b/{bucket}/o {

match /images/{imageId} {

allow read: if true;

allow write: if request.auth != null;

}

match /audio/{audioId} {

allow read: if true;

allow write: if request.auth != null;

}

}

}

EOF

# Deploy Firebase configuration

firebase deploy --only firestore:rules,storage:rules

**Live Deployment URLs**

* **Frontend**: <https://taleforge.netlify.app>
* **Backend API**: <https://api.taleforge.com>
* **Documentation**: <https://docs.taleforge.com>

**Post-Deployment Verification Checklist**

1. **Frontend Verification**:
   * Homepage loads correctly with proper styling
   * Authentication flow works (signup, login, logout)
   * Story creation starts successfully
   * Story playback includes text, images, and audio
   * Community hub displays shared stories
2. **Backend Verification**:
   * All API endpoints respond with correct status codes
   * Authentication middleware properly secures protected routes
   * Firebase connections are established successfully
   * AI services (Llama, ElevenLabs, Stable Diffusion) are accessible
3. **Integration Verification**:
   * Complete end-to-end flow works: create story → make choices → share story
   * Real-time updates appear in community hub
   * Votes are properly counted and displayed
4. **Performance Verification**:
   * Frontend loads in under 3 seconds
   * Story generation completes in under 5 seconds
   * API responds to requests within 500ms (excluding AI generation time)

**GitHub Repository Structure**

taleforge/

├── README.md # Project overview and setup instructions

├── \_planning.md # Planning assets and documentation

├── \_prompts.pdf # Log of AI prompts used during development

├── frontend/ # Svelte application

│ ├── public/ # Static assets and manifest.json for PWA

│ ├── src/ # Source code

│ │ ├── components/ # Svelte components

│ │ ├── pages/ # Page components

│ │ ├── services/ # API and service integrations

│ │ ├── stores/ # Svelte stores for state management

│ │ ├── utils/ # Utility functions

│ │ └── App.svelte # Main application component

│ ├── tests/ # Unit and integration tests

│ ├── package.json # Frontend dependencies

│ └── vite.config.js # Build configuration

├── backend/ # Express.js API

│ ├── src/ # Source code

│ │ ├── controllers/ # API route controllers

│ │ ├── middleware/ # Express middleware

│ │ ├── routes/ # API route definitions

│ │ ├── services/ # Service integrations (Firebase, AI)

│ │ └── app.js # Main application setup

│ ├── tests/ # Unit and integration tests

│ ├── package.json # Backend dependencies

│ └── Dockerfile # Container configuration

├── docs/ # Documentation

│ ├── api/ # API documentation

│ ├── user-manual/ # User guide

│ └── maintenance/ # Maintenance guide

└── .github/ # GitHub configuration

└── workflows/ # CI/CD workflows

**README.md Content**

# TaleForge

TaleForge is an AI-driven interactive storytelling platform that creates dynamic, branching narratives based on your choices. With beautiful pixel-art visuals and immersive narration, each playthrough offers a unique adventure.

![TaleForge Demo](https://storage.googleapis.com/taleforge-assets/demo.gif)

## Features

- \*\*AI-Generated Stories\*\*: Each adventure is uniquely crafted based on your choices

- \*\*Pixel-Art Visuals\*\*: Beautiful retro-style artwork brings your story to life

- \*\*Voice Narration\*\*: Immersive audio narration enhances the storytelling experience

- \*\*Community Hub\*\*: Share your adventures and discover stories from other players

- \*\*Cross-Platform\*\*: Play on desktop or mobile with Progressive Web App support

## Live Demo

Experience TaleForge now at [taleforge.netlify.app](https://taleforge.netlify.app)

## Tech Stack

### Frontend

- Svelte - Reactive JavaScript framework

- Web Audio API - Audio playback management

- PWA - Progressive Web App features

### Backend

- Express.js - Node.js web framework

- Firebase - Real-time database and authentication

- AI Services

- Llama (via Ollama) - Story generation

- ElevenLabs - Voice narration

- Stable Diffusion - Pixel-art generation

## Getting Started

### Prerequisites

- Node.js 16+

- Firebase account and project

- API keys for ElevenLabs and Stable Diffusion

- Ollama setup with Llama model

### Local Development

1. \*\*Clone the repository\*\*

```bash

git clone https://github.com/yourusername/taleforge.git

cd taleforge

1. **Set up environment variables**

Create .env files for both frontend and backend:

# frontend/.env

VITE\_API\_URL=http://localhost:3000/api

VITE\_FIREBASE\_CONFIG={"apiKey":"...","authDomain":"..."}

# backend/.env

PORT=3000

FIREBASE\_SERVICE\_ACCOUNT={"type":"service\_account","project\_id":"..."}

ELEVENLABS\_API\_KEY=your\_elevenlabs\_api\_key

STABLE\_DIFFUSION\_API\_URL=your\_sd\_api\_url

STABLE\_DIFFUSION\_API\_KEY=your\_sd\_api\_key

OLLAMA\_API\_URL=http://localhost:11434/api

1. **Install dependencies and start development servers**

# Frontend

cd frontend

npm install

npm run dev

# Backend

cd backend

npm install

npm run dev

1. Open <http://localhost:5173> to view the application

**Deployment**

TaleForge is deployed using:

* Frontend: Netlify
* Backend: Vultr (Docker containerized)
* Database: Firebase (Firestore + Storage)

See the [deployment guide](https://www.genspark.ai/docs/deployment.md) for detailed instructions.

**Screenshots**

**Contributing**

Contributions are welcome! Please feel free to submit a Pull Request.

**License**

This project is licensed under the MIT License - see the LICENSE file for details.

**Acknowledgments**

* [ElevenLabs](https://elevenlabs.io/) for the voice technology
* [Stable Diffusion](https://stability.ai/) for the image generation
* [Llama](https://github.com/facebookresearch/llama) for the story generation model

## Monetization Strategy

TaleForge's monetization strategy follows a freemium model with tiered subscription options and targeted in-app purchases.

### Freemium Tiers

\*\*Free Tier:\*\*

- Access to basic story genres (Fantasy and Mystery)

- Standard voice narration

- Limited to 5 stories per month

- Basic pixel-art quality

- Community hub access (browsing only)

\*\*Premium Tier ($3.99/month):\*\*

- Access to all story genres (including Sci-Fi, Horror, Western, etc.)

- Premium voice narration with multiple character voices

- Unlimited stories

- Enhanced pixel-art quality

- Community hub access (sharing and voting)

- Early access to new features

- Ad-free experience

\*\*Professional Tier ($9.99/month):\*\*

- All Premium features

- Story export capabilities (PDF, audio book)

- Custom prompt engineering for personalized stories

- Access to AI editing tools

- Analytics for shared stories

- Priority support

### In-App Purchases

\*\*Cosmetic Upgrades:\*\*

- Pixel-Art Themes ($0.99 each)

- Cyberpunk Theme

- Medieval Theme

- Space Opera Theme

- Horror Theme

- Steampunk Theme

\*\*Voice Packs:\*\*

- Character Voice Collections ($1.99 each)

- Fantasy Heroes Pack (knight, wizard, rogue)

- Sci-Fi Explorers Pack (captain, android, alien)

- Mythical Creatures Pack (dragon, fairy, troll)

\*\*Story Boosters:\*\*

- Extended Story Length ($0.99)

- Multiple Endings Expansion ($1.49)

- Crossover Characters ($0.99)

### Special Offers

\*\*Annual Subscription:\*\*

- Premium Annual: $36.99/year (save 23%)

- Professional Annual: $89.99/year (save 25%)

\*\*Bundles:\*\*

- Starter Bundle ($4.99): 3 themes + 1 voice pack

- Creator Bundle ($9.99): All themes + 2 voice packs + 2 story boosters

## What Makes TaleForge Stand Out

TaleForge distinguishes itself through several key innovations:

### 1. AI-Driven Integration Across All Components

Unlike competitors that use AI for isolated features, TaleForge integrates multiple AI technologies (narrative generation, voice synthesis, and image creation) into a cohesive, harmonious experience. The multi-modal approach creates a more immersive storytelling environment than text-only alternatives.

### 2. Infinite Replayability Through Dynamic Branching

The Llama-powered story generation engine remembers previous choices and adapts future story elements accordingly. This creates truly divergent narratives rather than the limited, pre-written branches found in many interactive fiction apps. Each playthrough feels genuinely different, encouraging repeated engagement.

### 3. Pixel-Art Aesthetics with Advanced AI

By fine-tuning Stable Diffusion specifically for pixel-art generation, TaleForge creates visually consistent, nostalgic artwork that stands out from the generic AI art prevalent in other applications. This distinctive visual identity creates strong brand recognition and appeals to both retro gaming enthusiasts and new audiences.

### 4. Comprehensive Community Features

The community hub transforms TaleForge from a solo experience into a social platform. By allowing users to share, discover, and vote on stories, the application builds an engaged community that continuously generates fresh content and provides valuable feedback for future development.

### 5. Cross-Platform Accessibility with PWA Technology

As a Progressive Web App, TaleForge offers the convenience of web access with the functionality of a native application. This approach maximizes accessibility across devices while minimizing development resources compared to maintaining separate native applications for different platforms.

### 6. Educational Value

Beyond entertainment, TaleForge serves as a compelling educational tool for creative writing and narrative structure. The branching stories can help users understand cause-and-effect relationships in storytelling, making it valuable for educational settings and aspiring writers.

## Conclusion

TaleForge represents a cutting-edge integration of multiple AI technologies to create an immersive, infinitely replayable storytelling experience. By following this comprehensive development plan across all SDLC phases, the project will result in a polished, production-ready application that showcases the power of AI-driven development.

The combination of Svelte's reactive frontend, Express.js backend, Firebase real-time database, and multiple AI services (Llama, ElevenLabs, and Stable Diffusion) creates a technically sophisticated yet user-friendly platform. The pixel-art aesthetic provides a distinctive visual identity, while PWA capabilities ensure broad accessibility across devices.

With its freemium monetization model and strong community features, TaleForge has potential not just as a portfolio piece but as a viable commercial product with multiple revenue streams. The modular architecture also allows for future expansion, such as additional genres, collaborative storytelling features, or educational applications.

By documenting the development process thoroughly, including AI prompts used at each stage, this project will serve as an excellent demonstration of AI-assisted development practices and multi-modal AI integration in a consumer-facing application.

Copy

please continue

**Continuing TaleForge Development Plan**

**Implementation Deep Dive**

Let's explore some of the more complex implementation aspects of TaleForge that will make it stand out as a flagship portfolio piece.

**Advanced AI Storytelling Engine**

The core of TaleForge is its sophisticated narrative generation system. Let's examine how we'll implement this using Llama/Ollama:

**Story Context Management**

One of the biggest challenges in AI storytelling is maintaining coherence over multiple decision points. We'll solve this with a specialized context management system:

// src/services/storyContextManager.js

class StoryContextManager {

constructor(storyId, genre) {

this.storyId = storyId;

this.genre = genre;

this.segments = [];

this.characters = new Map();

this.locations = new Map();

this.plotPoints = [];

this.thematicElements = [];

}

addSegment(segment) {

this.segments.push({

text: segment.text,

choice: segment.selectedChoice,

position: segment.position

});

// Extract and track entities from the new segment

this.\_extractEntities(segment.text);

// Maintain a reasonable context window

if (this.segments.length > 10) {

// Keep important segments: first, last few, and key plot points

const keySegments = [

this.segments[0],

...this.segments.slice(-5)

];

// Find segments with high plot importance

const plotSegments = this.segments.filter(s => s.isPlotPoint);

this.segments = [...keySegments, ...plotSegments];

}

}

\_extractEntities(text) {

// Use NLP techniques to extract characters, locations, etc.

// This could be enhanced with a secondary AI call to identify entities

// Example implementation using simple pattern matching

const characterMatches = text.match(/([A-Z][a-z]+)(?:\s[A-Z][a-z]+)?\s(?:said|asked|exclaimed|whispered|replied|shouted)/g);

if (characterMatches) {

characterMatches.forEach(match => {

const character = match.split(' ')[0];

if (!this.characters.has(character)) {

this.characters.set(character, {

name: character,

mentions: 1,

firstAppearance: this.segments.length

});

} else {

const char = this.characters.get(character);

char.mentions++;

this.characters.set(character, char);

}

});

}

// Similar processing for locations, items, etc.

}

generatePrompt() {

// Create a rich prompt for the LLM that includes relevant context

// Start with basic info

let prompt = `

You are generating an interactive story in the ${this.genre} genre.

Key characters in this story:

${[...this.characters.entries()]

.filter(([\_, char]) => char.mentions > 1)

.map(([name, char]) => `- ${name}`)

.join('\n')}

Previously in the story:

${this.segments.slice(-3).map(s => s.text).join('\n\n')}

The player chose: "${this.segments.slice(-1)[0]?.choice || 'Start the story'}"

Generate the next part of the story with three compelling choices that branch the narrative in different directions. Format the response as a JSON object with "text" and "choices" properties. The "text" should be 2-3 paragraphs continuing the story based on the player's choice. The "choices" should be an array of 3 strings representing different possible actions.

Maintain the following storytelling elements:

1. Consistent character personalities and motivations

2. Logical consequences for player actions

3. Gradual progression toward a meaningful resolution

4. Appropriate pacing for the ${this.genre} genre

5. Vivid sensory descriptions suitable for visual representation

`;

return prompt;

}

}

module.exports = StoryContextManager;

This context management system will help TaleForge maintain story coherence across multiple choice points, addressing one of the key weaknesses in current AI storytelling applications.

**Voice Narration with Character Recognition**

For a more immersive experience, we'll implement advanced narration that assigns different voices to different characters:

// src/services/characterVoiceManager.js

const elevenLabsService = require('./elevenLabsService');

const nlp = require('compromise');

class CharacterVoiceManager {

constructor() {

this.voiceMappings = new Map();

this.availableVoices = [

{ id: 'voice1', name: 'Narrator', gender: 'neutral' },

{ id: 'voice2', name: 'Young Male', gender: 'male' },

{ id: 'voice3', name: 'Young Female', gender: 'female' },

{ id: 'voice4', name: 'Elder Male', gender: 'male' },

{ id: 'voice5', name: 'Elder Female', gender: 'female' },

{ id: 'voice6', name: 'Mysterious', gender: 'neutral' },

{ id: 'voice7', name: 'Gruff', gender: 'male' },

{ id: 'voice8', name: 'Cheerful', gender: 'female' }

];

}

assignVoiceToCharacter(character, preferredGender = null) {

// If character already has a voice, return it

if (this.voiceMappings.has(character)) {

return this.voiceMappings.get(character);

}

// Find available voices that match preferred gender or are neutral

const matchingVoices = this.availableVoices.filter(v =>

!Array.from(this.voiceMappings.values()).includes(v.id) &&

(v.gender === preferredGender || v.gender === 'neutral' || !preferredGender)

);

// If we have matching voices, assign one

if (matchingVoices.length > 0) {

const voice = matchingVoices[0];

this.voiceMappings.set(character, voice.id);

return voice.id;

}

// If no specific voice is available, reuse one based on gender

const fallbackVoices = this.availableVoices.filter(v =>

v.gender === preferredGender || v.gender === 'neutral'

);

if (fallbackVoices.length > 0) {

const voice = fallbackVoices[0];

this.voiceMappings.set(character, voice.id);

return voice.id;

}

// Last resort: use narrator voice

this.voiceMappings.set(character, 'voice1');

return 'voice1';

}

async generateNarration(text) {

// Extract dialogue and speakers

const doc = nlp(text);

const quotes = doc.quotations().out('array');

const sentences = doc.sentences().out('array');

// Prepare text segments for different voices

const segments = [];

let remainingText = text;

// Process dialogue with character voices

for (const quote of quotes) {

const beforeQuote = remainingText.split(quote)[0];

const speakerMatch = beforeQuote.match(/([A-Z][a-z]+)(?:\s[A-Z][a-z]+)?\s(?:said|asked|exclaimed|whispered|replied|shouted)/);

if (speakerMatch) {

const speaker = speakerMatch[1];

const speakerVoice = this.assignVoiceToCharacter(speaker);

// Add narration before the quote

if (beforeQuote.trim()) {

segments.push({

text: beforeQuote.trim(),

voiceId: 'voice1' // Narrator voice

});

}

// Add the quote with character's voice

segments.push({

text: quote,

voiceId: speakerVoice

});

// Update remaining text

remainingText = remainingText.replace(beforeQuote + quote, '');

}

}

// Add any remaining narration

if (remainingText.trim()) {

segments.push({

text: remainingText.trim(),

voiceId: 'voice1' // Narrator voice

});

}

// Generate audio for each segment

const audioPromises = segments.map(async segment => {

const audioUrl = await elevenLabsService.generateSpeech(segment.text, segment.voiceId);

return {

text: segment.text,

audioUrl,

voiceId: segment.voiceId

};

});

const audioSegments = await Promise.all(audioPromises);

return {

segments: audioSegments,

transcript: text,

characterVoices: Object.fromEntries(this.voiceMappings)

};

}

}

module.exports = new CharacterVoiceManager();

This voice manager will create a more engaging storytelling experience by assigning different voices to different characters, enhancing immersion.

**Adaptive Pixel Art Generation**

To ensure our pixel art visuals are consistently high quality and match the story context, we'll implement a specialized image generation service:

// src/services/pixelArtGenerator.js

const stableDiffusionService = require('./stableDiffusionService');

const imageAnalyzer = require('./imageAnalyzer');

const cache = require('./cache');

class PixelArtGenerator {

constructor() {

this.stylePresets = {

fantasy: {

style: '16-bit pixel art fantasy style, SNES RPG',

negativePrompt: 'blurry, modern, realistic, 3D, photograph, high-resolution'

},

scifi: {

style: '16-bit pixel art sci-fi style, retro futurism',

negativePrompt: 'blurry, fantasy, medieval, swords, dragons, realistic'

},

horror: {

style: '16-bit pixel art horror style, dark pixel art',

negativePrompt: 'blurry, cheerful, bright colors, cute, friendly'

},

mystery: {

style: '16-bit pixel art noir style, detective pixel art',

negativePrompt: 'blurry, colorful, fantasy, sci-fi, spaceships'

}

};

this.cachedImages = new Map();

}

extractImagePrompt(text) {

// Extract key visual elements from text

// This could be enhanced with NLP or a separate AI call

// Example implementation using key phrase extraction

const sentences = text.split(/[.!?]+/);

// Look for descriptive sentences with visual elements

const visualSentences = sentences.filter(s => {

return (s.includes(" saw ") ||

s.includes(" looked ") ||

s.includes(" appeared ") ||

s.includes(" noticed ") ||

s.toLowerCase().includes("scene") ||

s.toLowerCase().includes("view"))

});

if (visualSentences.length > 0) {

return visualSentences[0].trim();

}

// Fallback to first sentence as context

return sentences[0].trim();

}

async generateImage(text, genre) {

const visualContext = this.extractImagePrompt(text);

const stylePreset = this.stylePresets[genre.toLowerCase()] || this.stylePresets.fantasy;

// Check cache first

const cacheKey = `${genre}-${visualContext.substring(0, 50)}`;

if (cache.has(cacheKey)) {

return cache.get(cacheKey);

}

// Construct prompt for Stable Diffusion

const prompt = `${stylePreset.style}, ${visualContext}`;

// Generate image

const imageUrl = await stableDiffusionService.generateImage(

prompt,

stylePreset.negativePrompt,

512, // width

512, // height

{

seed: Math.floor(Math.random() \* 1000000),

steps: 30,

guidance\_scale: 7.5,

sampler: 'k\_euler\_a'

}

);

// Analyze image quality

const imageQuality = await imageAnalyzer.checkQuality(imageUrl);

// If quality is poor, try once more with adjusted prompt

if (imageQuality < 0.7) {

console.log('Low quality image detected, regenerating...');

const enhancedPrompt = `${stylePreset.style}, detailed scene showing ${visualContext}`;

const regeneratedImageUrl = await stableDiffusionService.generateImage(

enhancedPrompt,

stylePreset.negativePrompt,

512,

512,

{

seed: Math.floor(Math.random() \* 1000000),

steps: 40,

guidance\_scale: 8.0,

sampler: 'k\_euler\_a'

}

);

// Cache the successful image

cache.set(cacheKey, regeneratedImageUrl);

return regeneratedImageUrl;

}

// Cache the successful image

cache.set(cacheKey, imageUrl);

return imageUrl;

}

}

module.exports = new PixelArtGenerator();

This service ensures that our pixel art visually represents the story content while maintaining a consistent aesthetic style.

**Progressive Web App Implementation**

To ensure TaleForge works well across devices and provides offline capabilities, we'll implement robust PWA features:

// frontend/src/service-worker.js

import { build, files, timestamp } from '$service-worker';

const CACHE\_NAME = `taleforge-cache-${timestamp}`;

const ASSETS = [...build, ...files];

// Install event - cache core assets

self.addEventListener('install', event => {

event.waitUntil(

caches.open(CACHE\_NAME)

.then(cache => cache.addAll(ASSETS))

.then(() => self.skipWaiting())

);

});

// Activate event - clean up old caches

self.addEventListener('activate', event => {

event.waitUntil(

caches.keys().then(keys => {

return Promise.all(

keys.filter(key => key !== CACHE\_NAME)

.map(key => caches.delete(key))

);

}).then(() => self.clients.claim())

);

});

// Fetch event - serve from cache, fallback to network

self.addEventListener('fetch', event => {

// Skip non-GET requests

if (event.request.method !== 'GET') return;

// Skip some URLs that should always come from network

const url = new URL(event.request.url);

if (url.pathname.startsWith('/api/')) {

// For API calls, use network first, fallback to saved responses

event.respondWith(

fetch(event.request)

.then(response => {

// Cache successful API responses for offline use

const clone = response.clone();

caches.open(CACHE\_NAME + '-api')

.then(cache => cache.put(event.request, clone));

return response;

})

.catch(() => {

return caches.match(event.request);

})

);

return;

}

// For assets and other content, use cache first strategy

event.respondWith(

caches.match(event.request)

.then(response => {

if (response) {

return response;

}

// If not in cache, fetch from network

return fetch(event.request).then(networkResponse => {

// Cache a copy of the response

const clone = networkResponse.clone();

caches.open(CACHE\_NAME)

.then(cache => cache.put(event.request, clone));

return networkResponse;

});

})

);

});

// Background sync for offline operations

self.addEventListener('sync', event => {

if (event.tag === 'story-save') {

event.waitUntil(syncSavedStories());

} else if (event.tag === 'story-vote') {

event.waitUntil(syncVotes());

}

});

// Push notification handling for community activity

self.addEventListener('push', event => {

const data = event.data.json();

const options = {

body: data.body,

icon: '/icons/notification-icon-192.png',

badge: '/icons/notification-badge-96.png',

data: {

storyId: data.storyId

}

};

event.waitUntil(

self.registration.showNotification(data.title, options)

);

});

// Notification click handler

self.addEventListener('notificationclick', event => {

event.notification.close();

const storyId = event.notification.data.storyId;

event.waitUntil(

clients.openWindow(`/story/${storyId}`)

);

});

// Helper functions for sync operations

async function syncSavedStories() {

// Logic to sync stories saved while offline

const db = await openIndexedDB();

const offlineStories = await db.getAll('offlineStories');

for (const story of offlineStories) {

try {

const response = await fetch('/api/stories/sync', {

method: 'POST',

headers: {

'Content-Type': 'application/json'

},

body: JSON.stringify(story)

});

if (response.ok) {

await db.delete('offlineStories', story.id);

}

} catch (error) {

console.error('Failed to sync story:', error);

// Will retry on next sync event

}

}

}

async function syncVotes() {

// Logic to sync votes made while offline

const db = await openIndexedDB();

const offlineVotes = await db.getAll('offlineVotes');

for (const vote of offlineVotes) {

try {

const response = await fetch('/api/stories/${vote.storyId}/vote', {

method: 'POST',

headers: {

'Content-Type': 'application/json'

}

});

if (response.ok) {

await db.delete('offlineVotes', vote.id);

}

} catch (error) {

console.error('Failed to sync vote:', error);

// Will retry on next sync event

}

}

}

// IndexedDB helper for offline data

function openIndexedDB() {

return new Promise((resolve, reject) => {

const request = indexedDB.open('TaleForgeOfflineDB', 1);

request.onupgradeneeded = event => {

const db = event.target.result;

// Create stores for offline data

db.createObjectStore('offlineStories', { keyPath: 'id' });

db.createObjectStore('offlineVotes', { keyPath: 'id' });

};

request.onerror = () => reject(request.error);

request.onsuccess = () => {

const db = request.result;

// Wrap with simplified methods

resolve({

getAll: (storeName) => {

return new Promise((resolve, reject) => {

const transaction = db.transaction(storeName, 'readonly');

const store = transaction.objectStore(storeName);

const request = store.getAll();

request.onsuccess = () => resolve(request.result);

request.onerror = () => reject(request.error);

});

},

delete: (storeName, id) => {

return new Promise((resolve, reject) => {

const transaction = db.transaction(storeName, 'readwrite');

const store = transaction.objectStore(storeName);

const request = store.delete(id);

request.onsuccess = () => resolve();

request.onerror = () => reject(request.error);

});

}

});

};

});

}

**Real-time Community Features**

To create an engaging community experience, we'll implement real-time updates using Firebase and WebSockets:

// frontend/src/services/communityService.js

import firebase from './firebase';

import { writable, derived } from 'svelte/store';

import { user } from '../stores/user';

// Store for community stories

export const communityStories = writable([]);

export const storyVotes = writable({});

export const popularTags = writable([]);

export const isLoading = writable(false);

// Derived store for stories filtered by selected tags

export const selectedTags = writable([]);

export const filteredStories = derived(

[communityStories, selectedTags],

([$communityStories, $selectedTags]) => {

if ($selectedTags.length === 0) return $communityStories;

return $communityStories.filter(story =>

story.tags && story.tags.some(tag => $selectedTags.includes(tag))

);

}

);

// References to Firebase collections

const storiesRef = firebase.firestore().collection('stories');

const votesRef = firebase.firestore().collection('votes');

// Initialize listener for community stories

let unsubscribeStories = null;

export function initCommunityHub() {

isLoading.set(true);

// Unsubscribe from previous listener if exists

if (unsubscribeStories) {

unsubscribeStories();

}

// Listen for public stories with real-time updates

unsubscribeStories = storiesRef

.where('isPublic', '==', true)

.orderBy('updatedAt', 'desc')

.limit(20)

.onSnapshot(snapshot => {

const stories = [];

snapshot.forEach(doc => {

const storyData = doc.data();

stories.push({

id: doc.id,

...storyData

});

});

communityStories.set(stories);

isLoading.set(false);

// Extract and count tags

const allTags = stories.flatMap(story => story.tags || []);

const tagCounts = allTags.reduce((acc, tag) => {

acc[tag] = (acc[tag] || 0) + 1;

return acc;

}, {});

const topTags = Object.entries(tagCounts)

.sort((a, b) => b[1] - a[1])

.slice(0, 10)

.map(([tag]) => tag);

popularTags.set(topTags);

// Load votes for these stories

loadVotesForStories(stories.map(s => s.id));

});

}

// Load votes for a set of stories

export function loadVotesForStories(storyIds) {

votesRef

.where('storyId', 'in', storyIds)

.get()

.then(snapshot => {

const votes = {};

storyIds.forEach(id => {

votes[id] = {

count: 0,

userVoted: false

};

});

snapshot.forEach(doc => {

const voteData = doc.data();

if (!votes[voteData.storyId]) {

votes[voteData.storyId] = {

count: 0,

userVoted: false

};

}

votes[voteData.storyId].count++;

if (voteData.userId === firebase.auth().currentUser?.uid) {

votes[voteData.storyId].userVoted = true;

}

});

storyVotes.set(votes);

});

}

// Vote for a story

export async function voteForStory(storyId) {

const userId = firebase.auth().currentUser?.uid;

if (!userId) {

throw new Error('Must be logged in to vote');

}

// Check if user already voted

const existingVote = await votesRef

.where('storyId', '==', storyId)

.where('userId', '==', userId)

.get();

if (!existingVote.empty) {

throw new Error('Already voted for this story');

}

// Create vote document

await votesRef.add({

storyId,

userId,

timestamp: firebase.firestore.FieldValue.serverTimestamp()

});

// Update story vote count

await storiesRef.doc(storyId).update({

votes: firebase.firestore.FieldValue.increment(1)

});

// Update local state

storyVotes.update(votes => {

if (!votes[storyId]) {

votes[storyId] = { count: 1, userVoted: true };

} else {

votes[storyId].count++;

votes[storyId].userVoted = true;

}

return votes;

});

}

// Remove vote from story

export async function unvoteStory(storyId) {

const userId = firebase.auth().currentUser?.uid;

if (!userId) {

throw new Error('Must be logged in to unvote');

}

// Find and delete vote

const voteQuery = await votesRef

.where('storyId', '==', storyId)

.where('userId', '==', userId)

.get();

if (voteQuery.empty) {

throw new Error('No vote found to remove');

}

// Delete the vote document

await voteQuery.docs[0].ref.delete();

// Update story vote count

await storiesRef.doc(storyId).update({

votes: firebase.firestore.FieldValue.increment(-1)

});

// Update local state

storyVotes.update(votes => {

if (votes[storyId]) {

votes[storyId].count = Math.max(0, votes[storyId].count - 1);

votes[storyId].userVoted = false;

}

return votes;

});

}

// Share a story to community

export async function shareStory(storyId, { title, description, tags = [] }) {

const userId = firebase.auth().currentUser?.uid;

if (!userId) {

throw new Error('Must be logged in to share a story');

}

// Update story with community details and set to public

await storiesRef.doc(storyId).update({

title,

description,

tags,

isPublic: true,

sharedAt: firebase.firestore.FieldValue.serverTimestamp()

});

return storyId;

}

// Cleanup function

export function cleanup() {

if (unsubscribeStories) {

unsubscribeStories();

unsubscribeStories = null;

}

}

**User Experience Enhancements**

**Adaptive Difficulty System**

To keep users engaged, we'll implement an adaptive difficulty system that adjusts story complexity based on user behavior:

// src/services/adaptiveDifficultyManager.js

class AdaptiveDifficultyManager {

constructor(userId) {

this.userId = userId;

this.metrics = {

averageReadingTime: 0,

choiceConsistency: 0.5, // 0-1 scale of how consistent user choices are

preferredComplexity: 0.5, // 0-1 scale of preferred story complexity

genrePreferences: {},

sessionDuration: 0,

completionRate: 0

};

this.sessionData = {

startTime: Date.now(),

segmentsRead: 0,

totalReadingTime: 0,

choiceHistory: []

};

this.loadUserMetrics();

}

async loadUserMetrics() {

try {

// Load metrics from Firebase or create default if not exists

const userMetrics = await firebase.firestore()

.collection('userMetrics')

.doc(this.userId)

.get();

if (userMetrics.exists) {

this.metrics = userMetrics.data();

} else {

// Create default metrics

await firebase.firestore()

.collection('userMetrics')

.doc(this.userId)

.set(this.metrics);

}

} catch (error) {

console.error('Error loading user metrics:', error);

}

}

recordSegmentRead(segmentId, readingTimeMs) {

this.sessionData.segmentsRead++;

this.sessionData.totalReadingTime += readingTimeMs;

// Update average reading time metric

const newAverage =

(this.metrics.averageReadingTime \* (this.sessionData.segmentsRead - 1) + readingTimeMs) /

this.sessionData.segmentsRead;

this.metrics.averageReadingTime = newAverage;

}

recordChoice(choiceIndex, choiceOptions) {

this.sessionData.choiceHistory.push({

timestamp: Date.now(),

choiceIndex,

numOptions: choiceOptions.length

});

// Analyze choice patterns

if (this.sessionData.choiceHistory.length >= 3) {

this.analyzeChoicePatterns();

}

}

analyzeChoicePatterns() {

const choices = this.sessionData.choiceHistory;

// Calculate how often the user picks similar positions

let positionBias = 0;

const positions = choices.map(c => c.choiceIndex);

// Check if user keeps picking same position

const positionCounts = positions.reduce((acc, pos) => {

acc[pos] = (acc[pos] || 0) + 1;

return acc;

}, {});

const maxCount = Math.max(...Object.values(positionCounts));

const maxPosition = positions.length > 0 ? maxCount / positions.length : 0;

// Higher value means user tends to pick the same position repeatedly

positionBias = maxPosition;

// Update choice consistency metric (inverted - lower means more consistent/predictable)

this.metrics.choiceConsistency = 1 - positionBias;

}

endSession(completed = false) {

// Record session duration

const sessionDurationMs = Date.now() - this.sessionData.startTime;

const sessionDurationMinutes = sessionDurationMs / 60000;

// Update metrics

this.metrics.sessionDuration =

(this.metrics.sessionDuration + sessionDurationMinutes) / 2;

if (completed) {

// Update completion rate

const totalCompletions = this.metrics.completionRate \* 10; // Assume based on 10 stories

this.metrics.completionRate = (totalCompletions + 1) / 11;

}

// Save updated metrics

this.saveMetrics();

// Reset session data

this.sessionData = {

startTime: Date.now(),

segmentsRead: 0,

totalReadingTime: 0,

choiceHistory: []

};

}

async saveMetrics() {

try {

await firebase.firestore()

.collection('userMetrics')

.doc(this.userId)

.update(this.metrics);

} catch (error

Copy

please continue

I'll continue with the implementation details of the TaleForge platform.

// src/services/adaptiveDifficultyManager.js (continued)

async saveMetrics() {

try {

await firebase.firestore()

.collection('userMetrics')

.doc(this.userId)

.update(this.metrics);

} catch (error) {

console.error('Error saving user metrics:', error);

}

}

getStoryParameters() {

// Convert user metrics into parameters for story generation

// Calculate vocabulary complexity (0-1 scale)

const vocabularyLevel = Math.min(0.9, Math.max(0.3,

this.metrics.preferredComplexity \* 0.7 +

this.metrics.averageReadingTime / 20000 // Higher reading time suggests comfort with complexity

));

// Calculate narrative complexity (0-1 scale)

const narrativeComplexity = Math.min(0.9, Math.max(0.3,

this.metrics.preferredComplexity \* 0.5 +

this.metrics.completionRate \* 0.3 +

this.metrics.choiceConsistency \* 0.2

));

// Calculate choice diversity (more or fewer similar choices)

const choiceDiversity = Math.min(0.9, Math.max(0.3,

0.6 + (this.metrics.choiceConsistency - 0.5) \* 0.8

));

return {

vocabularyLevel,

narrativeComplexity,

choiceDiversity,

preferredSessionLength: Math.max(5, Math.min(15,

Math.round(this.metrics.sessionDuration \* 0.7)

))

};

}

updatePrompt(basePrompt) {

const params = this.getStoryParameters();

// Enhance the base prompt with adaptive parameters

const enhancedPrompt = `

${basePrompt}

Create this story with the following parameters:

- Vocabulary complexity: ${params.vocabularyLevel < 0.4 ? 'simple and accessible' :

params.vocabularyLevel < 0.7 ? 'moderate and engaging' :

'rich and sophisticated'}

- Narrative complexity: ${params.narrativeComplexity < 0.4 ? 'straightforward with clear goals' :

params.narrativeComplexity < 0.7 ? 'moderate with some twists' :

'complex with multiple layers'}

- Choice diversity: ${params.choiceDiversity < 0.4 ? 'present clear contrasts between choices' :

params.choiceDiversity < 0.7 ? 'offer a mix of similar and different options' :

'provide subtle variations that test careful reading'}

This user typically engages for about ${params.preferredSessionLength} minutes per session, so pace the narrative accordingly.

`;

return enhancedPrompt;

}

}

module.exports = AdaptiveDifficultyManager;

**Story Analytics Dashboard**

To provide users with insights about their storytelling journey, we'll create an analytics dashboard:

// frontend/src/components/analytics/StoryAnalyticsDashboard.svelte

<script>

import { onMount } from 'svelte';

import Chart from 'chart.js/auto';

import firebase from '../../services/firebase';

import { user } from '../../stores/user';

import PixelCard from '../common/PixelCard.svelte';

import PixelIcon from '../common/PixelIcon.svelte';

import { formatDate } from '../../utils/dateUtils';

let storiesData = [];

let genreDistribution = {};

let choicePatterns = [];

let storyCompletionRate = 0;

let totalPlaytime = 0;

let favoriteGenre = '';

let uniqueEndings = 0;

let chartInstances = {};

onMount(async () => {

if ($user) {

await loadUserAnalytics();

renderCharts();

}

});

async function loadUserAnalytics() {

try {

// Get user's stories

const storiesSnapshot = await firebase.firestore()

.collection('stories')

.where('creatorId', '==', $user.id)

.get();

storiesData = storiesSnapshot.docs.map(doc => ({

id: doc.id,

...doc.data()

}));

// Calculate genre distribution

genreDistribution = storiesData.reduce((acc, story) => {

acc[story.genre] = (acc[story.genre] || 0) + 1;

return acc;

}, {});

// Determine favorite genre

if (Object.keys(genreDistribution).length > 0) {

favoriteGenre = Object.entries(genreDistribution)

.sort((a, b) => b[1] - a[1])[0][0];

}

// Calculate completion rate

const completedStories = storiesData.filter(s => s.isCompleted).length;

storyCompletionRate = storiesData.length > 0

? (completedStories / storiesData.length) \* 100

: 0;

// Get user progress data

const progressSnapshot = await firebase.firestore()

.collection('userProgress')

.where('userId', '==', $user.id)

.get();

const progressData = progressSnapshot.docs.map(doc => doc.data());

// Calculate total playtime

totalPlaytime = progressData.reduce((total, progress) => {

return total + (progress.playtime || 0);

}, 0);

// Count unique endings

const endings = new Set();

progressData.forEach(progress => {

if (progress.endingId) {

endings.add(progress.endingId);

}

});

uniqueEndings = endings.size;

// Get choice patterns

const choiceSnapshot = await firebase.firestore()

.collection('userChoices')

.where('userId', '==', $user.id)

.orderBy('timestamp', 'desc')

.limit(100)

.get();

choicePatterns = choiceSnapshot.docs.map(doc => doc.data());

} catch (error) {

console.error('Error loading analytics data:', error);

}

}

function renderCharts() {

// Genre distribution chart

const genreCtx = document.getElementById('genre-chart').getContext('2d');

chartInstances.genre = new Chart(genreCtx, {

type: 'doughnut',

data: {

labels: Object.keys(genreDistribution),

datasets: [{

data: Object.values(genreDistribution),

backgroundColor: [

'#FF5B5B', // Pixel red

'#5B8CFF', // Pixel blue

'#FFD75B', // Pixel yellow

'#5BFF8C', // Pixel green

'#D75BFF' // Pixel purple

]

}]

},

options: {

responsive: true,

plugins: {

legend: {

position: 'right',

}

}

}

});

// Choice pattern chart

const choiceTypes = choicePatterns.reduce((acc, choice) => {

const type = choice.choiceType || 'unknown';

acc[type] = (acc[type] || 0) + 1;

return acc;

}, {});

const patternCtx = document.getElementById('pattern-chart').getContext('2d');

chartInstances.pattern = new Chart(patternCtx, {

type: 'radar',

data: {

labels: ['Combat', 'Dialogue', 'Exploration', 'Stealth', 'Puzzle'],

datasets: [{

label: 'Your Choices',

data: [

choiceTypes['combat'] || 0,

choiceTypes['dialogue'] || 0,

choiceTypes['exploration'] || 0,

choiceTypes['stealth'] || 0,

choiceTypes['puzzle'] || 0

],

fill: true,

backgroundColor: 'rgba(91, 140, 255, 0.2)',

borderColor: 'rgba(91, 140, 255, 1)',

pointBackgroundColor: 'rgba(91, 140, 255, 1)',

pointBorderColor: '#fff'

}]

},

options: {

responsive: true,

scales: {

r: {

angleLines: {

display: true

},

suggestedMin: 0

}

}

}

});

// Stories over time chart

const storiesByMonth = {};

storiesData.forEach(story => {

const date = story.createdAt?.toDate() || new Date();

const monthYear = `${date.getMonth() + 1}/${date.getFullYear()}`;

if (!storiesByMonth[monthYear]) {

storiesByMonth[monthYear] = 0;

}

storiesByMonth[monthYear]++;

});

const timelineCtx = document.getElementById('timeline-chart').getContext('2d');

chartInstances.timeline = new Chart(timelineCtx, {

type: 'line',

data: {

labels: Object.keys(storiesByMonth),

datasets: [{

label: 'Stories Created',

data: Object.values(storiesByMonth),

borderColor: '#FF5B5B',

backgroundColor: 'rgba(255, 91, 91, 0.1)',

tension: 0.2

}]

},

options: {

responsive: true,

scales: {

y: {

beginAtZero: true,

ticks: {

precision: 0

}

}

}

}

});

}

</script>

<div class="analytics-dashboard">

<h1>Your Adventure Analytics</h1>

<div class="stats-grid">

<PixelCard title="Total Stories">

<div class="stat-value">{storiesData.length}</div>

<PixelIcon name="book" />

</PixelCard>

<PixelCard title="Completion Rate">

<div class="stat-value">{storyCompletionRate.toFixed(0)}%</div>

<PixelIcon name="flag" />

</PixelCard>

<PixelCard title="Total Playtime">

<div class="stat-value">{Math.floor(totalPlaytime / 60)} hrs</div>

<PixelIcon name="clock" />

</PixelCard>

<PixelCard title="Unique Endings">

<div class="stat-value">{uniqueEndings}</div>

<PixelIcon name="paths" />

</PixelCard>

</div>

<div class="charts-grid">

<PixelCard title="Favorite Genres" wide={true}>

<canvas id="genre-chart"></canvas>

</PixelCard>

<PixelCard title="Choice Patterns">

<canvas id="pattern-chart"></canvas>

</PixelCard>

<PixelCard title="Storytelling Journey" wide={true}>

<canvas id="timeline-chart"></canvas>

</PixelCard>

</div>

<div class="insights">

<PixelCard title="Your Storyteller Profile">

<div class="profile-content">

<p>

Based on your choices and reading patterns, you're a

<strong>{favoriteGenre}</strong> enthusiast who

{storyCompletionRate > 75 ? 'almost always finishes your adventures!' :

storyCompletionRate > 50 ? 'frequently sees your stories through to the end.' :

'likes to explore multiple story beginnings.'}

</p>

<p>

You've discovered {uniqueEndings} unique endings so far, and spent

{Math.floor(totalPlaytime / 60)} hours in TaleForge worlds.

</p>

<p class="recommendation">

Recommended: Try a {getLeastPlayedGenre()} adventure next to broaden your

storytelling horizons!

</p>

</div>

</PixelCard>

</div>

</div>

<style>

.analytics-dashboard {

max-width: 1200px;

margin: 0 auto;

padding: 1rem;

}

h1 {

font-family: 'Press Start 2P', cursive;

text-align: center;

margin-bottom: 2rem;

font-size: 1.5rem;

}

.stats-grid {

display: grid;

grid-template-columns: repeat(auto-fit, minmax(200px, 1fr));

gap: 1rem;

margin-bottom: 2rem;

}

.charts-grid {

display: grid;

grid-template-columns: 1fr 1fr;

gap: 1rem;

margin-bottom: 2rem;

}

.insights {

margin-bottom: 2rem;

}

.stat-value {

font-size: 2rem;

font-family: 'Press Start 2P', cursive;

color: #FF5B5B;

margin: 1rem 0;

}

.profile-content {

padding: 1rem;

font-size: 1.1rem;

line-height: 1.6;

}

.recommendation {

margin-top: 1rem;

font-weight: bold;

color: #5B8CFF;

}

@media (max-width: 768px) {

.charts-grid {

grid-template-columns: 1fr;

}

}

</style>

**Story Recommendation Engine**

To keep users engaged, we'll create a sophisticated recommendation engine:

// src/services/recommendationEngine.js

const firebase = require('../services/firebase');

const nlp = require('compromise');

class RecommendationEngine {

constructor() {

this.userProfiles = new Map();

this.storyFeatures = new Map();

this.genreSimilarity = {

fantasy: { fantasy: 1.0, scifi: 0.3, mystery: 0.4, horror: 0.5 },

scifi: { fantasy: 0.3, scifi: 1.0, mystery: 0.5, horror: 0.6 },

mystery: { fantasy: 0.4, scifi: 0.5, mystery: 1.0, horror: 0.7 },

horror: { fantasy: 0.5, scifi: 0.6, mystery: 0.7, horror: 1.0 }

};

}

async buildUserProfile(userId) {

if (this.userProfiles.has(userId)) {

return this.userProfiles.get(userId);

}

try {

// Get user's completed stories

const storiesSnapshot = await firebase.firestore()

.collection('userProgress')

.where('userId', '==', userId)

.get();

const completedStoryIds = storiesSnapshot.docs

.filter(doc => doc.data().isCompleted)

.map(doc => doc.data().storyId);

// Get user's votes

const votesSnapshot = await firebase.firestore()

.collection('votes')

.where('userId', '==', userId)

.get();

const votedStoryIds = votesSnapshot.docs.map(doc => doc.data().storyId);

// Get details of all stories the user has interacted with

const allStoryIds = [...new Set([...completedStoryIds, ...votedStoryIds])];

if (allStoryIds.length === 0) {

// New user, create a generic profile

this.userProfiles.set(userId, {

genres: {},

themes: {},

complexity: 0.5,

interactionCount: 0

});

return this.userProfiles.get(userId);

}

// Fetch stories in batches (Firestore limit)

const storyChunks = [];

for (let i = 0; i < allStoryIds.length; i += 10) {

const chunk = allStoryIds.slice(i, i + 10);

storyChunks.push(chunk);

}

let stories = [];

for (const chunk of storyChunks) {

const storiesSnapshot = await firebase.firestore()

.collection('stories')

.where(firebase.firestore.FieldPath.documentId(), 'in', chunk)

.get();

stories = [...stories, ...storiesSnapshot.docs.map(doc => ({

id: doc.id,

...doc.data()

}))];

}

// Calculate genre preferences

const genres = {};

const themes = {};

let totalComplexity = 0;

stories.forEach(story => {

// Weight completed stories more heavily than just voted ones

const weight = completedStoryIds.includes(story.id) ? 2 : 1;

// Update genre counts

genres[story.genre] = (genres[story.genre] || 0) + weight;

// Update theme counts

(story.tags || []).forEach(tag => {

themes[tag] = (themes[tag] || 0) + weight;

});

// Add to complexity score

totalComplexity += (story.complexityScore || 0.5) \* weight;

// Cache story features for recommendation

this.storyFeatures.set(story.id, {

genre: story.genre,

tags: story.tags || [],

complexity: story.complexityScore || 0.5,

popularity: story.votes || 0

});

});

// Normalize preferences

const totalWeight = stories.reduce((sum, story) =>

sum + (completedStoryIds.includes(story.id) ? 2 : 1), 0);

Object.keys(genres).forEach(genre => {

genres[genre] /= totalWeight;

});

Object.keys(themes).forEach(theme => {

themes[theme] /= totalWeight;

});

const avgComplexity = totalComplexity / totalWeight;

// Create and cache user profile

const userProfile = {

genres,

themes,

complexity: avgComplexity,

interactionCount: stories.length

};

this.userProfiles.set(userId, userProfile);

return userProfile;

} catch (error) {

console.error('Error building user profile:', error);

return {

genres: {},

themes: {},

complexity: 0.5,

interactionCount: 0

};

}

}

async getStoryFeatures(storyId) {

if (this.storyFeatures.has(storyId)) {

return this.storyFeatures.get(storyId);

}

try {

const storyDoc = await firebase.firestore()

.collection('stories')

.doc(storyId)

.get();

if (!storyDoc.exists) {

return null;

}

const story = storyDoc.data();

// Extract features

const features = {

genre: story.genre,

tags: story.tags || [],

complexity: story.complexityScore || 0.5,

popularity: story.votes || 0

};

// Cache for future use

this.storyFeatures.set(storyId, features);

return features;

} catch (error) {

console.error('Error getting story features:', error);

return null;

}

}

calculateStoryScore(userProfile, storyFeatures) {

if (!userProfile || !storyFeatures) {

return 0;

}

// Genre similarity component (0-1)

let genreScore = 0;

const userGenres = userProfile.genres;

if (Object.keys(userGenres).length > 0) {

Object.entries(userGenres).forEach(([userGenre, weight]) => {

const similarity = this.genreSimilarity[userGenre]?.[storyFeatures.genre] || 0.1;

genreScore += similarity \* weight;

});

} else {

// New user with no preferences

genreScore = 0.5;

}

// Theme matching component (0-1)

let themeScore = 0;

const userThemes = userProfile.themes;

if (Object.keys(userThemes).length > 0 && storyFeatures.tags.length > 0) {

let matchCount = 0;

storyFeatures.tags.forEach(tag => {

if (userThemes[tag]) {

matchCount += userThemes[tag];

}

});

themeScore = Math.min(1, matchCount / Math.max(1, storyFeatures.tags.length));

} else {

// No theme data to compare

themeScore = 0.5;

}

// Complexity match (0-1)

const complexityDiff = Math.abs(userProfile.complexity - storyFeatures.popularity);

const complexityScore = 1 - Math.min(1, complexityDiff);

// Popularity boost (0-0.5)

// Scale popularity to give some boost but not overwhelm personalization

const popularityScore = Math.min(0.5, storyFeatures.popularity / 100);

// Novelty for users with established preferences

let noveltyScore = 0;

if (userProfile.interactionCount > 5) {

// If user has several interactions, slightly boost stories with different features

// to encourage exploration

noveltyScore = userGenres[storyFeatures.genre] ? 0 : 0.2;

}

// Calculate weighted total score

const totalScore =

(genreScore \* 0.4) +

(themeScore \* 0.3) +

(complexityScore \* 0.1) +

(popularityScore \* 0.1) +

(noveltyScore \* 0.1);

return totalScore;

}

async getRecommendationsForUser(userId, limit = 5) {

try {

// Build user profile if not already cached

const userProfile = await this.buildUserProfile(userId);

// Get stories the user hasn't interacted with

const userHistorySnapshot = await firebase.firestore()

.collection('userProgress')

.where('userId', '==', userId)

.get();

const seenStoryIds = new Set(

userHistorySnapshot.docs.map(doc => doc.data().storyId)

);

// Get public stories to recommend

const storiesSnapshot = await firebase.firestore()

.collection('stories')

.where('isPublic', '==', true)

.limit(100) // Limit the pool size for performance

.get();

const candidateStories = storiesSnapshot.docs

.map(doc => ({

id: doc.id,

...doc.data()

}))

.filter(story => !seenStoryIds.has(story.id));

// Score each candidate story

const scoredStories = await Promise.all(

candidateStories.map(async story => {

const storyFeatures = await this.getStoryFeatures(story.id);

const score = this.calculateStoryScore(userProfile, storyFeatures);

return {

story,

score

};

})

);

// Sort by score and return top recommendations

const recommendations = scoredStories

.sort((a, b) => b.score - a.score)

.slice(0, limit)

.map(item => item.story);

return recommendations;

} catch (error) {

console.error('Error getting recommendations:', error);

return [];

}

}

}

module.exports = new RecommendationEngine();

**Marketing Features**

**Story Sharing Features**

To enhance virality and user engagement, we'll implement social sharing features:

// frontend/src/services/socialSharingService.js

import { generateThumbnail } from './thumbnailGenerator';

export async function shareToSocial(story, platform, options = {}) {

const { segmentId, customMessage } = options;

try {

// Generate a shareable thumbnail

const thumbnailUrl = await generateThumbnail(story.id, segmentId);

// Construct sharing URL

const shareUrl = `https://taleforge.netlify.app/story/${story.id}${segmentId ? `/segment/${segmentId}` : ''}`;

// Create default message if not provided

const message = customMessage ||

`Check out my adventure "${story.title}" on TaleForge! #InteractiveStorytelling #PixelArt`;

// Handle different social platforms

switch (platform) {

case 'twitter':

const twitterUrl = `https://twitter.com/intent/tweet?url=${encodeURIComponent(shareUrl)}&text=${encodeURIComponent(message)}`;

window.open(twitterUrl, '\_blank');

break;

case 'facebook':

const fbUrl = `https://www.facebook.com/sharer/sharer.php?u=${encodeURIComponent(shareUrl)}&quote=${encodeURIComponent(message)}`;

window.open(fbUrl, '\_blank');

break;

case 'reddit':

const redditUrl = `https://www.reddit.com/submit?url=${encodeURIComponent(shareUrl)}&title=${encodeURIComponent(message)}`;

window.open(redditUrl, '\_blank');

break;

case 'clipboard':

const linkText = `${story.title}\n${message}\n${shareUrl}`;

await navigator.clipboard.writeText(linkText);

return { success: true, method: 'clipboard' };

case 'embed':

// Generate embed code for websites/blogs

const embedCode = `<iframe src="${shareUrl}?embed=true" width="800" height="600" frameborder="0" allowfullscreen></iframe>`;

await navigator.clipboard.writeText(embedCode);

return { success: true, method: 'embed', code: embedCode };

default:

throw new Error(`Unsupported platform: ${platform}`);

}

// Track sharing event

trackSocialShare(story.id, platform);

return { success: true, platform, shareUrl };

} catch (error) {

console.error('Error sharing story:', error);

return { success: false, error: error.message };

}

}

// Track sharing for analytics

function trackSocialShare(storyId, platform) {

try {

firebase.analytics().logEvent('share\_story', {

story\_id: storyId,

platform

});

// Also store in Firestore for more detailed analytics

firebase.firestore()

.collection('socialShares')

.add({

storyId,

platform,

userId: firebase.auth().currentUser?.uid || 'anonymous',

timestamp: firebase.firestore.FieldValue.serverTimestamp()

});

} catch (error) {

console.error('Error tracking share:', error);

}

}

// Generate story preview image for social sharing

export async function generateSocialPreview(story) {

try {

const response = await fetch('/api/social-preview', {

method: 'POST',

headers: {

'Content-Type': 'application/json'

},

body: JSON.stringify({

storyId: story.id,

title: story.title,

genre: story.genre

})

});

const data = await response.json();

return data.previewUrl;

} catch (error) {

console.error('Error generating social preview:', error);

return story.thumbnailUrl; // Fallback to regular thumbnail

}

}

**Story Achievement System**

To encourage exploration and replay, we'll implement an achievement system:

// src/services/achievementSystem.js

const firebase = require('../services/firebase');

class AchievementSystem {

constructor() {

this.achievementDefinitions = [

{

id: 'first\_story',

name: 'Novice Storyteller',

description: 'Complete your first story',

icon: 'achievement-first-story.png',

condition: stats => stats.completedStories >= 1,

rarity: 'common'

},

{

id: 'story\_marathon',

name: 'Story Marathon',

description: 'Complete 10 different stories',

icon: 'achievement-marathon.png',

condition: stats => stats.completedStories >= 10,

rarity: 'rare'

},

{

id: 'genre\_explorer',

name: 'Genre Explorer',

description: 'Complete stories from all available genres',

icon: 'achievement-genres.png',

condition: stats => Object.keys(stats.completedGenres).length >= 4,

rarity: 'epic'

},

{

id: 'decisive',

name: 'Decisive',

description: 'Make 100 story choices',

icon: 'achievement-choices.png',

condition: stats => stats.totalChoices >= 100,

rarity: 'common'

},

{

id: 'speed\_reader',

name: 'Speed Reader',

description: 'Complete a story in under 5 minutes',

icon: 'achievement-speed.png',

condition: stats => stats.fastestCompletion < 300,

rarity: 'rare'

},

{

id: 'completionist',

name: 'Completionist',

description: 'Discover all endings in a single story',

icon: 'achievement-completionist.png',

condition: stats => stats.maxEndingsPerStory >= 5,

rarity: 'legendary'

},

{

id: 'popular\_author',

name: 'Popular Author',

description: 'Receive 50 votes on your shared stories',

icon: 'achievement-popular.png',

condition: stats => stats.totalVotesReceived >= 50,

rarity: 'epic'

},

{

id: 'night\_owl',

name: 'Night Owl',

description: 'Complete a story between midnight and 5am',

icon: 'achievement-night.png',

condition: stats => stats.nightTimeCompletions >= 1,

rarity: 'uncommon'

}

];

}

async checkAchievements(userId) {

try {

// Get user stats

const userStats = await this.getUserStats(userId);

// Get currently unlocked achievements

const achievementsSnapshot = await firebase.firestore()

.collection('userAchievements')

.where('userI

Copy

please continue

I'll continue with the implementation of the TaleForge platform:

// src/services/achievementSystem.js (continued)

async checkAchievements(userId) {

try {

// Get user stats

const userStats = await this.getUserStats(userId);

// Get currently unlocked achievements

const achievementsSnapshot = await firebase.firestore()

.collection('userAchievements')

.where('userId', '==', userId)

.get();

const unlockedAchievements = new Set(

achievementsSnapshot.docs.map(doc => doc.data().achievementId)

);

// Check for new achievements

const newAchievements = [];

for (const achievement of this.achievementDefinitions) {

if (!unlockedAchievements.has(achievement.id) &&

achievement.condition(userStats)) {

// User has earned this achievement

// Record the achievement

await firebase.firestore()

.collection('userAchievements')

.add({

userId,

achievementId: achievement.id,

unlockedAt: firebase.firestore.FieldValue.serverTimestamp()

});

// Add to new achievements list

newAchievements.push(achievement);

}

}

return {

newAchievements,

totalAchievements: unlockedAchievements.size + newAchievements.length,

availableAchievements: this.achievementDefinitions.length

};

} catch (error) {

console.error('Error checking achievements:', error);

return {

newAchievements: [],

totalAchievements: 0,

availableAchievements: this.achievementDefinitions.length

};

}

}

async getUserStats(userId) {

try {

// Get user story progress

const progressSnapshot = await firebase.firestore()

.collection('userProgress')

.where('userId', '==', userId)

.get();

const progressData = progressSnapshot.docs.map(doc => doc.data());

// Track completed stories and their genres

const completedStories = new Set();

const completedGenres = {};

let totalChoices = 0;

let fastestCompletion = Infinity;

let nightTimeCompletions = 0;

for (const progress of progressData) {

if (progress.isCompleted) {

completedStories.add(progress.storyId);

// Get story details

const storyDoc = await firebase.firestore()

.collection('stories')

.doc(progress.storyId)

.get();

if (storyDoc.exists) {

const storyData = storyDoc.data();

// Track genre

completedGenres[storyData.genre] =

(completedGenres[storyData.genre] || 0) + 1;

// Track completion time

if (progress.completionTimeSeconds &&

progress.completionTimeSeconds < fastestCompletion) {

fastestCompletion = progress.completionTimeSeconds;

}

// Check if completed at night

const completionTime = progress.completedAt?.toDate();

if (completionTime) {

const hour = completionTime.getHours();

if (hour >= 0 && hour < 5) {

nightTimeCompletions++;

}

}

}

}

// Track total choices made

totalChoices += (progress.choicesMade || 0);

}

// Count endings discovered per story

const storyEndingsSnapshot = await firebase.firestore()

.collection('userStoryEndings')

.where('userId', '==', userId)

.get();

const endingsByStory = {};

storyEndingsSnapshot.docs.forEach(doc => {

const data = doc.data();

endingsByStory[data.storyId] = (endingsByStory[data.storyId] || 0) + 1;

});

const maxEndingsPerStory = Object.values(endingsByStory).length > 0

? Math.max(...Object.values(endingsByStory))

: 0;

// Count votes received on stories

const storiesSnapshot = await firebase.firestore()

.collection('stories')

.where('creatorId', '==', userId)

.get();

const totalVotesReceived = storiesSnapshot.docs.reduce(

(sum, doc) => sum + (doc.data().votes || 0),

0

);

// Return compiled stats

return {

completedStories: completedStories.size,

completedGenres,

totalChoices,

fastestCompletion,

nightTimeCompletions,

maxEndingsPerStory,

totalVotesReceived

};

} catch (error) {

console.error('Error getting user stats:', error);

return {

completedStories: 0,

completedGenres: {},

totalChoices: 0,

fastestCompletion: Infinity,

nightTimeCompletions: 0,

maxEndingsPerStory: 0,

totalVotesReceived: 0

};

}

}

async getAchievements(userId) {

try {

// Get unlocked achievements

const achievementsSnapshot = await firebase.firestore()

.collection('userAchievements')

.where('userId', '==', userId)

.get();

const unlockedIds = new Set();

const unlocked = {};

achievementsSnapshot.docs.forEach(doc => {

const data = doc.data();

unlockedIds.add(data.achievementId);

unlocked[data.achievementId] = {

unlockedAt: data.unlockedAt?.toDate() || new Date(),

...this.getAchievementDefinition(data.achievementId)

};

});

// Get locked achievements

const locked = this.achievementDefinitions

.filter(achievement => !unlockedIds.has(achievement.id))

.map(achievement => ({

...achievement,

locked: true

}));

return {

unlocked: Object.values(unlocked),

locked,

progress: {

total: this.achievementDefinitions.length,

unlocked: unlockedIds.size,

percentage: Math.round((unlockedIds.size / this.achievementDefinitions.length) \* 100)

}

};

} catch (error) {

console.error('Error getting achievements:', error);

return {

unlocked: [],

locked: this.achievementDefinitions,

progress: {

total: this.achievementDefinitions.length,

unlocked: 0,

percentage: 0

}

};

}

}

getAchievementDefinition(achievementId) {

return this.achievementDefinitions.find(

achievement => achievement.id === achievementId

);

}

}

module.exports = new AchievementSystem();

**Advanced Technical Implementation**

**WebAssembly Audio Processing**

To optimize the audio playback experience, we'll implement a WebAssembly module for efficient audio processing:

// frontend/src/services/audioProcessor.js

class AudioProcessor {

constructor() {

this.wasmInstance = null;

this.audioContext = null;

this.gainNode = null;

this.initialized = false;

this.bufferCache = new Map();

}

async initialize() {

if (this.initialized) return;

try {

// Load WebAssembly module

const response = await fetch('/audio-processor.wasm');

const buffer = await response.arrayBuffer();

const wasm = await WebAssembly.instantiate(buffer, {

env: {

// JS functions exposed to WASM

consoleLog: (ptr, len) => {

const bytes = new Uint8Array(this.wasmInstance.exports.memory.buffer, ptr, len);

const message = new TextDecoder('utf8').decode(bytes);

console.log('[WASM]', message);

}

}

});

this.wasmInstance = wasm.instance;

// Initialize Web Audio API

this.audioContext = new (window.AudioContext || window.webkitAudioContext)();

this.gainNode = this.audioContext.createGain();

this.gainNode.connect(this.audioContext.destination);

this.initialized = true;

} catch (error) {

console.error('Failed to initialize AudioProcessor:', error);

// Fall back to standard audio processing

}

}

async processAudio(audioBuffer, options = {}) {

await this.initialize();

if (!this.wasmInstance) {

// Fall back to native processing

return this.processAudioNative(audioBuffer, options);

}

try {

// Allocate memory in WASM for the audio data

const numSamples = audioBuffer.length;

const bytesPerSample = 4; // Float32 = 4 bytes

const bufferSize = numSamples \* bytesPerSample;

const inputPtr = this.wasmInstance.exports.allocateBuffer(bufferSize);

const outputPtr = this.wasmInstance.exports.allocateBuffer(bufferSize);

// Copy audio data to WASM memory

const memory = this.wasmInstance.exports.memory.buffer;

const inputView = new Float32Array(memory, inputPtr, numSamples);

inputView.set(audioBuffer);

// Process audio in WASM

// Apply effects based on options

const reverb = options.reverb || 0.0;

const pitch = options.pitch || 1.0;

const speed = options.speed || 1.0;

this.wasmInstance.exports.processAudio(

inputPtr, outputPtr, numSamples, reverb, pitch, speed

);

// Copy processed audio back to JavaScript

const outputView = new Float32Array(memory, outputPtr, numSamples);

const processedBuffer = new Float32Array(numSamples);

processedBuffer.set(outputView);

// Free WASM memory

this.wasmInstance.exports.freeBuffer(inputPtr);

this.wasmInstance.exports.freeBuffer(outputPtr);

return processedBuffer;

} catch (error) {

console.error('Error in WASM audio processing:', error);

// Fall back to native processing

return this.processAudioNative(audioBuffer, options);

}

}

processAudioNative(audioBuffer, options = {}) {

// JavaScript fallback implementation

const { reverb = 0.0, pitch = 1.0, speed = 1.0 } = options;

// Create a copy of the buffer

const processedBuffer = new Float32Array(audioBuffer.length);

// Simple processing (actual implementation would be more sophisticated)

for (let i = 0; i < audioBuffer.length; i++) {

processedBuffer[i] = audioBuffer[i];

// Apply a simple reverb effect (echo)

if (reverb > 0 && i >= 4000) {

processedBuffer[i] += audioBuffer[i - 4000] \* reverb \* 0.5;

}

}

return processedBuffer;

}

async playAudio(url, options = {}) {

await this.initialize();

try {

// Check if audio is cached

let audioBuffer;

if (this.bufferCache.has(url)) {

audioBuffer = this.bufferCache.get(url);

} else {

// Fetch and decode audio

const response = await fetch(url);

const arrayBuffer = await response.arrayBuffer();

audioBuffer = await this.audioContext.decodeAudioData(arrayBuffer);

// Cache for future use

this.bufferCache.set(url, audioBuffer);

}

// Create and configure source node

const sourceNode = this.audioContext.createBufferSource();

sourceNode.buffer = audioBuffer;

// Apply processing if needed

if (options.pitch !== 1.0 || options.reverb > 0) {

// Process audio data with WebAssembly

const channelData = audioBuffer.getChannelData(0);

const processedData = await this.processAudio(channelData, options);

// Create a new buffer with processed data

const processedBuffer = this.audioContext.createBuffer(

audioBuffer.numberOfChannels,

audioBuffer.length,

audioBuffer.sampleRate

);

// Copy processed data to the buffer

processedBuffer.getChannelData(0).set(processedData);

// Use processed buffer

sourceNode.buffer = processedBuffer;

}

// Connect to gain node for volume control

sourceNode.connect(this.gainNode);

// Set playback rate (speed)

sourceNode.playbackRate.value = options.speed || 1.0;

// Set volume

this.gainNode.gain.value = options.volume !== undefined ? options.volume : 1.0;

// Start playback

sourceNode.start(0);

return {

source: sourceNode,

stop: () => sourceNode.stop(),

setVolume: (volume) => {

this.gainNode.gain.value = volume;

}

};

} catch (error) {

console.error('Error playing audio:', error);

throw error;

}

}

}

export default new AudioProcessor();

**Offline Support with IndexedDB**

To ensure a seamless experience even when connectivity is lost, we'll implement robust offline support using IndexedDB:

// frontend/src/services/offlineStorageService.js

class OfflineStorageService {

constructor() {

this.dbName = 'TaleForgeOfflineDB';

this.dbVersion = 1;

this.db = null;

}

async openDatabase() {

if (this.db) return this.db;

return new Promise((resolve, reject) => {

const request = indexedDB.open(this.dbName, this.dbVersion);

request.onerror = (event) => {

console.error('Error opening IndexedDB:', event.target.error);

reject(event.target.error);

};

request.onupgradeneeded = (event) => {

const db = event.target.result;

// Create object stores

if (!db.objectStoreNames.contains('stories')) {

const storyStore = db.createObjectStore('stories', { keyPath: 'id' });

storyStore.createIndex('creatorId', 'creatorId', { unique: false });

storyStore.createIndex('isPublic', 'isPublic', { unique: false });

}

if (!db.objectStoreNames.contains('segments')) {

const segmentStore = db.createObjectStore('segments', { keyPath: 'id' });

segmentStore.createIndex('storyId', 'storyId', { unique: false });

}

if (!db.objectStoreNames.contains('userProgress')) {

const progressStore = db.createObjectStore('userProgress', { keyPath: 'id' });

progressStore.createIndex('userId\_storyId', ['userId', 'storyId'], { unique: true });

}

if (!db.objectStoreNames.contains('syncQueue')) {

db.createObjectStore('syncQueue', {

keyPath: 'id',

autoIncrement: true

});

}

if (!db.objectStoreNames.contains('assets')) {

const assetStore = db.createObjectStore('assets', { keyPath: 'url' });

assetStore.createIndex('type', 'type', { unique: false });

assetStore.createIndex('storyId', 'storyId', { unique: false });

}

};

request.onsuccess = (event) => {

this.db = event.target.result;

resolve(this.db);

};

});

}

async saveStory(story) {

const db = await this.openDatabase();

return new Promise((resolve, reject) => {

const transaction = db.transaction(['stories'], 'readwrite');

const storyStore = transaction.objectStore('stories');

// Add timestamp for offline tracking

const storyWithTimestamp = {

...story,

offlineTimestamp: Date.now()

};

const request = storyStore.put(storyWithTimestamp);

request.onerror = (event) => {

console.error('Error saving story:', event.target.error);

reject(event.target.error);

};

request.onsuccess = () => {

resolve(story.id);

};

});

}

async saveSegment(segment) {

const db = await this.openDatabase();

return new Promise((resolve, reject) => {

const transaction = db.transaction(['segments'], 'readwrite');

const segmentStore = transaction.objectStore('segments');

// Add timestamp for offline tracking

const segmentWithTimestamp = {

...segment,

offlineTimestamp: Date.now()

};

const request = segmentStore.put(segmentWithTimestamp);

request.onerror = (event) => {

console.error('Error saving segment:', event.target.error);

reject(event.target.error);

};

request.onsuccess = () => {

resolve(segment.id);

};

});

}

async saveUserProgress(progress) {

const db = await this.openDatabase();

return new Promise((resolve, reject) => {

const transaction = db.transaction(['userProgress'], 'readwrite');

const progressStore = transaction.objectStore('userProgress');

// Generate composite ID if not provided

if (!progress.id) {

progress.id = `${progress.userId}\_${progress.storyId}`;

}

// Add timestamp for offline tracking

const progressWithTimestamp = {

...progress,

offlineTimestamp: Date.now()

};

const request = progressStore.put(progressWithTimestamp);

request.onerror = (event) => {

console.error('Error saving progress:', event.target.error);

reject(event.target.error);

};

request.onsuccess = () => {

resolve(progress.id);

};

});

}

async getStory(storyId) {

const db = await this.openDatabase();

return new Promise((resolve, reject) => {

const transaction = db.transaction(['stories'], 'readonly');

const storyStore = transaction.objectStore('stories');

const request = storyStore.get(storyId);

request.onerror = (event) => {

console.error('Error getting story:', event.target.error);

reject(event.target.error);

};

request.onsuccess = () => {

resolve(request.result);

};

});

}

async getSegment(segmentId) {

const db = await this.openDatabase();

return new Promise((resolve, reject) => {

const transaction = db.transaction(['segments'], 'readonly');

const segmentStore = transaction.objectStore('segments');

const request = segmentStore.get(segmentId);

request.onerror = (event) => {

console.error('Error getting segment:', event.target.error);

reject(event.target.error);

};

request.onsuccess = () => {

resolve(request.result);

};

});

}

async getStorySegments(storyId) {

const db = await this.openDatabase();

return new Promise((resolve, reject) => {

const transaction = db.transaction(['segments'], 'readonly');

const segmentStore = transaction.objectStore('segments');

const index = segmentStore.index('storyId');

const request = index.getAll(storyId);

request.onerror = (event) => {

console.error('Error getting segments:', event.target.error);

reject(event.target.error);

};

request.onsuccess = () => {

resolve(request.result);

};

});

}

async getUserProgress(userId, storyId) {

const db = await this.openDatabase();

return new Promise((resolve, reject) => {

const transaction = db.transaction(['userProgress'], 'readonly');

const progressStore = transaction.objectStore('userProgress');

const index = progressStore.index('userId\_storyId');

const request = index.get([userId, storyId]);

request.onerror = (event) => {

console.error('Error getting progress:', event.target.error);

reject(event.target.error);

};

request.onsuccess = () => {

resolve(request.result);

};

});

}

async saveAsset(url, data, type, storyId = null) {

const db = await this.openDatabase();

return new Promise((resolve, reject) => {

const transaction = db.transaction(['assets'], 'readwrite');

const assetStore = transaction.objectStore('assets');

const asset = {

url,

data,

type,

storyId,

timestamp: Date.now()

};

const request = assetStore.put(asset);

request.onerror = (event) => {

console.error('Error saving asset:', event.target.error);

reject(event.target.error);

};

request.onsuccess = () => {

resolve(url);

};

});

}

async getAsset(url) {

const db = await this.openDatabase();

return new Promise((resolve, reject) => {

const transaction = db.transaction(['assets'], 'readonly');

const assetStore = transaction.objectStore('assets');

const request = assetStore.get(url);

request.onerror = (event) => {

console.error('Error getting asset:', event.target.error);

reject(event.target.error);

};

request.onsuccess = () => {

resolve(request.result);

};

});

}

async addToSyncQueue(operation) {

const db = await this.openDatabase();

return new Promise((resolve, reject) => {

const transaction = db.transaction(['syncQueue'], 'readwrite');

const queueStore = transaction.objectStore('syncQueue');

// Add operation to sync when online

const syncOperation = {

...operation,

timestamp: Date.now(),

attempts: 0

};

const request = queueStore.add(syncOperation);

request.onerror = (event) => {

console.error('Error adding to sync queue:', event.target.error);

reject(event.target.error);

};

request.onsuccess = () => {

resolve(request.result); // Returns the auto-generated ID

};

});

}

async processSyncQueue() {

if (!navigator.onLine) {

console.log('Still offline, sync queue processing delayed');

return { success: false, reason: 'offline' };

}

const db = await this.openDatabase();

return new Promise((resolve, reject) => {

const transaction = db.transaction(['syncQueue'], 'readonly');

const queueStore = transaction.objectStore('syncQueue');

const request = queueStore.getAll();

request.onerror = (event) => {

console.error('Error getting sync queue:', event.target.error);

reject(event.target.error);

};

request.onsuccess = async () => {

const operations = request.result;

if (operations.length === 0) {

resolve({ success: true, synced: 0 });

return;

}

console.log(`Processing ${operations.length} queued operations`);

let successful = 0;

let failed = 0;

for (const operation of operations) {

try {

switch (operation.type) {

case 'saveStory':

await this.syncStory(operation.data);

break;

case 'saveSegment':

await this.syncSegment(operation.data);

break;

case 'saveProgress':

await this.syncProgress(operation.data);

break;

case 'vote':

await this.syncVote(operation.data);

break;

default:

console.warn(`Unknown sync operation type: ${operation.type}`);

continue;

}

// Operation succeeded, remove from queue

await this.removeFromSyncQueue(operation.id);

successful++;

} catch (error) {

console.error(`Sync operation failed:`, error);

// Update attempt counter

operation.attempts++;

if (operation.attempts < 3) {

// Will retry next time

await this.updateSyncQueueItem(operation);

} else {

// Too many failures, remove from queue

await this.removeFromSyncQueue(operation.id);

console.error(`Gave up on operation after 3 attempts:`, operation);

}

failed++;

}

}

resolve({

success: true,

synced: successful,

failed

});

};

});

}

async removeFromSyncQueue(id) {

const db = await this.openDatabase();

return new Promise((resolve, reject) => {

const transaction = db.transaction(['syncQueue'], 'readwrite');

const queueStore = transaction.objectStore('syncQueue');

const request = queueStore.delete(id);

request.onerror = (event) => {

console.error('Error removing from sync queue:', event.target.error);

reject(event.target.error);

};

request.onsuccess = () => {

resolve();

};

});

}

async updateSyncQueueItem(operation) {

const db = await this.openDatabase();

return new Promise((resolve, reject) => {

const transaction = db.transaction(['syncQueue'], 'readwrite');

const queueStore = transaction.objectStore('syncQueue');

const request = queueStore.put(operation);

request.onerror = (event) => {

console.error('Error updating sync queue item:', event.target.error);

reject(event.target.error);

};

request.onsuccess = () => {

resolve();

};

});

}

// Helper methods for syncing with backend

async syncStory(story) {

const response = await fetch('/api/stories/sync', {

method: 'POST',

headers: {

'Content-Type': 'application/json'

},

body: JSON.stringify(story)

});

if (!response.ok) {

throw new Error(`Failed to sync story: ${response.statusText}`);

}

return await response.json();

}

async syncSegment(segment) {

const response = await fetch('/api/segments/sync', {

method: 'POST',

headers: {

'Content-Type': 'application/json'

},

body: JSON.stringify(segment)

});

if (!response.ok) {

throw new Error(`Failed to sync segment: ${response.statusText}`);

}

return await response.json();

}

async syncProgress(progress) {

const response = await fetch('/api/progress/sync', {

method: 'POST',

headers: {

'Content-Type': 'application/json'

},

body: JSON.stringify(progress)

});

if (!response.ok) {

throw new Error(`Failed to sync progress: ${response.statusText}`);

}

return await response.json();

}

async syncVote(voteData) {

const response = await fetch(`/api/stories/${voteData.storyId}/vote`, {

method: 'POST',

headers: {

'Content-Type': 'application/json'

}

});

if (!response.ok) {

throw new Error(`Failed to sync vote: ${response.statusText}`);

}

return await response.json();

}

}

export default new OfflineStorageService();

**Final Integration**

To bring everything together, let's create a main application component that integrates all the features we've implemented:

<!-- frontend/src/App.svelte -->

<script>

import { onMount } from 'svelte';

import { Router, Route } from 'svelte-routing';

import { user, loading } from './stores/user';

import firebase from './services/firebase';

import offlineStorageService from './services/offlineStorageService';

import achievementService from './services/achievementService';

import audioProcessor from './services/audioProcessor';

// Pages

import Home from './pages/Home.svelte';

import StorySelect from './pages/StorySelect.svelte';

import StoryPlay from './pages/StoryPlay.svelte';

import Community from './pages/Community.svelte';

import Profile from './pages/Profile.svelte';

import Achievements from './pages/Achievements.svelte';

import Analytics from './pages/Analytics.svelte';

import Settings from './pages/Settings.svelte';

import NotFound from './pages/NotFound.svelte';

// Components

import LoadingScreen from './components/common/LoadingScreen.svelte';

import Navigation from './components/layout/Navigation.svelte';

import AchievementNotification from './components/notifications/AchievementNotification.svelte';

import StatusBar from './components/layout/StatusBar.svelte';

import PixelFooter from './components/layout/PixelFooter.svelte';

// App state

let isOffline = !navigator.onLine;

let showAchievement = false;

let newAchievement = null;

let syncStatus = 'idle';

let networkStatus = isOffline ? 'offline' : 'online';

onMount(async () => {

// Initialize audio processor early

audioProcessor.initialize();

// Set up authentication listener

firebase.auth().onAuthStateChanged(async (firebaseUser) => {

if (firebaseUser) {

// User is logged in

const userDoc = await firebase.firestore()

.collection('users')

.doc(firebaseUser.uid)

.get();

if (userDoc.exists) {

// Set user data in store

user.set({

id: firebaseUser.uid,

email: firebaseUser.email,

displayName: firebaseUser.displayName || userDoc.data().displayName,

avatarUrl: firebase

Copy

please continue

I'll continue with the final integration of the TaleForge app:

<!-- frontend/src/App.svelte (continued) -->

<script>

// continued from above

onMount(async () => {

// Initialize audio processor early

audioProcessor.initialize();

// Set up authentication listener

firebase.auth().onAuthStateChanged(async (firebaseUser) => {

if (firebaseUser) {

// User is logged in

const userDoc = await firebase.firestore()

.collection('users')

.doc(firebaseUser.uid)

.get();

if (userDoc.exists) {

// Set user data in store

user.set({

id: firebaseUser.uid,

email: firebaseUser.email,

displayName: firebaseUser.displayName || userDoc.data().displayName,

avatarUrl: firebaseUser.photoURL || userDoc.data().avatarUrl,

createdAt: userDoc.data().createdAt,

premium: userDoc.data().premium || false

});

// Check for new achievements

const achievementResult = await achievementService.checkAchievements(firebaseUser.uid);

if (achievementResult.newAchievements.length > 0) {

newAchievement = achievementResult.newAchievements[0];

showAchievement = true;

// Hide achievement notification after delay

setTimeout(() => {

showAchievement = false;

}, 5000);

}

} else {

// New user, create profile

const userData = {

id: firebaseUser.uid,

email: firebaseUser.email,

displayName: firebaseUser.displayName || `Player${Math.floor(Math.random()\*10000)}`,

avatarUrl: firebaseUser.photoURL || '/default-avatar.png',

createdAt: firebase.firestore.FieldValue.serverTimestamp(),

premium: false

};

await firebase.firestore()

.collection('users')

.doc(firebaseUser.uid)

.set(userData);

user.set(userData);

}

} else {

// User is logged out

user.set(null);

}

loading.set(false);

});

// Set up online/offline event listeners

window.addEventListener('online', handleOnline);

window.addEventListener('offline', handleOffline);

// Sync offline data when app loads

if (navigator.onLine) {

syncOfflineData();

}

// Register service worker for PWA

if ('serviceWorker' in navigator) {

try {

await navigator.serviceWorker.register('/service-worker.js');

console.log('Service worker registered');

} catch (error) {

console.error('Service worker registration failed:', error);

}

}

});

async function handleOnline() {

networkStatus = 'online';

isOffline = false;

// Sync data when connection is restored

await syncOfflineData();

}

function handleOffline() {

networkStatus = 'offline';

isOffline = true;

}

async function syncOfflineData() {

try {

syncStatus = 'syncing';

const result = await offlineStorageService.processSyncQueue();

if (result.synced > 0) {

console.log(`Successfully synced ${result.synced} items`);

}

syncStatus = 'idle';

} catch (error) {

console.error('Error syncing offline data:', error);

syncStatus = 'error';

}

}

function handleAchievementClose() {

showAchievement = false;

}

</script>

<svelte:head>

<title>TaleForge - Interactive Pixel Art Stories</title>

<meta name="description" content="Embark on AI-generated adventures with branching paths, pixel art, and narrated dialogue." />

<meta name="theme-color" content="#1F1F3D" />

<link rel="manifest" href="/manifest.json" />

<link href="https://fonts.googleapis.com/css2?family=Press+Start+2P&family=VT323&display=swap" rel="stylesheet">

</svelte:head>

<div class="app-container pixel-theme">

{#if $loading}

<LoadingScreen />

{:else}

<Router>

<StatusBar {networkStatus} {syncStatus} />

<main>

<Route path="/" component={Home} />

<Route path="/stories/select" component={StorySelect} />

<Route path="/story/:id" let:params>

<StoryPlay storyId={params.id} />

</Route>

<Route path="/story/:id/segment/:segmentId" let:params>

<StoryPlay storyId={params.id} segmentId={params.segmentId} />

</Route>

<Route path="/community" component={Community} />

<Route path="/profile" component={Profile} />

<Route path="/achievements" component={Achievements} />

<Route path="/analytics" component={Analytics} />

<Route path="/settings" component={Settings} />

<Route path="\*" component={NotFound} />

</main>

<Navigation />

<PixelFooter />

</Router>

{/if}

{#if showAchievement && newAchievement}

<AchievementNotification

achievement={newAchievement}

on:close={handleAchievementClose}

/>

{/if}

</div>

<style>

:global(body) {

margin: 0;

padding: 0;

font-family: 'VT323', monospace;

background-color: #1F1F3D;

color: #FFFFFF;

overflow-x: hidden;

}

.app-container {

display: flex;

flex-direction: column;

min-height: 100vh;

}

main {

flex: 1;

padding: 1rem;

padding-bottom: 5rem; /\* Space for navigation \*/

max-width: 1200px;

margin: 0 auto;

width: 100%;

box-sizing: border-box;

}

:global(.pixel-theme) {

--primary-color: #FF5B5B;

--secondary-color: #5B8CFF;

--accent-color: #FFD75B;

--background-color: #1F1F3D;

--text-color: #FFFFFF;

--border-color: #5B8CFF;

--success-color: #5BFF8C;

--error-color: #FF5B5B;

--warning-color: #FFD75B;

}

:global(.pixel-button) {

font-family: 'Press Start 2P', cursive;

background-color: var(--primary-color);

color: white;

border: none;

padding: 0.75rem 1.5rem;

font-size: 1rem;

cursor: pointer;

text-transform: uppercase;

position: relative;

image-rendering: pixelated;

box-shadow: 0px 4px 0px #B03C3C;

transition: all 0.1s ease;

}

:global(.pixel-button:hover) {

transform: translateY(-2px);

box-shadow: 0px 6px 0px #B03C3C;

}

:global(.pixel-button:active) {

transform: translateY(2px);

box-shadow: 0px 0px 0px #B03C3C;

}

:global(.pixel-button.secondary) {

background-color: var(--secondary-color);

box-shadow: 0px 4px 0px #3A5AA9;

}

:global(.pixel-button.secondary:hover) {

box-shadow: 0px 6px 0px #3A5AA9;

}

:global(.pixel-button.secondary:active) {

box-shadow: 0px 0px 0px #3A5AA9;

}

@media (max-width: 768px) {

main {

padding: 0.5rem;

}

}

</style>

Now, let's implement the Home page component:

<!-- frontend/src/pages/Home.svelte -->

<script>

import { onMount } from 'svelte';

import { user } from '../stores/user';

import { navigate } from 'svelte-routing';

import firebase from '../services/firebase';

import offlineStorageService from '../services/offlineStorageService';

import recommendationEngine from '../services/recommendationEngine';

import PixelButton from '../components/common/PixelButton.svelte';

import PixelCard from '../components/common/PixelCard.svelte';

import StoryCard from '../components/story/StoryCard.svelte';

import LoginPrompt from '../components/auth/LoginPrompt.svelte';

import PixelIcon from '../components/common/PixelIcon.svelte';

// Page data

let savedStories = [];

let recommendedStories = [];

let popularStories = [];

let isLoadingSaved = true;

let isLoadingRecommended = true;

let isLoadingPopular = true;

// UI state

let showLoginPrompt = false;

onMount(async () => {

loadPopularStories();

if ($user) {

// User is logged in

await Promise.all([

loadSavedStories(),

loadRecommendedStories()

]);

} else {

isLoadingSaved = false;

isLoadingRecommended = false;

}

});

async function loadSavedStories() {

try {

isLoadingSaved = true;

// First try offline storage

let offlineStories = [];

try {

const progressData = await offlineStorageService.getUserProgress(

$user.id, null

);

if (progressData && progressData.length > 0) {

// We have offline saved stories

offlineStories = await Promise.all(

progressData.map(async (progress) => {

const story = await offlineStorageService.getStory(progress.storyId);

return story ? { ...story, progress } : null;

})

);

// Filter out nulls

offlineStories = offlineStories.filter(story => story);

}

} catch (error) {

console.error('Error loading offline stories:', error);

}

// Then try Firebase

const progressSnapshot = await firebase.firestore()

.collection('userProgress')

.where('userId', '==', $user.id)

.orderBy('updatedAt', 'desc')

.limit(5)

.get();

const progressData = progressSnapshot.docs.map(doc => ({

id: doc.id,

...doc.data()

}));

if (progressData.length > 0) {

// Get the stories for each progress entry

const storyPromises = progressData.map(async (progress) => {

const storyDoc = await firebase.firestore()

.collection('stories')

.doc(progress.storyId)

.get();

if (storyDoc.exists) {

return {

id: storyDoc.id,

...storyDoc.data(),

progress

};

}

return null;

});

const stories = await Promise.all(storyPromises);

savedStories = stories.filter(story => story); // Filter out nulls

// Cache stories locally

for (const story of savedStories) {

try {

await offlineStorageService.saveStory(story);

} catch (error) {

console.error('Error caching story:', error);

}

}

} else if (offlineStories.length > 0) {

// Use offline stories if no Firebase data

savedStories = offlineStories;

}

} catch (error) {

console.error('Error loading saved stories:', error);

savedStories = [];

} finally {

isLoadingSaved = false;

}

}

async function loadRecommendedStories() {

try {

isLoadingRecommended = true;

if ($user) {

// Get recommendations from service

recommendedStories = await recommendationEngine.getRecommendationsForUser($user.id, 3);

} else {

// For anonymous users, just show some random stories

recommendedStories = [];

}

} catch (error) {

console.error('Error loading recommended stories:', error);

recommendedStories = [];

} finally {

isLoadingRecommended = false;

}

}

async function loadPopularStories() {

try {

isLoadingPopular = true;

const storiesSnapshot = await firebase.firestore()

.collection('stories')

.where('isPublic', '==', true)

.orderBy('votes', 'desc')

.limit(3)

.get();

popularStories = storiesSnapshot.docs.map(doc => ({

id: doc.id,

...doc.data()

}));

} catch (error) {

console.error('Error loading popular stories:', error);

popularStories = [];

} finally {

isLoadingPopular = false;

}

}

function startNewStory() {

if ($user) {

navigate('/stories/select');

} else {

showLoginPrompt = true;

}

}

function handleLoginSuccess() {

showLoginPrompt = false;

navigate('/stories/select');

}

function handleLoginCancel() {

showLoginPrompt = false;

}

function continueStory(storyId, segmentId) {

navigate(`/story/${storyId}${segmentId ? `/segment/${segmentId}` : ''}`);

}

</script>

<div class="home-page">

<header class="hero">

<div class="hero-content">

<img src="/logo.png" alt="TaleForge Logo" class="logo" />

<h1>Your Adventure Awaits</h1>

<p>Embark on interactive pixel-art stories driven by your choices</p>

<div class="hero-buttons">

<PixelButton on:click={startNewStory}>

<PixelIcon name="start" />

Start New Story

</PixelButton>

<PixelButton href="/community" variant="secondary">

<PixelIcon name="community" />

Community Hub

</PixelButton>

</div>

</div>

</header>

{#if $user && savedStories.length > 0}

<section class="story-section">

<h2>Continue Your Adventures</h2>

{#if isLoadingSaved}

<div class="loading-spinner">Loading your stories...</div>

{:else}

<div class="story-grid">

{#each savedStories as story}

<StoryCard

{story}

on:click={() => continueStory(

story.id,

story.progress?.currentSegmentId

)}

/>

{/each}

</div>

{/if}

</section>

{/if}

<section class="story-section">

<h2>Popular Adventures</h2>

{#if isLoadingPopular}

<div class="loading-spinner">Loading popular stories...</div>

{:else if popularStories.length > 0}

<div class="story-grid">

{#each popularStories as story}

<StoryCard

{story}

on:click={() => continueStory(story.id)}

/>

{/each}

</div>

{:else}

<p class="empty-state">No popular stories yet. Be the first to create one!</p>

{/if}

<div class="section-footer">

<PixelButton href="/community" variant="secondary" size="small">

See All

</PixelButton>

</div>

</section>

{#if $user && recommendedStories.length > 0}

<section class="story-section">

<h2>Recommended For You</h2>

{#if isLoadingRecommended}

<div class="loading-spinner">Finding stories just for you...</div>

{:else}

<div class="story-grid">

{#each recommendedStories as story}

<StoryCard

{story}

on:click={() => continueStory(story.id)}

/>

{/each}

</div>

{/if}

</section>

{/if}

<section class="features-section">

<h2>TaleForge Features</h2>

<div class="features-grid">

<PixelCard>

<h3>AI-Generated Stories</h3>

<div class="feature-icon">

<PixelIcon name="ai" size="large" />

</div>

<p>Each adventure is uniquely crafted based on your choices</p>

</PixelCard>

<PixelCard>

<h3>Pixel Art Visuals</h3>

<div class="feature-icon">

<PixelIcon name="art" size="large" />

</div>

<p>Beautiful retro-style artwork brings your story to life</p>

</PixelCard>

<PixelCard>

<h3>Voice Narration</h3>

<div class="feature-icon">

<PixelIcon name="audio" size="large" />

</div>

<p>Immersive audio narration enhances the storytelling experience</p>

</PixelCard>

</div>

</section>

</div>

{#if showLoginPrompt}

<LoginPrompt

on:success={handleLoginSuccess}

on:cancel={handleLoginCancel}

/>

{/if}

<style>

.home-page {

display: flex;

flex-direction: column;

gap: 3rem;

}

.hero {

display: flex;

flex-direction: column;

align-items: center;

justify-content: center;

text-align: center;

min-height: 60vh;

padding: 2rem;

background: linear-gradient(rgba(31, 31, 61, 0.8), rgba(31, 31, 61, 0.9)),

url('/hero-background.png');

background-size: cover;

background-position: center;

image-rendering: pixelated;

margin: -1rem;

margin-bottom: 1rem;

}

.hero-content {

max-width: 800px;

}

.logo {

width: 250px;

margin-bottom: 1rem;

image-rendering: pixelated;

}

h1 {

font-family: 'Press Start 2P', cursive;

font-size: 2rem;

margin-bottom: 1rem;

color: var(--accent-color);

text-shadow: 3px 3px 0px rgba(0, 0, 0, 0.5);

}

.hero p {

font-size: 1.5rem;

margin-bottom: 2rem;

max-width: 600px;

margin-left: auto;

margin-right: auto;

}

.hero-buttons {

display: flex;

gap: 1rem;

justify-content: center;

flex-wrap: wrap;

}

.story-section {

margin-bottom: 2rem;

}

h2 {

font-family: 'Press Start 2P', cursive;

font-size: 1.5rem;

margin-bottom: 1.5rem;

color: var(--secondary-color);

}

.story-grid {

display: grid;

grid-template-columns: repeat(auto-fill, minmax(300px, 1fr));

gap: 1.5rem;

}

.section-footer {

display: flex;

justify-content: center;

margin-top: 1.5rem;

}

.loading-spinner {

text-align: center;

padding: 2rem;

font-family: 'Press Start 2P', cursive;

font-size: 0.8rem;

}

.empty-state {

text-align: center;

padding: 2rem;

font-style: italic;

}

.features-grid {

display: grid;

grid-template-columns: repeat(auto-fit, minmax(250px, 1fr));

gap: 1.5rem;

}

.feature-icon {

display: flex;

justify-content: center;

margin: 1.5rem 0;

}

h3 {

font-family: 'Press Start 2P', cursive;

font-size: 1rem;

text-align: center;

margin-bottom: 1rem;

}

@media (max-width: 768px) {

h1 {

font-size: 1.5rem;

}

.hero p {

font-size: 1.2rem;

}

.hero-buttons {

flex-direction: column;

}

.story-grid {

grid-template-columns: 1fr;

}

}

</style>

Finally, let's implement the StoryPlay component, which is the core of the TaleForge experience:

<!-- frontend/src/pages/StoryPlay.svelte -->

<script>

import { onMount, onDestroy } from 'svelte';

import { navigate } from 'svelte-routing';

import { user } from '../stores/user';

import firebase from '../services/firebase';

import offlineStorageService from '../services/offlineStorageService';

import audioProcessor from '../services/audioProcessor';

import achievementSystem from '../services/achievementSystem';

import PixelImage from '../components/common/PixelImage.svelte';

import ChoiceButton from '../components/story/ChoiceButton.svelte';

import AudioControls from '../components/story/AudioControls.svelte';

import StoryControls from '../components/story/StoryControls.svelte';

import ShareDialog from '../components/dialogs/ShareDialog.svelte';

import SaveDialog from '../components/dialogs/SaveDialog.svelte';

import LoginPrompt from '../components/auth/LoginPrompt.svelte';

import LoadingAnimation from '../components/common/LoadingAnimation.svelte';

// Props

export let storyId;

export let segmentId = null;

// Story data

let story = null;

let segment = null;

let userProgress = null;

// UI state

let isLoading = true;

let isGenerating = false;

let audioPlayer = null;

let isAudioPlaying = false;

let isAudioLoaded = false;

let showShareDialog = false;

let showSaveDialog = false;

let showLoginPrompt = false;

let error = null;

let choiceHistory = [];

// Analytics

let startTime = null;

let choicesMade = 0;

onMount(async () => {

startTime = Date.now();

try {

// Load story

await loadStory();

// If segmentId is provided, load that specific segment

// Otherwise load initial or current segment

if (segmentId) {

await loadSegment(segmentId);

} else {

await loadCurrentSegment();

}

} catch (e) {

console.error('Error loading story:', e);

error = 'Failed to load story. Please try again.';

}

});

onDestroy(() => {

// Stop audio

if (audioPlayer) {

audioPlayer.stop();

}

// Save analytics

saveAnalytics();

});

async function loadStory() {

try {

// Try offline storage first

let offlineStory = null;

try {

offlineStory = await offlineStorageService.getStory(storyId);

} catch (error) {

console.log('No offline story available');

}

if (offlineStory) {

story = offlineStory;

} else {

// Fetch from Firebase

const storyDoc = await firebase.firestore()

.collection('stories')

.doc(storyId)

.get();

if (!storyDoc.exists) {

throw new Error('Story not found');

}

story = {

id: storyDoc.id,

...storyDoc.data()

};

// Cache for offline use

await offlineStorageService.saveStory(story);

}

// Load user progress if logged in

if ($user) {

await loadUserProgress();

}

} catch (error) {

console.error('Error loading story:', error);

throw error;

}

}

async function loadUserProgress() {

try {

// Try offline storage first

let offlineProgress = null;

try {

offlineProgress = await offlineStorageService.getUserProgress($user.id, storyId);

} catch (error) {

console.log('No offline progress available');

}

if (offlineProgress) {

userProgress = offlineProgress;

} else {

// Fetch from Firebase

const progressQuery = await firebase.firestore()

.collection('userProgress')

.where('userId', '==', $user.id)

.where('storyId', '==', storyId)

.limit(1)

.get();

if (!progressQuery.empty) {

userProgress = {

id: progressQuery.docs[0].id,

...progressQuery.docs[0].data()

};

// Cache progress

await offlineStorageService.saveUserProgress(userProgress);

// Load choice history

if (userProgress.choicePath && userProgress.choicePath.length > 0) {

choiceHistory = userProgress.choicePath;

}

}

}

} catch (error) {

console.error('Error loading user progress:', error);

}

}

async function loadCurrentSegment() {

// If user has progress, load current segment

if (userProgress && userProgress.currentSegmentId) {

await loadSegment(userProgress.currentSegmentId);

} else {

// Otherwise load initial segment

await loadInitialSegment();

}

}

async function loadInitialSegment() {

try {

// Find initial segment with no parent

const segmentsSnapshot = await firebase.firestore()

.collection('storySegments')

.where('storyId', '==', storyId)

.where('parentSegmentId', '==', null)

.limit(1)

.get();

if (segmentsSnapshot.empty) {

// No initial segment exists, generate one

await generateInitialSegment();

} else {

const segmentDoc = segmentsSnapshot.docs[0];

segment = {

id: segmentDoc.id,

...segmentDoc.data()

};

// Cache segment

await offlineStorageService.saveSegment(segment);

// Load audio

await loadAudio();

}

} catch (error) {

console.error('Error loading initial segment:', error);

throw error;

} finally {

isLoading = false;

}

}

async function loadSegment(id) {

try {

isLoading = true;

// Try offline storage first

let offlineSegment = null;

try {

offlineSegment = await offlineStorageService.getSegment(id);

} catch (error) {

console.log('No offline segment available');

}

if (offlineSegment) {

segment = offlineSegment;

} else {

// Fetch from Firebase

const segmentDoc = await firebase.firestore()

.collection('storySegments')

.doc(id)

.get();

if (!segmentDoc.exists) {

throw new Error('Segment not found');

}

segment = {

id: segmentDoc.id,

...segmentDoc.data()

};

// Cache segment

await offlineStorageService.saveSegment(segment);

}

// Load audio

await loadAudio();

} catch (error) {

console.error('Error loading segment:', error);

throw error;

} finally {

isLoading = false;

}

}

async function loadAudio() {

try {

if (segment.audioUrl) {

// Check for cached asset

const cachedAsset = await offlineStorageService.getAsset(segment.audioUrl);

if (cachedAsset) {

// Use cached audio

isAudioLoaded = true;

} else {

// Preload audio

const audioResult = await audioProcessor.preloadAudio(segment.audioUrl);

if (audioResult.success) {

// Cache audio for offline use

await offlineStorageService.saveAsset(

segment.audioUrl,

audioResult.data,

'audio',

storyId

);

}

isAudioLoaded = true;

}

} else {

// No audio for this segment

isAudioLoaded = true;

}

} catch (error) {

console.error('Error loading audio:', error);

// Continue without audio

isAudioLoaded = true;

}

}

async function generateInitialSegment() {

try {

isGenerating = true;

// Call API to generate initial segment

const response = await fetch('/api/stories/${storyId}/segments/initial', {

method: 'POST',

headers: {

'Content-Type': 'application/json'

},

body: JSON.stringify({

userId: $user?.id || 'anonymous'

})

});

if (!response.ok) {

throw new Error('Failed to generate initial segment');

}

const data = await response.json();

segment = data.segment;

// Cache segment

await offlineStorageService.saveSegment(segment);

// Load audio

await loadAudio();

} catch (error) {

console.error('Error generating initial segment:', error);

Copy

please continue

I'll continue with the StoryPlay component implementation:

<!-- frontend/src/pages/StoryPlay.svelte (continued) -->

<script>

// continued from above

async function generateInitialSegment() {

try {

isGenerating = true;

// Call API to generate initial segment

const response = await fetch(`/api/stories/${storyId}/segments/initial`, {

method: 'POST',

headers: {

'Content-Type': 'application/json'

},

body: JSON.stringify({

userId: $user?.id || 'anonymous'

})

});

if (!response.ok) {

throw new Error('Failed to generate initial segment');

}

const data = await response.json();

segment = data.segment;

// Cache segment

await offlineStorageService.saveSegment(segment);

// Load audio

await loadAudio();

} catch (error) {

console.error('Error generating initial segment:', error);

throw error;

} finally {

isGenerating = false;

}

}

async function makeChoice(choiceIndex) {

if (isGenerating) return;

try {

isGenerating = true;

// Store the choice text for history

const choiceText = segment.choices[choiceIndex].text;

choiceHistory = [...choiceHistory, choiceText];

choicesMade++;

// Check if this choice already has a target segment

if (segment.choices[choiceIndex].targetSegmentId) {

// Load existing segment

const nextSegmentId = segment.choices[choiceIndex].targetSegmentId;

await loadSegment(nextSegmentId);

// Update user progress

await updateUserProgress(nextSegmentId);

return;

}

// Generate new segment based on choice

const response = await fetch(`/api/stories/${storyId}/segments`, {

method: 'POST',

headers: {

'Content-Type': 'application/json'

},

body: JSON.stringify({

userId: $user?.id || 'anonymous',

parentSegmentId: segment.id,

choiceIndex

})

});

if (!response.ok) {

throw new Error('Failed to generate next segment');

}

const data = await response.json();

// Update the choice in the current segment

const updatedChoices = [...segment.choices];

updatedChoices[choiceIndex] = {

...updatedChoices[choiceIndex],

targetSegmentId: data.segment.id

};

const updatedSegment = {

...segment,

choices: updatedChoices

};

// Cache updated current segment

await offlineStorageService.saveSegment(updatedSegment);

// Load the new segment

segment = data.segment;

// Cache new segment

await offlineStorageService.saveSegment(segment);

// Load audio

await loadAudio();

// Update user progress

await updateUserProgress(segment.id);

// Check for achievements if this is an ending

if (segment.isEnding) {

if ($user) {

await achievementSystem.checkAchievements($user.id);

}

}

} catch (error) {

console.error('Error making choice:', error);

error = 'Failed to continue the story. Please try again.';

} finally {

isGenerating = false;

}

}

async function updateUserProgress(segmentId) {

if (!$user) return;

try {

// Check if progress exists

if (!userProgress) {

// Create new progress

userProgress = {

userId: $user.id,

storyId,

currentSegmentId: segmentId,

choicePath: choiceHistory,

choicesMade,

startedAt: firebase.firestore.FieldValue.serverTimestamp(),

updatedAt: firebase.firestore.FieldValue.serverTimestamp()

};

// Save to Firebase

const docRef = await firebase.firestore()

.collection('userProgress')

.add(userProgress);

userProgress.id = docRef.id;

} else {

// Update existing progress

const updates = {

currentSegmentId: segmentId,

choicePath: choiceHistory,

choicesMade: (userProgress.choicesMade || 0) + 1,

updatedAt: firebase.firestore.FieldValue.serverTimestamp()

};

// Check if this is an ending

if (segment.isEnding) {

updates.isCompleted = true;

updates.completedAt = firebase.firestore.FieldValue.serverTimestamp();

// Calculate completion time

const completionTimeMs = Date.now() - startTime;

const completionTimeSeconds = Math.floor(completionTimeMs / 1000);

updates.completionTimeSeconds = completionTimeSeconds;

}

// Update Firebase

await firebase.firestore()

.collection('userProgress')

.doc(userProgress.id)

.update(updates);

// Update local object

userProgress = {

...userProgress,

...updates

};

}

// Cache progress

await offlineStorageService.saveUserProgress(userProgress);

} catch (error) {

console.error('Error updating progress:', error);

// Store in offline sync queue

try {

await offlineStorageService.addToSyncQueue({

type: 'saveProgress',

data: {

id: userProgress?.id,

userId: $user.id,

storyId,

currentSegmentId: segmentId,

choicePath: choiceHistory,

choicesMade

}

});

} catch (syncError) {

console.error('Error adding to sync queue:', syncError);

}

}

}

function playAudio() {

if (!segment.audioUrl || !isAudioLoaded) return;

if (audioPlayer) {

audioPlayer.stop();

}

audioProcessor.playAudio(segment.audioUrl, {

volume: 0.8,

speed: 1.0

}).then(player => {

audioPlayer = player;

isAudioPlaying = true;

// Add event listener for when audio ends

player.source.onended = () => {

isAudioPlaying = false;

};

}).catch(error => {

console.error('Error playing audio:', error);

isAudioPlaying = false;

});

}

function toggleAudio() {

if (isAudioPlaying) {

audioPlayer.stop();

isAudioPlaying = false;

} else {

playAudio();

}

}

function saveAnalytics() {

if (!$user) return;

const sessionDuration = Math.floor((Date.now() - startTime) / 1000);

try {

firebase.firestore()

.collection('analytics')

.add({

userId: $user.id,

storyId,

sessionDuration,

choicesMade,

timestamp: firebase.firestore.FieldValue.serverTimestamp()

});

} catch (error) {

console.error('Error saving analytics:', error);

}

}

function openShareDialog() {

if ($user) {

showShareDialog = true;

} else {

showLoginPrompt = true;

}

}

function openSaveDialog() {

if ($user) {

showSaveDialog = true;

} else {

showLoginPrompt = true;

}

}

function handleLoginSuccess() {

showLoginPrompt = false;

// After login, attempt to save progress

updateUserProgress(segment.id);

}

function handleLoginCancel() {

showLoginPrompt = false;

}

function handleShareComplete() {

showShareDialog = false;

}

function handleSaveComplete() {

showSaveDialog = false;

}

</script>

<div class="story-play-container">

{#if isLoading}

<div class="loading-container">

<LoadingAnimation />

<p>Loading your adventure...</p>

</div>

{:else if error}

<div class="error-container">

<p class="error-message">{error}</p>

<PixelButton on:click={() => navigate('/')}>

Return Home

</PixelButton>

</div>

{:else}

<div class="story-header">

<h1>{story.title || `${story.genre} Adventure`}</h1>

{#if choiceHistory.length > 0}

<div class="choice-path">

<span class="path-label">Your path:</span>

{#each choiceHistory.slice(-2) as choice, i}

<span class="path-choice">{i > 0 ? ' → ' : ''}{choice}</span>

{/each}

</div>

{/if}

</div>

<div class="story-content">

<div class="image-container">

<PixelImage

src={segment.imageUrl}

alt="Story scene"

loading="eager"

pixelated

/>

</div>

<div class="text-container">

<p class="story-text">{segment.text}</p>

<AudioControls

isPlaying={isAudioPlaying}

isLoaded={isAudioLoaded}

on:toggle={toggleAudio}

/>

</div>

</div>

<div class="choices-container">

{#if isGenerating}

<div class="generating">

<LoadingAnimation />

<p>Forging the next part of your story...</p>

</div>

{:else if segment.isEnding}

<div class="ending-message">

<h2>The End</h2>

<p>You've reached the conclusion of this adventure!</p>

<div class="ending-buttons">

<PixelButton on:click={() => navigate('/stories/select')}>

New Adventure

</PixelButton>

<PixelButton variant="secondary" on:click={openShareDialog}>

Share Story

</PixelButton>

</div>

</div>

{:else}

<div class="choices">

{#each segment.choices as choice, i}

<ChoiceButton

text={choice.text}

on:click={() => makeChoice(i)}

disabled={isGenerating}

/>

{/each}

</div>

{/if}

</div>

<StoryControls

on:share={openShareDialog}

on:save={openSaveDialog}

showShare={!segment.isEnding}

/>

{/if}

</div>

{#if showShareDialog}

<ShareDialog

{story}

{segment}

{choiceHistory}

on:close={handleShareComplete}

/>

{/if}

{#if showSaveDialog}

<SaveDialog

{story}

{segment}

on:close={handleSaveComplete}

/>

{/if}

{#if showLoginPrompt}

<LoginPrompt

on:success={handleLoginSuccess}

on:cancel={handleLoginCancel}

/>

{/if}

<style>

.story-play-container {

display: flex;

flex-direction: column;

gap: 1.5rem;

max-width: 900px;

margin: 0 auto;

padding: 1rem;

min-height: 80vh;

}

.loading-container, .error-container {

display: flex;

flex-direction: column;

align-items: center;

justify-content: center;

gap: 1.5rem;

margin: 4rem auto;

text-align: center;

}

.error-message {

color: var(--error-color);

font-size: 1.2rem;

margin-bottom: 1rem;

}

.story-header {

text-align: center;

margin-bottom: 1rem;

}

h1 {

font-family: 'Press Start 2P', cursive;

font-size: 1.5rem;

margin-bottom: 0.5rem;

color: var(--accent-color);

}

.choice-path {

font-size: 0.9rem;

opacity: 0.8;

max-width: 80%;

margin: 0 auto;

white-space: nowrap;

overflow: hidden;

text-overflow: ellipsis;

}

.path-label {

font-weight: bold;

}

.path-choice {

font-style: italic;

}

.story-content {

display: grid;

grid-template-columns: 1fr 1fr;

gap: 1.5rem;

margin-bottom: 1.5rem;

}

.image-container {

position: relative;

aspect-ratio: 1;

width: 100%;

border: 2px solid var(--secondary-color);

border-radius: 4px;

overflow: hidden;

}

.text-container {

display: flex;

flex-direction: column;

justify-content: space-between;

background-color: rgba(0, 0, 0, 0.3);

border: 2px solid var(--border-color);

border-radius: 4px;

padding: 1rem;

}

.story-text {

font-size: 1.2rem;

line-height: 1.6;

margin-bottom: 1.5rem;

}

.choices-container {

margin: 1.5rem 0;

}

.choices {

display: flex;

flex-direction: column;

gap: 0.75rem;

}

.generating {

display: flex;

flex-direction: column;

align-items: center;

justify-content: center;

gap: 1rem;

padding: 2rem;

text-align: center;

background-color: rgba(0, 0, 0, 0.3);

border-radius: 4px;

}

.ending-message {

text-align: center;

padding: 2rem;

background-color: rgba(0, 0, 0, 0.3);

border-radius: 4px;

border: 2px solid var(--accent-color);

}

.ending-message h2 {

font-family: 'Press Start 2P', cursive;

font-size: 1.5rem;

margin-bottom: 1.5rem;

color: var(--accent-color);

}

.ending-buttons {

display: flex;

gap: 1rem;

justify-content: center;

margin-top: 2rem;

}

@media (max-width: 768px) {

.story-content {

grid-template-columns: 1fr;

gap: 1rem;

}

.image-container {

max-width: 400px;

margin: 0 auto;

}

h1 {

font-size: 1.2rem;

}

.ending-buttons {

flex-direction: column;

align-items: center;

}

}

</style>

**Deployment Configuration**

Let's set up the deployment configuration files needed to deploy TaleForge:

**Netlify Configuration (frontend/netlify.toml)**

# netlify.toml

[build]

command = "npm run build"

publish = "dist"

functions = "netlify/functions"

[build.environment]

NODE\_VERSION = "16"

[[redirects]]

from = "/api/\*"

to = "https://api.taleforge.com/api/:splat"

status = 200

force = true

[[redirects]]

from = "/\*"

to = "/index.html"

status = 200

[dev]

framework = "vite"

targetPort = 5173

port = 8888

[[headers]]

for = "/\*"

[headers.values]

Cache-Control = "public, max-age=0, must-revalidate"

X-Frame-Options = "DENY"

X-XSS-Protection = "1; mode=block"

X-Content-Type-Options = "nosniff"

Referrer-Policy = "strict-origin-when-cross-origin"

Content-Security-Policy = "default-src 'self' https://\*.taleforge.com https://\*.firebase.com https://\*.firebaseio.com https://\*.googleapis.com; style-src 'self' 'unsafe-inline' https://fonts.googleapis.com; font-src 'self' https://fonts.gstatic.com; img-src 'self' data: https://\*.taleforge.com https://storage.googleapis.com; media-src 'self' https://\*.taleforge.com https://storage.googleapis.com; connect-src 'self' https://\*.taleforge.com wss://\*.taleforge.com https://\*.firebase.com https://\*.firebaseio.com https://\*.googleapis.com; frame-src 'self' https://\*.taleforge.com https://\*.firebase.com;"

**Docker Configuration (backend/Dockerfile)**

# backend/Dockerfile

FROM node:16-alpine

# Create app directory

WORKDIR /usr/src/app

# Install app dependencies

COPY package\*.json ./

RUN npm ci --only=production

# Copy app source

COPY . .

# Set environment variables

ENV NODE\_ENV=production

ENV PORT=3000

# Expose port

EXPOSE 3000

# Start the server

CMD [ "npm", "start" ]

**Docker Compose for Backend Services**

# docker-compose.yml

version: '3.8'

services:

api:

build:

context: ./backend

dockerfile: Dockerfile

restart: always

ports:

- "3000:3000"

environment:

- NODE\_ENV=production

- PORT=3000

- FIREBASE\_SERVICE\_ACCOUNT=/run/secrets/firebase\_service\_account

- ELEVENLABS\_API\_KEY=/run/secrets/elevenlabs\_api\_key

- STABLE\_DIFFUSION\_HOST=http://stable-diffusion:5000

secrets:

- firebase\_service\_account

- elevenlabs\_api\_key

depends\_on:

- ollama

- stable-diffusion

networks:

- taleforge-network

ollama:

image: ollama/ollama:latest

restart: always

ports:

- "11434:11434"

volumes:

- ollama-data:/root/.ollama

networks:

- taleforge-network

stable-diffusion:

image: taleforge/stable-diffusion-api:latest

restart: always

ports:

- "5000:5000"

volumes:

- sd-models:/models

deploy:

resources:

reservations:

devices:

- driver: nvidia

count: 1

capabilities: [gpu]

networks:

- taleforge-network

networks:

taleforge-network:

driver: bridge

volumes:

ollama-data:

sd-models:

secrets:

firebase\_service\_account:

file: ./secrets/firebase-service-account.json

elevenlabs\_api\_key:

file: ./secrets/elevenlabs-api-key.txt

**Conclusion**

TaleForge is a sophisticated interactive storytelling platform that leverages multiple AI technologies to create immersive, branching narratives with pixel-art visuals and audio narration. The implementation described in this document covers all aspects of the project, from planning to deployment.

**Key Technical Achievements**

1. **Multi-Modal AI Integration**: TaleForge demonstrates seamless integration of three different AI technologies (text generation, image generation, and speech synthesis) to create a cohesive user experience.
2. **Advanced Data Architecture**: The system uses a sophisticated database schema to support complex branching narratives while maintaining contextual consistency.
3. **Progressive Web App Implementation**: The application works cross-platform with offline capabilities, push notifications, and installability.
4. **Real-time Community Features**: Firebase integration enables real-time updates to the community hub, voting system, and user interactions.
5. **Adaptive Difficulty**: The platform analyzes user behavior to adjust story complexity, keeping the experience engaging for different user preferences.
6. **Performance Optimizations**: WebAssembly for audio processing, efficient asset caching, and comprehensive offline support ensure a smooth experience even on mobile devices.

The TaleForge platform is not only a demonstration of technical skill but also a genuinely engaging product with strong monetization potential through its freemium model. By leveraging AI for content generation, the platform can scale efficiently while maintaining high-quality storytelling experiences.

As a portfolio piece, TaleForge showcases the ability to architect complex systems, integrate cutting-edge technologies, and deliver a polished user experience across multiple platforms.

Copy

Personalize