————————————————————

- 启动动画：展示文字如丝线在织机上逐渐成布的过程。
- 参数控件：用"纺锤、丝线、梭子"等图标替代传统滑块。

核心功能
智能故事生成
- 6 种主流文学类型（奇幻、科幻、悬疑、爱情、恐怖、冒险）
- 多维度参数控制（主角、背景、冲突、语调、长度、视角）
- 实时生成历史叙述内容
个性化定制
- 8 种不同语调风格
- 灵活的故事长度选项
- 第一/第三人称视角切换
- 自定义主题和特殊要求
便捷的操作体验
- 初始故事模板快速启动
- 一键复制/导出功能
- 实时预览和编辑
- 响应式设计车载各种设备

设计亮点
- 视觉冲击力：深度主题配合升级，神秘创意创意
- 交互动效：加载动画、暂停效果、状态反馈
- 信息架构：清晰的标签导航和分步操作流程
- 细节体验：浮动操作栏、智能提示、指示状态

当前技术路径
根据您提供的文件 narrative-generator.tsx，其技术路径分析如下：
技术路径：
- 前端： React + TypeScript + Tailwind CSS
- 后端： 无。该文件是一个纯前端应用，故事生成是通过本地的 setTimeout 模拟异步请求，没有实际的网络调用。
- 核心引擎： 本地 JavaScript 逻辑模拟。文件通过 generateSampleStory 函数拼接预设的字符串来模拟 AI 生成故事，并非调用真实的 AI 模型。
- 状态管理： React Hooks (useState, useEffect)。
- 本地存储： 无。文件没有使用 localStorage 或 IndexedDB 进行数据持久化存储。它使用了 navigator.clipboard API 实现复制功能，以及通过 Blob 和 URL.createObjectURL 实现文本下载功能。

我想集成真正的 AI API（如 deepseek），添加故事续写功能，实际用户历史记录，支持多人协作创作，下面是实现这些功能所需的步骤、技术路径、组件划分和基本的交互逻辑。

## 第一阶段：集成真实 AI 与实现用户历史记录（单用户模式）

这是从模拟工具到真正产品的核心一步，为后续功能打下基础。

### 技术路径

- 

**前端 (Frontend):**

- 
  - **库/框架:** 保持 **React + TypeScript + Tailwind CSS**。
  - **数据请求:** 使用 `axios` 或 `fetch` API 来与后端通信。
  - **状态管理:** 对于更复杂的跨组件状态（如用户信息），可以引入 **Zustand** 或 **Redux Toolkit**，但对于此阶段，React Context 结合 `useState` 可能依然足够。
- 

**后端 (Backend):**

- 
  - **框架:** 使用 **Node.js + Express** 或 **NestJS** (一个更结构化的 Node.js 框架)。
  - **API 设计:** 设计成 **RESTful API**。例如 `POST /api/stories/generate`，`GET /api/stories`。
  - **安全:** API 密钥（如 DeepSeek 的 key）**绝不能**放在前端。后端服务器作为安全代理，接收前端请求，附加密钥后再去调用 AI API。
  - **数据库:**
    - **SQL: PostgreSQL** 是一个稳健的选择，适合结构化数据。
    - **NoSQL: MongoDB** 也很合适，其文档模型可以轻松存储故事及其相关设置。
  - **用户认证:** 使用 **Passport.js + JWT (JSON Web Tokens)**。用户登录后，前端保存 JWT，并在后续请求的 Header 中携带它，以便后端识别用户身份。
- 

**核心引擎 (AI Integration):**

- 

- **API 调用:** 在后端，使用 `axios` 或 `node-fetch` 调用 **DeepSeek API** 的聊天或文本补全接口。
- **Prompt 工程:** 设计更精细的 Prompt。例如，对于故事续写，你需要将“现有故事内容”和“续写指令”组合成一个新的 Prompt。

### 实现步骤

1.

**搭建后端服务:**

2.

- 创建一个 Node.js 项目。
- 设置 Express 路由，如 `/api/auth` (登录/注册), `/api/stories` (获取历史), `/api/stories/generate` (生成故事)。
- 在 `/generate` 路由中，接收前端传来的表单数据，构造 Prompt，然后调用 DeepSeek API。将 AI 的返回结果再传回给前端。

3.

**建立数据库模型:**

4.

- 创建 `users` 表：`id`, `username`, `password_hash`。
- 创建 `stories` 表：`id`, `user_id` (外键关联用户), `title`, `content` (存储完整故事), `generation_settings` (JSON 格式存储生成时的参数), `created_at`。

5.

**改造前端:**

6.

- 修改 `generateStory` 函数，将原来的模拟逻辑替换为对后端 `/api/stories/generate` 的 `axios.post` 请求。
- 创建登录/注册页面和相关的认证逻辑。
- 创建一个新的“仪表盘”或“我的故事”页面，通过调用 `/api/stories` 获取并展示用户的历史记录。

---

## 第二阶段：添加故事续写与多人协作功能

这是产品从单用户工具走向协作平台的关键。

## 技术路径

- 

### 前端 (Frontend):

- 

  - **实时通信:** 引入 `socket.io-client` 库来处理与后端的 WebSocket 连接。
  - **文本编辑器:** 考虑使用更专业的富文本编辑器库，如 **Tiptap** 或 **Slate.js**，它们能更好地处理复杂编辑状态和多人协作光标。

- 

### 后端 (Backend):

- 

  - **实时通信:** 在 Express 服务器上集成 **Socket.io**，用于处理实时的数据广播。
  - **数据库模型扩展:**

    - 需要一个中间表 `story_collaborators` 来管理故事和用户之间的多对多关系 (`story_id`, `user_id`, `role` [如 owner, editor])。

  - **冲突处理 (高级):** 对于多人实时编辑，简单的"最后写入者获胜"会产生问题。专业解决方案通常采用：

    - **Operational Transformation (OT):** 如 Google Docs 使用的算法，转换操作以解决并发冲突。
    - **CRDTs (Conflict-free Replicated Data Types):** 一种数据结构，天生能无冲突地合并并发修改。

- 

### 核心引擎 (功能扩展):

- 

    - **续写 Prompt:** 设计专门的 Prompt 格式，例如：

```
你是一个故事续写助手。这是故事的现有部分：
"{story_content}"

请根据以下要求继续写下去：
"{user_continuation_prompt}"
```

- 

**实现步骤**

1. 

**实现故事续写:**

2. 

   - **后端:** 创建新 API 路由 `POST /api/stories/:id/continue`。它会获取已有故事内容，结合用户的续写指令，调用 AI API，然后将返回的新段落追加到数据库的 `stories.content` 字段。
   - **前端:** 在故事结果页面添加一个“续写”按钮和输入框。点击后调用上述 API，并实时更新页面上的故事内容。

3. 

**实现多人协作基础:**

4. 

   - **后端:**

     - 当用户打开一个故事编辑页时，后端将其加入一个以 `story_id` 命名的 Socket.io "房间" (room)。
     - 当一个用户在编辑器中输入内容时，前端通过 WebSocket 发送一个事件（如 `text_change`）及内容到后端。
     - 后端收到该事件后，将其广播给房间内所有其他用户。
     - 同时，后端可以定期或在用户停止输入时，将最新的内容保存到数据库。

   - **前端:**

     - 进入编辑器页面时，连接到 Socket.io 服务器并加入对应的房间。
     - 监听来自服务器的 `text_change` 事件，接收到后更新本地编辑器的内容。
     - 当本地用户输入时，向服务器发送 `text_change` 事件。

## 组件划分建议

你可以将原来的 `NarrativeGenerator.tsx` 文件拆分成更小、更专注的组件。

```
/src
|-- /pages
```

```
|   |-- HomePage.tsx        # 网站首页
|   |-- EditorPage.tsx      # 核心创作页面，包含设置和编辑器
|   |-- DashboardPage.tsx   # 用户历史记录仪表盘
|   |-- LoginPage.tsx       # 登录/注册页
|
|-- /components
|   |-- /editor
|   |   |-- Editor.tsx          # 核心文本编辑器区域 (集成 Tiptap/Slate)
|   |   |-- SettingsPanel.tsx   # 故事生成/续写设置的侧边栏
|   |   |-- CollaboratorAvatars.tsx # 显示当前协作者头像的组件
|   |
|   |-- /dashboard
|   |   |-- StoryList.tsx       # 历史故事列表
|   |   |-- StoryListItem.tsx   # 列表中的单个故事项
|   |
|   |-- /common
|   |   |-- Header.tsx          # 全局页头，包含用户信息和导航
|   |   |-- Button.tsx          # 通用按钮
|   |   |-- LoadingSpinner.tsx  # 加载动画
|
|-- /hooks                  # 自定义 Hooks
|   |-- useAuth.ts          # 管理用户认证状态
|   |-- useSocket.ts        # 封装 Socket.io 连接和事件
|
|-- /services               # API 请求服务
|   |-- api.ts              # 配置 axios 实例
|   |-- storyService.ts     # 封装所有与故事相关的 API 调用
```

# 基本交互逻辑

## 1. 故事续写交互逻辑

1. **用户** 在 `DashboardPage` 点击一个已存在的故事，进入 `EditorPage`。
2. `EditorPage` 加载故事内容并展示在 `Editor` 组件中。
3. **用户** 在 `SettingsPanel` 中切换到“续写”模式。
4. **用户** 在续写提示框中输入“主角发现了一个神秘的地图”。
5. **用户** 点击“生成续写”按钮。
6. **前端** 调用 `storyService.continueStory(storyId, "主角发现了一个神秘的地图")`。
7. **后端** 收到请求，构造 Prompt 调用 DeepSeek API。
8. **AI** 返回续写内容。
9. **后端** 将新内容追加到数据库，并通过 WebSocket (或直接在 API 响应中) 将完整故事返回。
10. **前端** 的 `Editor` 组件内容被更新，用户看到故事变长了。

## 2. 多人协作交互逻辑

1. **用户 A** 在 `DashboardPage` 创建一个新故事，并点击“分享”按钮，获得一个独特的链接。
2. **用户 A** 将链接发送给 **用户 B**。
3. **用户 B** 点击链接，如果未登录则跳转到 `LoginPage`，登录后进入 `EditorPage`，并自动加入该故事的协作房间。
4. `CollaboratorAvatars` 组件现在同时显示用户 A 和用户 B 的头像。
5. **用户 A** 在 `Editor` 中输入 "从前有座山..."。
6. **用户 A 的浏览器** 通过 WebSocket 向服务器发送 `text_change` 事件，并附带最新内容。
7. **服务器** 接收到事件，并将其广播给房间内的所有其他客户端。
8. **用户 B 的浏览器** 接收到 `text_change` 事件，并立即更新 `Editor` 组件的内容，实时看到了用户 A 输入的话。
9. 整个过程无缝进行，实现了共同创作。