Adventure Master - Manuale dello Sviluppatore

1. Introduzione

Autori: Lorenzo Guadagnini, Matteo Cardinale, Michela Pagliarani

Scopo del Manuale

Questo manuale fornisce una guida dettagliata per gli sviluppatori di **Adventure Master**, un'applicazione basata su Vue.js, Firebase e Aisuru. L'obiettivo è documentare l'architettura, il flusso di sviluppo e il funzionamento dei componenti chiave per agevolare la manutenzione e l'estensione del progetto.

Panoramica dell'Applicazione

Adventure Master è un'applicazione interattiva che permette agli utenti di vivere esperienze narrative dinamiche attraverso storie personalizzabili. Il sistema si integra con Aisuru, un agente conversazionale, per gestire l'interfaccia utente e la logica di interazione.

Tecnologie Principali

- **Vue.js**: Per la creazione dell'interfaccia utente e il montaggio dinamico dei componenti.
- **Firebase**: Per l'autenticazione, il database Firestore e lo storage dei file.
- **Aisuru**: Motore conversazionale che gestisce il flusso dell'applicazione attraverso contenuti interattivi.

---

2. Setup dell'Ambiente di Sviluppo

Requisiti Preliminari

Per iniziare lo sviluppo su **Adventure Master**, è necessario installare i seguenti strumenti:

- **Node.js** (versione consigliata: 16+)
- **npm** o **yarn** per la gestione delle dipendenze
- **Git** per il versionamento del codice
- Un editor di codice come **VS Code**

Clonazione del Repository

Per ottenere il codice sorgente, eseguire il seguente comando:

```
git clone https://github.com/CardiMatt/SWENG_2324.git
cd SWENG_2324/frontend
```

Installazione delle Dipendenze

Dopo aver clonato il repository, installare le dipendenze con:

```
npm install
```

Configurazione delle Variabili d'Ambiente

L'applicazione utilizza un file `.env` per la gestione delle variabili d'ambiente. Creare un file `.env` nella directory `frontend/` e inserire le seguenti chiavi:

```
//Aisuru
VITE_BASE_URL=https://engine.memori.ai
VITE_MEMORI_ID=5788e361-9bfa-4a03-a094-1d10c52a449d
VITE_TAG=👷
VITE_PIN=*******

//Firebase
VITE_API_KEY=**********************************
VITE_AUTH_DOMAIN=sweng-2324.firebaseapp.com
VITE_PROJECT_ID=sweng-2324
VITE_STORAGE_BUCKET=sweng-2324.firebasestorage.app
VITE_MESSAGING_SENDER_ID=619282552515
```

```
VITE_APP_ID=1:619282552515:web:0b74eb7387abf69ef63476
```

Avvio dell'Applicazione in Modalità Sviluppo

Per avviare l'applicazione in modalità sviluppo, eseguire:

```
npm run dev
```

L'app sarà accessibile all'indirizzo `http://localhost:5173/`.

3. Architettura dell'Applicazione

Panoramica Generale

L'architettura di **Adventure Master** è progettata per garantire modularità, manutenibilità e scalabilità. L'applicazione si basa su **Vue.js** per la gestione dell'interfaccia utente, **Firebase** per l'autenticazione e la persistenza dei dati, e **Aisuru** per il motore conversazionale che guida l'esperienza utente.

L'architettura è suddivisa in diversi moduli chiave:

- **Componenti Vue**: Definiscono l'interfaccia grafica e interagiscono con i servizi.
- **Servizi**: Il servizio principale, `AisuruService`, gestisce le comunicazioni con l' API di Aisuru e apre sessioni admin.
- **Repository**: Gestiscono il recupero e la persistenza dei dati.

Struttura delle Cartelle

L'organizzazione del codice è strutturata come segue:

```
frontend/
├── src/
|   ├── assets/             # Risorse statiche
|   ├── components/         # Componenti Vue
|   ├── models/             # Definizione dei modelli dati
```

```
|   ├── repositories/        # Strato di accesso ai dati
|   ├── services/            # Logica di business
|   ├── tests/               # Test per servizi e repository
|   ├── App.vue              # Entry point Vue
|   ├── main.ts              # Inizializzazione dell'applicazione
|   ├── exposeVueComponents.js # Montaggio dinamico dei componenti
|   ├── firebase.ts          # Configurazione Firebase
```

Componenti Vue e Montaggio Dinamico

Adventure Master utilizza componenti Vue montati dinamicamente per aggiornare l'interfaccia in base alle interazioni dell'utente con **Aisuru**. Il file `exposeVueComponents.js` gestisce questa funzionalità.

Esempio di montaggio di un componente:

```
function mountVueComponentsInChat(className, vueComponent) {
  const containers = document.querySelectorAll(`.${className}:not(.processe

  containers.forEach((container) => {
    if (!container.classList.contains('processed')) {
      createApp(vueComponent).mount(container);
      container.classList.add('processed');
      console.log(`Componente Vue montato in ${container}`);
    }
  });
}

function mountVueComponentsInExtention(vueComponent, props = {}) {
  const extension = document.getElementById("extension");

  // Creazione del div interno
  const innerDiv = document.createElement("div");
  innerDiv.style.width = "100%"; // Riempie il contenitore
  innerDiv.style.height = "auto"; // Altezza dinamica

  // Aggiungi il div interno al contenitore principale
  extension.appendChild(innerDiv);

  // Monta il componente Vue all'interno del div interno
  createApp(vueComponent, props).mount(innerDiv);
```

```
}
```

I componenti possono essere montati in due sezioni dell'applicazione:

- **Chat** (per interazioni dirette con l'utente)
- **Estensione UI** (per elementi persistenti nell'interfaccia)

Flusso di Comunicazione tra Aisuru e l'Applicazione

L'agente conversazionale Aisuru interagisce con l'app attraverso snippet javascript che vengono eseguiti in pagina.
Anche il montaggio dei component avviene grazie a questo meccanismo.

Esempio di comando ricevuto da Aisuru:

```
mountVueComponentsInChat("dynamic-container",Login)
```

Il sistema processa il comando e monta il componente corrispondente.

Gestione dei Dati Persistenti

L'app utilizza Firebase Firestore per la persistenza dei dati. I dati vengono gestiti attraverso repository che astraggono l'accesso al database.

Esempio di repository per il salvataggio di dati:

```
// src/repositories/LogRepository.ts
import { collection, addDoc, getDocs, query, where, getFirestore } from 'fi
import { db } from '../firebase';
import type { Log } from '../models/Log';

const logCollection = collection(db, 'logs');

export class LogRepository {
  static async saveLog(log: Omit<Log, 'id'>): Promise<string> {
    const docRef = await addDoc(logCollection, log);
    return docRef.id;
```

```
  }

  static async getLogsByUserId(userId: string): Promise<Log[]> {
    const logsQuery = query(logCollection, where('userId', '==', userId));
    const querySnapshot = await getDocs(logsQuery);
    return querySnapshot.docs.map((doc) => ({
      id: doc.id,
      ...doc.data(),
    })) as Log[];
  }

  static async getAllLogs(): Promise<Log[]> {
    const querySnapshot = await getDocs(logCollection);
    return querySnapshot.docs.map((doc) => ({
      id: doc.id,
      ...doc.data(),
    })) as Log[];
  }
}
```

Capitolo 4 - Dettaglio sui file

Panoramica

Il capitolo 4 fornisce una documentazione dettagliata dei file principali dell'applicazione Adventure Master. Ogni file viene descritto nel suo ruolo chiave, illustrando le sue funzionalità e la sua interazione con gli altri componenti. La documentazione include riferimenti ai metodi ed estratti di codice.

`main.ts` - Ingresso dell'Applicazione

Ruolo e Scopo

`main.ts` è il punto di ingresso dell'applicazione Vue. Il suo compito principale è:

- Inizializzare l'istanza dell'app Vue.
- Configurare e registrare plugin e librerie esterne.
- Definire interazioni globali con il sistema.
- Montare l'applicazione nel DOM.

Struttura e Funzionamento

Il file importa le risorse necessarie, configura i plugin e crea l'applicazione Vue:

```
import './assets/main.css';
import { createApp } from 'vue';
import App from './App.vue';

// Script per l'inserimento dei componenti
import './exposeVueComponents';

// Importa VueFire e il modulo di autenticazione
import { VueFire, VueFireAuth, VueFireFirestoreOptionsAPI } from 'vuefire';
import { firebaseApp } from './firebase';

// Bootstrap
import "bootstrap/dist/css/bootstrap.min.css";
import "bootstrap";

// Vuetify
import "vuetify/styles";
import { createVuetify } from "vuetify";
import * as components from "vuetify/components";
import * as directives from "vuetify/directives";

const vuetify = createVuetify({
    components,
    directives,
});

const app = createApp(App);

// Configura VueFire con Firebase
app.use(VueFire, {
  firebaseApp,
  modules: [
    VueFireFirestoreOptionsAPI(),
```

```
    VueFireAuth()
  ],
});

// Vuetify
app.use(vuetify);

// Monta l'app
app.mount('#app');
```

Funzionamento

1. **Importa risorse e librerie**: include CSS, Bootstrap, Vuetify e Firebase.
2. **Crea l'istanza dell'app Vue** con `createApp(App)`.
3. **Configura plugin e moduli**: abilita autenticazione e database Firestore tramite VueFire.
4. **Integra Vuetify**: definisce componenti e direttive UI.
5. **Monta l'app nel DOM**, associandola all'elemento `#app` in `index.html`.
6. **Definisce interazioni globali**: espone API nel `window` per comunicazione con Aisuru.

Interazioni Globali

Il file definisce un'interfaccia globale per l'interazione con il memori-client:

```
declare global {
  interface Window {
    typeMessage: (
      message: string,
      waitForPrevious?: boolean,
      hidden?: boolean,
      typingText?: string,
      useLoaderTextAsMsg?: boolean,
      hasBatchQueued?: boolean
    ) => void;

    typeBatchMessages: (
      messages: {
        message: string;
        waitForPrevious?: boolean;
        hidden?: boolean;
        typingText?: string;
```

```
        useLoaderTextAsMsg?: boolean;
      }[]
    ) => void;

    getMemoriState: (integrationId?: string) => object;
  }
}
```

I metodi esposti sono segnati come void o object generico ma ciò non è un problema poiché l'unico obbiettivo è quello di far rilevare i tre metodi al resto del progetto.
Essi saranno immediatamente sovrascritti dagli script di memori-client.
Ciò ci permette di interagire con memori-client anche usando typescript e non javascript.

`firebase.ts` - Configurazione di Firebase

Ruolo e Scopo

Il file `firebase.ts` gestisce l'inizializzazione e la configurazione di Firebase nell'applicazione Adventure Master. Fornisce l'accesso ai servizi di autenticazione, database Firestore e storage, centralizzando la gestione delle istanze Firebase per un utilizzo efficiente nei componenti e nei servizi dell'applicazione.

Struttura e Funzionamento

Il file segue una struttura standard per l'integrazione di Firebase in un progetto Vue:

```
import { initializeApp } from 'firebase/app';
import { getAuth } from 'firebase/auth';
import { getFirestore } from 'firebase/firestore';
import { getStorage } from 'firebase/storage';

const firebaseConfig = {
  apiKey: import.meta.env.VITE_FIREBASE_API_KEY,
  authDomain: import.meta.env.VITE_FIREBASE_AUTH_DOMAIN,
  projectId: import.meta.env.VITE_FIREBASE_PROJECT_ID,
  storageBucket: import.meta.env.VITE_FIREBASE_STORAGE_BUCKET,
  messagingSenderId: import.meta.env.VITE_FIREBASE_MESSAGING_SENDER_ID,
```

```
  appId: import.meta.env.VITE_FIREBASE_APP_ID,
};

export const firebaseApp = initializeApp(firebaseConfig);

export const auth = getAuth(app);
export const db = getFirestore(app);
export const storage = getStorage(app);
```

Funzionamento

1. **Importa le librerie Firebase** per autenticazione, database e storage.
2. **Definisce la configurazione Firebase**, utilizzando variabili d'ambiente per la sicurezza.
3. **Inizializza Firebase** con `initializeApp(firebaseConfig)`.
4. **Esporta le istanze di `auth`, `db` e `storage`**, rendendole disponibili per il resto dell'applicazione.

Considerazioni

- Le credenziali Firebase non sono hardcoded nel file, ma sono gestite tramite variabili d'ambiente per garantire sicurezza e flessibilità.
- Firestore viene utilizzato per l'archiviazione e il recupero dei dati in modo scalabile.
- Firebase Storage permette di gestire file e media in modo integrato con l'ecosistema Firebase.
- Il file `firebase.ts` funge da punto centrale per l'accesso ai servizi Firebase, semplificando la gestione del backend dell'applicazione.
- L'uso di `import.meta.env` assicura compatibilità con Vite e un accesso più efficiente alle variabili d'ambiente.

`exposeVueComponents.js` - Gestione del Montaggio Dinamico dei Componenti Vue

Ruolo e Scopo

Il file `exposeVueComponents.js` è progettato per gestire il montaggio e smontaggio dinamico dei componenti Vue all'interno dell'interfaccia utente, rispondendo alle interazioni con l'agente conversazionale Aisuru. Le funzioni principali di questo file consentono di montare i componenti nelle aree chat o esterne, nonché di smontarli quando non necessari.

Struttura e Funzionamento

1. Importazione delle Dipendenze

Il file importa `createApp` da Vue, che viene utilizzato per istanziare nuovi componenti Vue e montarli dinamicamente nel DOM.

```
import { createApp } from 'vue';
import Login from './components/Login.vue';
import Register from './components/Register.vue';
import Logout from './components/Logout.vue';
import Catalog from './components/Catalog.vue';
import CatalogCard from './components/CatalogCard.vue';
import SaveGame from './components/SaveGame.vue';
import BrowseSaves from './components/BrowseSaves.vue';
import BrowseCreatedStory from './components/BrowseCreatedStory.vue';
import BrowseCreatedStoryCard from './components/BrowseCreatedStoryCard.vue
import BrowseCreatedStoryScenarios from './components/BrowseCreatedStorySce
import CreateScenario from './components/CreateScenario.vue';
import CreateStory from './components/CreateStory.vue';
```

2. Esposizione dei Componenti Vue

I componenti Vue che possono essere montati dinamicamente sono semplicemente importati direttamente nel file e passati come argomenti alle funzioni di montaggio. Non viene utilizzato un registro centralizzato per i componenti, ma ciascun componente viene gestito in modo indipendente, direttamente nel codice delle funzioni di montaggio.

Per esempio, i componenti come `Login`, `Register`, `Catalog`, e altri vengono importati in cima al file e poi passati alle funzioni di montaggio come `mountVueComponentsInChat` o `mountVueComponentsInExtention`, a seconda dell'area in cui devono essere montati.

```
// Esportare la funzione e i componenti
window.mountVueComponents = mountVueComponents;
window.mountVueComponentsInChat = mountVueComponentsInChat;
```

```
window.mountVueComponentsInExtention = mountVueComponentsInExtention;
window.unmountVueComponentsInChat = unmountVueComponentsInChat;
window.unmountVueComponentsInExtention = unmountVueComponentsInExtention;
window.Login = Login;
window.Register = Register;
window.Logout = Logout;
window.Catalog = Catalog;
window.CatalogCard = CatalogCard;
window.SaveGame = SaveGame;
window.CreateStory = CreateStory;
window.CreateScenario = CreateScenario;
window.BrowseSaves = BrowseSaves;
window.BrowseCreatedStory = BrowseCreatedStory;
window.BrowseCreatedStoryCard = BrowseCreatedStoryCard;
window.BrowseCreatedStoryScenarios = BrowseCreatedStoryScenarios;
```

3. Funzioni di Montaggio

3.1 Montaggio del Componente nella Chat

La funzione `mountVueComponentsInChat` permette di montare un componente Vue all'interno della chat, creando dinamicamente un nodo DOM per ogni contenitore identificato tramite la classe CSS specificata.

```
function mountVueComponentsInChat(className, vueComponent) {
  const containers = document.querySelectorAll(`.${className}:not(.processe

  containers.forEach((container) => {
    if (!container.classList.contains('processed')) {
      createApp(vueComponent).mount(container);
      container.classList.add('processed');
      console.log(`Componente Vue montato in ${container}`);
    }
  });
}
```

3.2 Montaggio del Componente in un'Aggiornamento Esterno

`mountVueComponentsInExtention` monta i componenti Vue all'interno di una sezione esterna all'interfaccia principale, come una finestra o un pannello dedicato. La funzione gestisce il layout e lo stile dell'estensione per garantire una visualizzazione coerente.

```
function mountVueComponentsInExtention(vueComponent, props = {}) {
  const extension = document.getElementById("extension");
  extension.innerHTML = ""; // Resetta il contenuto
  extension.style.position = "fixed"; // Posizione fissa
  extension.style.top = "120px"; // Posizione alta
  extension.style.left = "30px"; // Posizione sinistra
  extension.style.width = "600px"; // Larghezza
  extension.style.height = "80vh"; // Altezza
  extension.style.backgroundColor = "#ffffff"; // Sfondo bianco
  extension.style.border = "1px solid #ddd"; // Bordo
  extension.style.borderRadius = "8px"; // Angoli arrotondati
  extension.style.boxShadow = "0 4px 10px rgba(0, 0, 0, 0.1)"; // Ombra
  extension.style.padding = "15px"; // Spaziatura
  extension.style.overflowY = "auto"; // Scroll se necessario
  extension.style.zIndex = "9999"; // In primo piano

  // Montaggio del componente Vue
  const innerDiv = document.createElement("div");
  innerDiv.style.width = "100%";
  innerDiv.style.height = "auto";
  extension.appendChild(innerDiv);
  createApp(vueComponent, props).mount(innerDiv);
}
```

3.3 Smontaggio dei Componenti

Le funzioni `unmountVueComponentsInChat` e `unmountVueComponentsInExtention` smontano rispettivamente i componenti dalla chat e dall'estensione.

```
function unmountVueComponentsInChat(className) {
  const containers = document.querySelectorAll(`.${className}.processed`);
  containers.forEach((container) => {
    const app = container.__vue_app__;
    if (app) {
      app.unmount();
      delete container.__vue_app__;
      container.classList.remove('processed');
      console.log(`Componente Vue smontato da ${container}`);
    }
  });
}

function unmountVueComponentsInExtention() {
  const extension = document.getElementById("extension");
```

```
    extension.innerHTML = "";
    extension.style.zIndex = "-200";
}
```

Interazione con Aisuru

Quando Aisuru invia un comando per montare o smontare un componente, `exposeVueComponents.js` gestisce la richiesta utilizzando le funzioni di cui sopra. Ecco un esempio di come un comando potrebbe essere ricevuto e gestito:

```
mountVueComponentsInChat("dynamic-container",Login)
```

`App.vue` - Componente Root dell'Applicazione

Ruolo e Scopo

`App.vue` è il componente principale che definisce la struttura di base dell'interfaccia utente dell'applicazione **Adventure Master**. Gestisce il layout dell'app, includendo la barra superiore, la logica di salvataggio e logout, e la sezione in cui viene montato il componente `Memori`. Questo componente funge da contenitore per le interazioni dell'utente con il sistema, esponendo la configurazione iniziale per l'agente conversazionale Memori.

Struttura e Collegamenti con Altri File

- **Template**: Definisce la struttura visiva del layout, compresa la barra del menu e il contenuto principale.
- **Componente `Memori.vue`**: Viene montato dinamicamente nel layout con la configurazione passata tramite la proprietà `memoriConfig`.
- **Componente `SaveGame`**: Gestisce il salvataggio del gioco, incluso nel menu della barra superiore.
- **Componente `Logout`**: Gestisce la logica di logout, anch'esso incluso nella barra superiore.

Analisi Dettagliata del Codice

```
<template>
  <v-app>
    <v-app-bar app>
      <v-toolbar-title>Adventures Master</v-toolbar-title>
      <v-spacer></v-spacer>
      <SaveGame />
      <Logout />
    </v-app-bar>
    <v-main>
      <v-container fluid>
        <v-row>
          <v-col cols="12" md="6" class="left-column">
          </v-col>
          <v-col cols="12" md="12" class="memori-container">
            <Memori :memoriConfig="someMemoriConfig" />
          </v-col>
        </v-row>
      </v-container>
    </v-main>
  </v-app>
</template>
```

- **`<v-app>`** : Contenitore principale che definisce l'applicazione Vuetify.
- **`<v-app-bar>`** : Barra superiore con il titolo dell'app e i componenti `SaveGame` e `Logout` per gestire il salvataggio e il logout.
- **`<v-main>`** : Sezione principale dove è visualizzato il contenuto dell'app, che include una griglia di Vuetify per la disposizione dei componenti.
- **`<v-col>`** : Definisce le colonne per il layout, separando la parte sinistra (attualmente vuota) dalla parte destra in cui viene montato il componente `Memori` .

```
<script lang="ts">
import { defineComponent, ref } from 'vue';
import Memori from "./components/Memori.vue";
import SaveGame from './components/SaveGame.vue';
import Logout from './components/Logout.vue';
import type { MemoriConfig, GameSave } from "@/models/GameSave";

export default defineComponent({
  name: "App",
  components: {
    Memori,
    SaveGame,
    Logout,
```

```
  },
  setup() {
    const someMemoriConfig = ref<MemoriConfig>({
      context: "AUTH:NON_AUTENTICATO,STORIA: NULL",
      initialQuestion: "Benvenuto"
    });

    return {
      someMemoriConfig,
    };
  },
});
</script>
```

- **`setup()`** : Utilizza la Composition API di Vue 3 per configurare lo stato del componente. In questo caso, viene creato un oggetto `someMemoriConfig` con la configurazione iniziale per il componente `Memori` .
- **`ref<MemoriConfig>`** : Viene utilizzato per creare una variabile reattiva che contiene la configurazione iniziale dell'agente conversazionale.

```
<style scoped>
.left-column {
  padding: 16px;
}

.dynamic-content {
  height: 50px; /* Altezza placeholder */
  border: 1px dashed #ccc;
  margin-bottom: 16px;
}

.memori-container {
  padding: 16px;
  border-left: 1px solid #ccc; /* Separatore tra le due colonne */
}
</style>
```

- **CSS Scoped**: Gli stili sono applicati solo all'interno di questo componente, grazie alla direttiva `scoped` .
  - **`.left-column`** : Definisce la colonna sinistra con una spaziatura interna.

- **`.memori-container`** : Aggiunge padding e una linea di separazione tra la colonna sinistra e quella destra dove viene montato il componente `Memori`.

`Memori.vue` - Integrazione con l'Agente Conversazionale Memori

Ruolo e Scopo

`Memori.vue` è il componente responsabile per l'integrazione dell'agente conversazionale Memori all'interno dell'interfaccia utente dell'applicazione **Adventure Master**. Gestisce la visualizzazione dell'interfaccia di Memori tramite il componente `memori-client`, configurandolo dinamicamente in base ai dati forniti tramite le props.
Questo componente si occupa di aggiornare la configurazione di Memori quando vengono ricevuti nuovi dati tramite eventi e fornisce un meccanismo per il refresh dinamico dell'interfaccia.

Struttura e Collegamenti con Altri File

- **Componente `memori-client`** : Questo web-component esterno è utilizzato per visualizzare l'interfaccia dell'agente conversazionale e interagire con Aisuru.
- **`eventBus`** : Utilizzato per ascoltare e ricevere eventi esterni (come la modifica della configurazione di Memori) e aggiornare dinamicamente la configurazione del componente.

Analisi Dettagliata del Codice

```
<template>
  <div>
    <memori-client
      :key="refreshKey"
      memoriName="Adventures Master"
      ownerUserName="matteocardinale2002"
      memoriID="fa22537f-6cd4-45f9-b21e-347747222db6"
      ownerUserID="fcfa7ae4-aedc-4a52-a137-4d2a858d6561"
      tenantID="www.aisuru.com"
      engineURL="https://engine.memori.ai"
```

```
        apiURL="https://backend.memori.ai"
        baseURL="https://www.aisuru.com"
        uiLang="IT"
        spokenLang="IT"
        layout="ZOOMED_FULL_BODY"
        showInstruct="false"
        showSettings="true"
        showClear="false"
        showAIicon="true"
        showWhyThisAnswer="true"
        showTypingText="false"
        showOnlyLastMessages="true"
        showTranslationOriginal="false"
        showCopyButton="false"
        showShare="true"
        showLogin="false"
        useMathFormatting="false"
        showUpload="false"
        autoStart="false"
        enableAudio="true"
        integrationID="5f2ab2ab-5574-4ec6-9034-fd5fb948a449"
        :context="finalMemoriConfig.context"
        :initialQuestion="finalMemoriConfig.initialQuestion"
      />
    </div>
</template>
```

- **`<memori-client>`** : Questo è il componente principale che carica e visualizza l'interfaccia dell'agente conversazionale Memori.

```
<script setup lang="ts">
import { ref, computed } from 'vue';
import type { MemoriConfig, GameSave } from '@/models/GameSave';
import { onMounted, onUnmounted } from 'vue';
import eventBus from '@/eventBus';

const props = defineProps<{
  memoriConfig?: MemoriConfig;
  gameSaveData?: GameSave;
}>();

onMounted(() => {
  eventBus.on('updateMemoriConfig', (newConfig: MemoriConfig) => {
    refreshMemori(newConfig);
```

```
  });
});

onUnmounted(() => {
  eventBus.off('updateMemoriConfig');
});

// Fallback default configuration
const defaultMemoriConfig: MemoriConfig = {
  context: 'AUTH:NON_AUTENTICATO,STORIA:NULL',
  initialQuestion: 'Benvenuto',
};

// Reactive key for forcing a refresh
const refreshKey = ref(0);

// Final configuration based on props
const finalMemoriConfig = computed<MemoriConfig>(() => {
  if (props.gameSaveData && props.gameSaveData.memoriConfig) {
    return props.gameSaveData.memoriConfig;
  } else if (props.memoriConfig) {
    return props.memoriConfig;
  }
  return defaultMemoriConfig;
});

function refreshMemori(newConfig: MemoriConfig) {
  if (props.memoriConfig) {
    Object.assign(props.memoriConfig, newConfig);
  }
  refreshKey.value += 1;
}

// Expose method for parent components
defineExpose({
  refreshMemori,
});
</script>
```

- **`finalMemoriConfig`** : Questa variabile computata determina la configurazione finale di Memori, che può essere presa da `props.gameSaveData` , `props.memoriConfig` , o, se entrambe le opzioni non sono disponibili, viene utilizzata una configurazione di fallback (`defaultMemoriConfig`).

- **`refreshMemori`** : La funzione aggiorna dinamicamente la configurazione di Memori e forza il refresh del componente incrementando il `refreshKey` , che è legato alla proprietà `:key` di `memori-client` per forzare un re-rendering.

Gestione dell'Autenticazione e Registrazione

Panoramica

L'applicazione **Adventure Master** include i componenti di **Login** e **Registrazione** come parte del flusso di autenticazione. I componenti `Login.vue` e `Register.vue` vengono montati dinamicamente nell'interfaccia tramite Aisuru, utilizzando la funzione `mountVueComponentsInChat` , che permette di visualizzare questi moduli di login e registrazione direttamente nella chat. I dati dell'utente vengono gestiti tramite **Firebase** per l'autenticazione e **Firestore** per il salvataggio delle informazioni.

Ruolo e Scopo dei Componenti

1. **`Login.vue`** :
    - **Funzione**: Gestisce l'autenticazione dell'utente. Permette l'inserimento di email e password per il login, e si interfaccia con Firebase per l'autenticazione dell'utente.
    - **Montaggio**: Viene montato dinamicamente nella chat di Aisuru tramite `mountVueComponentsInChat` .
2. **`Register.vue`** :
    - **Funzione**: Gestisce la registrazione di nuovi utenti. Gli utenti possono inserire la propria email, password e conferma della password, e il componente registra il nuovo utente tramite Firebase.
    - **Montaggio**: Viene montato nella chat, come il componente `Login.vue` , tramite `mountVueComponentsInChat` .
3. **`UserRepository.ts`** :

- **Funzione**: Si occupa della gestione dei dati utente nel database Firestore, fornendo metodi per salvare, aggiornare e recuperare informazioni sugli utenti.
- **Interazione**: Dopo che un nuovo utente si registra tramite `Register.vue`, i suoi dati vengono salvati nel database utilizzando `UserRepository`. Inoltre, la gestione delle sessioni di login è tracciata per la registrazione dei log di accesso.

Interazione tra i Componenti

1. **Montaggio dei Componenti nella Chat tramite Aisuru**:
   - `Login.vue` e `Register.vue` vengono montati dinamicamente nella chat di Aisuru usando `mountVueComponentsInChat`. Quando Aisuru invia un comando, questi componenti vengono visualizzati nell'interfaccia utente dell'applicazione. Un esempio di comando potrebbe essere:

```
mountVueComponentsInChat("dynamic-container",Register)
```

2. **Autenticazione tramite Firebase**:
   - **Login**: Quando l'utente inserisce email e password, il componente `Login.vue` utilizza `signInWithEmailAndPassword` per autenticare l'utente tramite Firebase. Se il login ha successo, un log dell'evento "User logged in" viene registrato nel database tramite `LogRepository`.
   - **Registrazione**: Se l'utente si registra, il componente `Register.vue` usa `createUserWithEmailAndPassword` per creare un nuovo account su Firebase. I dati dell'utente vengono poi salvati nel database tramite `UserRepository` e un log dell'evento "User registered" viene registrato tramite `LogRepository`.

3. **Salvataggio dei Dati Utente e Log**:
   - **`UserRepository`**: Una volta che un utente si registra, i suoi dati vengono salvati nel database Firestore tramite il metodo `saveUser` di `UserRepository`. In caso di login, i dati dell'utente vengono utilizzati per configurare la sessione nell'app.

- **Log degli Eventi**: Ogni accesso e registrazione dell'utente vengono tracciati nel database tramite `LogRepository`.

4. **Gestione della Sessione e Memori**:

   - Dopo il login, la configurazione di Memori può essere aggiornata dinamicamente in base allo stato dell'utente. I messaggi e le interazioni di Aisuru vengono inviati tramite `window.typeBatchMessages`, utilizzando la configurazione di Memori per interagire con l'utente. Questi messaggi attivano contenuti appositi per cambiare il contesto AUTH e far restituire il messaggio di benvenuto al menù che porterà con s'è le relative opzioni.

Catalog.vue, CatalogCard.vue e StoryRepository.ts

Ruolo e Scopo

Questi file gestiscono il catalogo delle storie nell'applicazione **Adventure Master**, permettendo agli utenti di visualizzare, filtrare e selezionare una storia per avviare l'esperienza di gioco.

- **Catalog.vue**: Componente principale che visualizza tutte le storie disponibili, includendo un sistema di filtri.
- **CatalogCard.vue**: Componente secondario che rappresenta ogni storia come una card cliccabile.
- **StoryRepository.ts**: Repository che gestisce l'accesso alle storie nel database Firestore.

Struttura e Collegamenti tra i file

1. **Catalog.vue**:

   - Recupera le storie tramite `StoryRepository.getAllStories()`.
   - Filtra le storie in base ad autore, genere e titolo.
   - Mostra ogni storia utilizzando il componente `CatalogCard.vue`.

2. **CatalogCard.vue**:

- Riceve i dati della storia come `prop`.
- Mostra titolo, descrizione, immagine, autore e genere.
- Al click, richiama `window.typeMessage` per avviare la storia selezionata.

3. **StoryRepository.ts**:

   - Contiene metodi per recuperare, creare, aggiornare ed eliminare storie in Firestore.
   - `getAllStories()`: recupera tutte le storie dal database.
   - `getStoriesByGenre(genre)`: filtra le storie per genere.
   - `getStoriesByAuthor(author)`: filtra le storie per autore.

Analisi Dettagliata del Codice

Catalog.vue

Il componente **Catalog.vue** gestisce l'interfaccia principale del catalogo e implementa un sistema di filtri dinamici.

- **Recupero storie da Firestore**:

```
StoryRepository.getAllStories().then((data) => {
    stories.value = data;
});
```

- **Filtraggio delle storie**:

```
const filteredStories = computed(() => {
    return stories.value.filter((story) => {
        const matchesAuthor = !filters.value.author || story.author.to
        const matchesGenre = !filters.value.genre || story.genre === f
        const matchesTitle = !filters.value.title || story.title.toLow
        return matchesAuthor && matchesGenre && matchesTitle;
    });
});
```

- **Rendering dinamico delle card**:

```
<CatalogCard
  v-for="story in filteredStories"
  :key="story.id"
  :story="story"
/>
```

CatalogCard.vue

- **Struttura del componente**:

```
<template>
  <div class="card h-100">
    <img :src="story.image" class="card-img-top" :alt="story.title" />
    <div class="card-body">
      <h5 class="card-title">{{ story.title }}</h5>
      <p class="card-text">{{ story.description }}</p>
      <p class="text-muted">Autore: {{ story.author }}</p>
      <p class="text-muted">Genere: {{ story.genre }}</p>
      <button class="btn btn-primary" @click="playStory">Gioca</button
    </div>
  </div>
</template>
```

- **Gestione dell'evento di selezione della storia**:

```
methods: {
  playStory() {
      if (typeof window.typeMessage === 'function') {
          window.typeMessage(this.story.title, true, true);
          window.unmountVueComponentsInExtention("Catalog");
      } else {
          console.error('window.typeMessage non è definita');
      }
  },
}
```

StoryRepository.ts

Il repository gestisce le interazioni con Firestore.

- **Recupero di tutte le storie**:

```
static async getAllStories(): Promise<Story[]> {
  const snapshot = await getDocs(storyCollection);
  return snapshot.docs.map((doc) => ({
    id: doc.id,
    ...(doc.data() as Omit<Story, 'id'>)
  }));
}
```

- **Filtraggio per autore**:

```
static async getStoriesByAuthor(author: string): Promise<Story[]> {
  const authorQuery = query(storyCollection, where('author', '==', aut
  const snapshot = await getDocs(authorQuery);
  return snapshot.docs.map((doc) => ({
    id: doc.id,
    title: doc.data().title || '',
    description: doc.data().description || '',
    image: doc.data().image || '',
    author: doc.data().author || '',
    genre: doc.data().genre || '',
  }));
}
```

SaveGame.vue, BrowseSaves.vue e GameSaveRepository.ts

Ruolo e Scopo

- **SaveGame.vue**: Permette all'utente di salvare i progressi della storia corrente nel database Firestore, interagendo con **GameSaveRepository.ts** e ottenendo lo stato dal sistema Memori.
- **BrowseSaves.vue**: Mostra i salvataggi disponibili, permettendo il caricamento o l'eliminazione di un salvataggio.
- **GameSaveRepository.ts**: Repository che gestisce il salvataggio, il recupero e la rimozione dei progressi di gioco in Firestore.

Struttura e Collegamenti tra i file

1. **SaveGame.vue**:
   - Recupera lo stato della storia tramite `window.getMemoriState()`.
   - Prepara i dati di salvataggio (progresso, inventario, stato Memori).
   - Salva i progressi tramite `GameSaveRepository.saveGameSave()`.
2. **BrowseSaves.vue**:
   - Recupera i salvataggi dell'utente autenticato tramite `GameSaveRepository.getGameSavesByUserId()`.
   - Permette il caricamento di un salvataggio aggiornando la configurazione di Memori.
   - Gestisce l'eliminazione dei salvataggi tramite `GameSaveRepository.deleteGameSaveById()`.
3. **GameSaveRepository.ts**:
   - `saveGameSave(gameSave)` : Crea un nuovo salvataggio in Firestore.
   - `getGameSavesByUserId(userId)` : Recupera tutti i salvataggi di un utente.
   - `deleteGameSaveById(saveId)` : Elimina un salvataggio specifico.
   - `getGameSaveById(saveId)` : Recupera un singolo salvataggio per ID.

Analisi Dettagliata del Codice

SaveGame.vue

- **Recupero dello stato Memori**:

```
const fetchMemoriState = async () => {
  try {
    const memoriState = await window.getMemoriState();
    const contextVarsString = JSON.stringify(memoriState.contextVars);
    const parsedContextVars = JSON.parse(contextVarsString);
    storyId.value = parsedContextVars.STORIA || 'NULL';
  } catch (error) {
    console.error("Errore nel recupero dello stato Memori:", error);
  }
};
```

- **Salvataggio del progresso**:

```
const saveProgress = async () => {
  if (storyId.value === "NULL") return;

  isSaving.value = true;
  errorMessage.value = '';

  try {
    const userId = auth.currentUser?.uid;
    if (!userId) {
      throw new Error('User not authenticated');
    }

    const memoriState = await window.getMemoriState();
    const progress = memoriState.contextVars.SCENARIO || 'default-prog

    const docId = await GameSaveRepository.saveGameSave({
      userId,
      storyId: storyId.value,
      progress,
      saveDate: new Date(),
    });

    alert(`Salvataggio completato! ID: ${docId}`);
  } catch (err) {
    errorMessage.value = 'Errore durante il salvataggio. Riprova.';
  } finally {
    isSaving.value = false;
  }
};
```

BrowseSaves.vue

- **Recupero dei salvataggi**:

```
onMounted(async () => {
  saves.value = await GameSaveRepository.getGameSavesByUserId(auth.cur
});
```

- **Caricamento di un salvataggio**:

```
const loadSave = (save) => {
  const memoriConfig = save.memoriConfig;
  if (memoriConfig) {
    eventBus.emit('updateMemoriConfig', memoriConfig);
  } else {
    console.error('Configurazione di Memori non valida.');
  }
};
```

- **Eliminazione di un salvataggio**:

```
const deleteSave = async (id) => {
  await GameSaveRepository.deleteGameSaveById(id);
  saves.value = saves.value.filter(save => save.id !== id);
};
```

GameSaveRepository.ts

- **Salvataggio di un progresso di gioco**:

```
static async saveGameSave(gameSave: Omit<GameSave, 'id'>): Promise<str
  const docRef = await addDoc(gameSaveCollection, gameSave);
  return docRef.id;
}
```

- **Recupero di tutti i salvataggi di un utente**:

```
static async getGameSavesByUserId(userId: string): Promise<GameSave[]>
  const userQuery = query(gameSaveCollection, where('userId', '==', us
  const querySnapshot = await getDocs(userQuery);
  return querySnapshot.docs.map(doc => ({
    id: doc.id,
    ...doc.data()
  } as GameSave));
}
```

- **Eliminazione di un salvataggio**:

```
  static async deleteGameSaveById(saveId: string): Promise<void> {
    const gameSaveRef = doc(gameSaveCollection, saveId);
    await deleteDoc(gameSaveRef);
  }
```

BrowseCreatedStory.vue, BrowseCreatedStoryCard.vue, BrowseCreatedStoryScenarios.vue

Ruolo e Scopo

- **BrowseCreatedStory.vue**: Gestisce la visualizzazione delle storie create dall'utente, permettendo il filtraggio e la gestione delle modifiche.
- **BrowseCreatedStoryCard.vue**: Mostra ogni storia in formato card, fornendo accesso rapido ai dettagli e alle funzionalità di modifica.
- **BrowseCreatedStoryScenarios.vue**: Consente di visualizzare e modificare i contenuti delle storie salvate, interagendo con **AisuruService.ts**.

Struttura e Collegamenti tra i file

1. **BrowseCreatedStory.vue**:
    - Recupera le storie create dall'utente tramite `StoryRepository.getStoriesByAuthor()`.
    - Filtra le storie per genere e titolo.
    - Permette la modifica delle storie tramite `BrowseCreatedStoryCard.vue`.
    - Accede ai contenuti delle storie tramite `BrowseCreatedStoryScenarios.vue`.
2. **BrowseCreatedStoryCard.vue**:
    - Visualizza le informazioni principali di una storia (titolo, descrizione, genere).
    - Fornisce pulsanti per modificare dettagli e contenuti della storia.
    - Emissione di eventi `edit-details` e `edit-content` per attivare le funzionalità di modifica.
3. **BrowseCreatedStoryScenarios.vue**:

- Recupera i contenuti delle storie utilizzando `AisuruService.ts`.
- Permette la modifica delle risposte nelle memorie di una storia.
- Invia le modifiche aggiornate tramite `updateMemory()`.

Analisi Dettagliata del Codice

BrowseCreatedStory.vue

- **Recupero delle storie dell'utente**:

```
StoryRepository.getStoriesByAuthor(currentUser.uid).then((data) => {
    stories.value = data;
});
```

- **Filtraggio delle storie**:

```
const filteredStories = computed(() => {
    return stories.value.filter((story) => {
       const matchesGenre = !filters.value.genre || story.genre === fi
       const matchesTitle = !filters.value.title || story.title.toLowe
       return matchesGenre && matchesTitle;
    });
});
```

- **Cambio vista tra catalogo e modifica**:

```
const editStory = (story) => {
    selectedStory.value = story;
    currentView.value = 'editStory';
};

const editContent = (story) => {
    selectedStory.value = story;
    currentView.value = 'editContent';
};
```

BrowseCreatedStoryCard.vue

- **Emissione di eventi per la modifica**:

```
<button class="btn btn-primary" @click="$emit('edit-details', story)">
<button class="btn btn-primary" @click="$emit('edit-content', story)">
```

BrowseCreatedStoryScenarios.vue

- **Recupero delle memorie di una storia**:

```
const fetchMemories = async () => {
    memories.value = await aisuruService.filteredPaginatedMemories(pro
};
```

- **Modifica delle risposte nelle memorie**:

```
const saveEdit = async (memoryID) => {
    const memory = {
        memoryType: 'Question',
        memoryID: editingMemoryId.value,
        answers: [{ text: editedAnswer.value, lastChangeTimestamp: new
    };
    await aisuruService.updateMemory(memory);
    cancelEdit();
};
```

CreateStory.vue, CreateScenario.vue, StoryRepository.ts

Ruolo e Scopo

Questi file lavorano insieme per consentire agli utenti di creare, modificare e gestire storie e scenari all'interno dell'applicazione.

- **CreateStory.vue** fornisce un'interfaccia intuitiva per la creazione e modifica delle storie, consentendo agli utenti di inserire un titolo, una descrizione, un'immagine e un genere.
- **CreateScenario.vue** permette di aggiungere scenari alla storia, integrandosi con `AisuruService.ts` per memorizzare le informazioni relative agli eventi della storia.

- **StoryRepository.ts** gestisce l'accesso ai dati nel database Firestore, fornendo funzioni per salvare, recuperare, aggiornare ed eliminare storie.

Struttura e Collegamenti tra i file

Il processo inizia con **CreateStory.vue**, dove l'utente inserisce le informazioni di base della storia. Se la storia è nuova, dopo il salvataggio viene avviato automaticamente **CreateScenario.vue**, che consente di aggiungere scenari collegati alla storia. Tutti i dati vengono gestiti tramite **StoryRepository.ts**, che interagisce con Firestore per garantire la persistenza delle informazioni.

1. **CreateStory.vue**:
    - Permette agli utenti di creare o modificare una storia.
    - Salva le informazioni nel database attraverso `StoryRepository.ts`.
    - Se la storia è nuova, apre `CreateScenario.vue` per la creazione degli scenari associati.
2. **CreateScenario.vue**:
    - Consente di aggiungere e gestire scenari per una storia specifica.
    - Comunica con `AisuruService.ts` per creare le memorie della storia e definire il flusso narrativo.
    - Utilizza variabili di contesto per gestire la progressione della storia.
3. **StoryRepository.ts**:
    - Si occupa di tutte le operazioni di gestione delle storie nel database Firestore.
    - Include metodi per salvare, aggiornare, recuperare ed eliminare storie.

Analisi Dettagliata del Codice

Creazione e modifica di una storia con CreateStory.vue

Quando un utente vuole creare o modificare una storia, **CreateStory.vue** gestisce il flusso con un'interfaccia semplice e intuitiva. Se l'utente sta modificando una storia esistente, i campi vengono precompilati con i dati salvati. Se invece sta creando una nuova storia, i campi saranno vuoti e pronti per essere compilati.

- **Salvataggio di una storia**:

```
const saveStory = async () => {
  if (!auth.currentUser) {
    throw new Error("Utente non autenticato");
  }
  story.author = auth.currentUser.uid;
  story.image = selectedImage ? await ImageRepository.uploadImage(sele

  if (story.id) {
    await StoryRepository.updateStory(story.id, story);
  } else {
    story.id = await StoryRepository.saveStory(story);
    storySaved.value = true;
  }
};
```

Gestione degli scenari con CreateScenario.vue

Dopo aver salvato una storia, l'utente può definirne gli scenari. **CreateScenario.vue** permette di creare eventi e situazioni che influenzano la narrazione. Gli scenari vengono gestiti come "memorie" che contengono domande, risposte e condizioni di attivazione, tutte memorizzate tramite **AisuruService.ts**.

- **Salvataggio di uno scenario**:

```
const saveScenario = async () => {
  const memory = {
    memoryType: 'Question',
    title: scenario.title,
    answers: [{ text: scenario.answer.text, preformatted: scenario.ans
    conclusive: scenario.isFinal,
    hints: scenario.hints,
    contextVarsToMatch: { STORIA: storyId },
    tags: [storyTitle],
  };
  await aisuruService.addMemory(memory);
};
```

Gestione dei dati con StoryRepository.ts

Per garantire la persistenza delle storie nel database, **StoryRepository.ts** fornisce una serie di metodi per interagire con Firestore. Oltre a salvare e aggiornare le storie, permette di recuperarle filtrando per autore o genere, facilitando la gestione e l'organizzazione dei contenuti.

- **Salvataggio e aggiornamento di una storia**:

```
  static async saveStory(story: Story): Promise<string> {
    const storyRef = doc(storyCollection);
    await setDoc(storyRef, story);
    return storyRef.id
  }
```

- **Recupero delle storie di un autore**:

```
static async getStoriesByAuthor(author: string): Promise<Story[]> {
  const authorQuery = query(storyCollection, where('author', '==', aut
  const snapshot = await getDocs(authorQuery);
  return snapshot.docs.map((doc) => ({
    id: doc.id,
    ...(doc.data() as Story)
  }));
}
```

Design Pattern Adottati e Vantaggi Ottenuti

Pattern Principali Utilizzati

Pattern Repository

Il **Pattern Repository** è stato utilizzato per gestire l'accesso ai dati, separando la logica di persistenza dal resto dell'applicazione. Questo approccio consente di:

- Centralizzare le operazioni di lettura e scrittura su Firestore.
- Ridurre la duplicazione del codice.
- Facilitare eventuali modifiche al sistema di storage senza impattare il codice dell'interfaccia utente.

Utilizzo di Service per l'interazione con Aisuru

L'interazione con l'agente conversazionale Aisuru avviene tramite **AisuruService.ts**, che fornisce metodi dedicati per l'invio e il recupero di dati. Questo approccio permette di:

- Incapsulare la logica di comunicazione con l'API di Memori.
- Garantire una gestione centralizzata della sessione.

In questo modo, i componenti Vue non interagiscono direttamente con l'API, ma utilizzano un'interfaccia intermedia che standardizza la comunicazione.

Factory Pattern

Il **Factory Pattern** viene utilizzato per centralizzare la creazione di oggetti complessi, migliorando la coerenza del codice e riducendo la duplicazione. Un esempio è il metodo `createScenario()`, che viene impiegato per generare nuovi scenari in modo strutturato:

```
function createScenario(title: string = "") {
  return {
    title,
    answer: {
      text: "",
      preformatted: true,
    },
    memoryType: "Question",
    conclusive: true,
    notPickable: true,
    help: false,
    hints: [] as string[],
    contextVarsToSet: "",
    contextVarsToMatch: "",
    isFinal: false,
  };
}
```

Questo approccio garantisce che ogni scenario venga inizializzato con una struttura coerente, evitando errori dovuti a inizializzazioni incomplete o incoerenti.

Strategy Pattern (Parsing di JSON)

Per la gestione sicura del parsing di dati JSON, viene utilizzato un'implementazione simile al **Strategy Pattern**, separando la logica di validazione ed evitando errori in fase di esecuzione:

```
function parseContextVars(input: string): Record<string, string> {
  if (!input.trim()) return {}; // Se vuoto, restituisce un oggetto vuoto
  try {
    const parsed = JSON.parse(input);
    if (typeof parsed !== "object" || Array.isArray(parsed)) {
      throw new Error("Il JSON deve essere un oggetto.");
    }
    return parsed;
  } catch {
    throw new Error("Formato JSON non valido.");
  }
}
```

Questa funzione viene utilizzata per validare le variabili di contesto JSON all'interno degli scenari, assicurandosi che siano nel formato corretto prima di essere processate dall'applicazione.