Introduction

Project Overview

The insightstream.io-dev project is a js-based web application that follows a modular architecture. It includes essential configurations, reusable components, state management, and routing mechanisms to ensure scalability and maintainability. The project is structured using industry best practices and relies on Webpack for bundling and Vuex for state management.

The team consist of five members:

- DEVANAND S
- SANTHOSH KUMAR P
- THIRUMALAI S
- DEEPAK A
- KRISHNAMOORTHY M

Technology Stack

State Management: Vuex

Routing: Vue Router

Build Tool: Webpack

Styling: CSS / SCSS (if applicable)

Package Management: npm (Node Package Manager)

Key Features

Modular Vue components for reusability

Vue Router for navigation

Vuex store for centralized state management

Webpack configuration for optimized builds

Static asset management for icons and images

Development and production environment configurations

Project Structure

The project follows a structured directory setup:

build/: Webpack and build scripts

config/: Environment configurations (development, production, testing)

src/: Core application files, including components, router, store, and utilities

static/: Static files such as images and the manifest.json file

Root files for linting, package management, and project settings

Description

About the Project

The insightstream.io-dev project is a Vue.js-based web application designed with a modular structure for scalability and maintainability. It leverages Vue’s component-based architecture, Vuex for state management, Vue Router for navigation, and Webpack for efficient bundling.

Core Functionalities

Component-Based Architecture: The project is divided into reusable Vue components, ensuring better organization and reusability of UI elements.

Routing with Vue Router: The application uses Vue Router to handle page navigation dynamically.

State Management with Vuex: Centralized data management is achieved using Vuex, making state handling efficient across components.

Webpack Configuration: The build system is optimized with Webpack, ensuring faster load times and code splitting.

Static Asset Handling: The project includes a dedicated static/ directory for managing icons, images, and other assets.

Development and Production Environment Configurations: The project has separate configurations for different environments (dev.env.js, prod.env.js, test.env.js).

Project Workflow

1. The application starts from main.js, which initializes Vue and mounts App.vue.

2. The router/index.js file defines all application routes and maps them to respective components.

3. The store/index.js file manages application-wide state and logic.

4. Components within src/components/ handle specific UI features, categorized into shared/ (generic components) and vuejs/ (page-specific components).

5. The build/ directory contains Webpack configurations that manage the build process, including development and production optimizations.

Use Cases

This project can be used as a foundation for building:

Single Page Applications (SPA)

Content-based web platforms

Vue.js-based admin dashboards

Scalable front-end applications with modular components

Scenario-Based Introduction

Project Scenario

Imagine a team of developers working on a dynamic and scalable web application that requires seamless user interaction, efficient state management, and a modular structure. The insightstream.io-dev project is built to address these needs using Vue.js, ensuring a smooth and efficient development workflow.

Key Objectives

1. Modular and Maintainable Codebase

Implement a component-based architecture that allows reusability and easy maintenance.

Structure the code in a way that supports future scalability.

2. Efficient State Management

Utilize Vuex for centralized state management to ensure consistency across components.

Minimize unnecessary data fetching and optimize performance.

3. Seamless Navigation and Routing

Implement Vue Router to manage dynamic page transitions.

Ensure efficient navigation handling without page reloads.

4. Optimized Build and Performance

Leverage Webpack configurations to optimize assets and improve load times.

Separate development and production environments for efficient deployment.

5. User-Friendly and Responsive UI

Create a visually appealing and interactive user interface using Vue components.

Ensure mobile responsiveness and cross-browser compatibility.

6. Scalability and Extensibility

Design the project to support future enhancements and feature additions.

Maintain a structured and well-documented codebase for easy collaboration.

Features of InsightStream

1. Modular Component-Based Architecture

Reusable Vue components for better maintainability.

Organized structure with shared and page-specific components.

2. Efficient State Management

Vuex is used to manage application-wide state.

Ensures data consistency across components.

3. Dynamic Routing with Vue Router

Enables seamless navigation without page reloads.

Supports dynamic and nested routing.

4. Optimized Build System

Webpack-based build process for optimized performance.

Minified and bundled assets for faster load times.

5. Static Asset Management

Dedicated static/ folder for images and other assets.

Icons and logos stored separately for easy access.

6. Development & Production Configurations

Separate configurations for development, testing, and production.

Environment variables for flexible deployment.

7. User-Friendly UI Components

Pre-built UI elements like cards, lists, and forms.

Responsive design for mobile and desktop users.

8. Secure API Handling

Utility functions for handling API requests efficiently.

Error management for better debugging and troubleshooting.

9. Customizable Theming & Styling

Supports SCSS/CSS customization for a unique look and feel.

Easily adaptable styles with a structured approach.

10. Future Scalability & Extensibility

Designed to support additional features and enhancements.

Easy integration with third-party APIs and libraries.

Technical Architecture of InsightStream

1. High-Level Overview

InsightStream is built using Vue.js as the frontend framework, Vuex for state management, Vue Router for navigation, and Webpack for module bundling. The project is structured in a way that ensures scalability, modularity, and maintainability.

2. Architecture Layers

a) Presentation Layer (Frontend UI - Vue.js)

Components (src/components/):

Reusable UI elements for dynamic content rendering.

Categorized into shared and page-specific components.

Assets (src/assets/):

Images, logos, and other static resources.

Views (src/components/vuejs/):

Page-level components for different sections of the application.

b) Application Layer (Logic & State Management)

State Management (src/store/):

Uses Vuex to manage global application state.

Ensures consistent data flow between components.

Routing (src/router/index.js):

Uses Vue Router for navigation between views.

Supports nested and dynamic routes.

Core Utilities (src/core/):

API Handling (xhr.js): Handles HTTP requests and responses.

Error Handling (errors.js): Manages exceptions and error messages.

Local Storage (lstorage.js): Manages session-based and persistent data.

c) Build & Configuration Layer

Build System (build/):

Webpack configuration for optimized asset bundling.

Supports hot reloading in development mode.

Environment Configurations (config/):

Separate settings for development, testing, and production (dev.env.js, prod.env.js, test.env.js).

d) Static Content & Deployment

Static Files (static/):

Icons, manifest files, and other assets.

Deployment Configuration:

Supports Firebase, Netlify, or any static hosting solution.

Webpack optimizations for efficient deployment.

3. Data Flow in InsightStream

1. User Interaction → Clicks a button or navigates a page.

2. Component Triggers Vuex Action → Updates state in store/index.js.

3. API Call (if required) → Handled in xhr.js for fetching/sending data.

4. Vuex Updates State → Components reactively update based on new data.

5. Vue Router Handles Navigation → Renders the appropriate view dynamically.

4. Key Architectural Principles

Component-Based Design → Promotes code reusability and maintainability.

Centralized State Management → Ensures a single source of truth with Vuex.

Lazy Loading & Code Splitting → Optimized performance using Webpack.

Environment-Specific Configurations → Supports seamless transitions between development, testing, and production.

Pre-requisites to Get the Application Project from Drive

Before setting up and running the InsightStream project, ensure you have the following prerequisites installed on your system.

1. System Requirements

Operating System: Windows, macOS, or Linux

RAM: Minimum 4GB (8GB recommended for smooth development)

Storage: At least 2GB of free space

3. Navigate to the Project Folder

Open the terminal or command prompt.

Move into the project directory:

```
cd path/to/insightstream.io-dev
```

4. Install Dependencies

Run the following command to install project dependencies:

```
npm install
```

Project Flow of InsightStream

2. User Navigation & Routing

Vue Router (src/router/index.js) handles navigation between pages.

The user navigates through different sections via the router-view component.

Routes are dynamically loaded to improve performance (lazy loading).

Routing Example:

Homepage (/) → Loads home.vue.

About Page (/about) → Loads about.vue.

Article Management (/articles) → Routes to create.vue and view.vue.

---

3. Data Flow & State Management

Vuex (src/store/index.js) manages the global state.

When a user triggers an action (e.g., submitting a form), the flow is:

1. Component Dispatches an Action → Calls Vuex action.

2. Vuex Action Calls API (if needed) → xhr.js handles API requests.

3. Mutation Updates the State → Vue components reactively update.

Example: Fetching Articles

1. User visits /articles.

2. The component dispatches fetchArticles() to Vuex.

3. Vuex calls an API via xhr.js and updates the store.

4. The UI automatically updates with the fetched articles.

---

4. Handling API Requests

src/core/xhr.js handles HTTP requests efficiently.

Uses fetch() or Axios to communicate with back-end APIs.

Implements error handling through errors.js.

---

5. Rendering Components & UI Updates

Shared components (src/components/shared/) are used for reusable UI elements like grids, cards, and forms.

Page-specific components are rendered dynamically inside router-view.

Vue’s reactivity ensures seamless UI updates based on state changes.

---

6. Asset & Static File Handling

Static images, icons, and other resources are stored in static/img/.

The manifest.json helps in defining metadata for PWA support.

---

7. Deployment & Build Process

Webpack optimizes and compiles the project for production (npm run build).

Static assets are bundled and minified for better performance.

The final build can be hosted on Firebase, Netlify, or any static server.

8. Deploying the Project

You can deploy the app using:

Firebase Hosting:

firebase deploy

Netlify:

Drag and drop the dist/ folder onto Netlify.

-

6. Create Article Form (components/vuejs/articles/create.vue)

A form that submits a new article to the store and API.

```
<template>
  <div>
    <h2>Create Article</h2>
    <form @submit.prevent="submitArticle">
      <input v-model="title" placeholder="Title" required />
      <input v-model="author" placeholder="Author" required />
      <button type="submit">Submit</button>
    </form>
  </div>
</template>

<script>
import { postData } from '@/core/xhr.js';

export default {
  data() {
    return { title: '', author: '' };
  },
  methods: {
    submitArticle() {
      postData('/api/articles', { title: this.title, author: this.author })
```

```
        .then(() => {
          this.title = '';
          this.author = '';
          alert('Article Created!');
        });
    }
  }
};
</script>
```

---

7. Webpack Configuration (build/webpack.base.conf.js)

Handles bundling, code splitting, and optimizations.

```
const path = require('path');

module.exports = {
  entry: './src/main.js',
  output: {
    path: path.resolve(__dirname, '../dist'),
    filename: 'bundle.js'
  },
  resolve: {
    alias: {
      '@': path.resolve(__dirname, '../src')
    }
  },
  module: {
    rules: [
      { test: /\.vue$/, loader: 'vue-loader' },
      { test: /\.js$/, loader: 'babel-loader', exclude: /node_modules/ }
    ]
  }
};
```

---

8. Environment Configurations (config/dev.env.js)

Defines settings for different environments (development, production).

```
module.exports = {
  NODE_ENV: '"development"',
  API_URL: '"http://localhost:5000/api"'
};
```

These code snippets highlight the core functionalities of InsightStream, including:

App initialization (main.js)

Routing (router/index.js)

State management (store/index.js)

API handling (core/xhr.js)

Vue components (dynamic UI rendering)

Build configuration (Webpack)

Environment settings (configurations)

Breakdown of Code Dependencies in InsightStream

The InsightStream project relies on several dependencies to function efficiently. These dependencies are primarily managed using npm and are listed in the package.json file. Below is a breakdown of the key dependencies:

---

1. Core Vue Dependencies

These packages provide the foundation for the Vue.js application.

---

2. Development & Build Tools

These tools assist in development, linting, and building the project.

---

3. UI & Styling

These dependencies are responsible for handling styles, animations, and UI components.

---

4. HTTP Requests & API Handling

Handles external API calls for fetching and sending data.

---

5. Environment Configuration

Used for defining environment-specific settings (development, production).

---

6. Production Optimization & Performance Enhancements

Used to improve performance in production environments.

---

7. Other Utilities

Additional tools that help in development and debugging.

How to Install Dependencies

To install all project dependencies, run:

```
npm install
```

How to Check Installed Dependencies

To list installed dependencies and their versions, use:

```
npm list --depth=0
```

How to Update Dependencies

To update all dependencies to their latest compatible versions:

```
npm update
```

For a specific package update:

```
npm install package-name@latest
```

The InsightStream project relies on these dependencies to efficiently manage:

✔ Vue-based UI development.

✔ State management with Vuex.

✔ Routing with Vue Router.

✔ API communication with Axios.

✔ Build optimization with Webpack.

✔ Styling with SCSS/SASS.

✔ Production-ready performance enhancements.

Libraries Used in InsightStream

The InsightStream project utilizes several libraries to enhance functionality, improve development efficiency, and optimize performance. Below is a breakdown of the key libraries used:

---

1. Core Vue.js Libraries

These libraries provide the foundational structure for the application.

---

2. HTTP & API Handling

Libraries used to communicate with external APIs.

---

3. Build & Development Tools

These libraries help with project bundling, compiling, and optimization.

---

4. UI & Styling Libraries

Libraries used for improving UI design and responsiveness.

---

5. Environment & Configuration Management

Handles different environments like development and production.

---

6. Performance Optimization

Libraries that enhance performance in production.

---

7. Additional Utility Libraries

Other libraries that support various functionalities.

How to Install These Libraries

All libraries are listed in package.json. To install them, run:

```
npm install
```

If you need to install a specific library manually, use:

```
npm install library-name
```

The InsightStream project relies on these libraries to:

- Build a robust Vue.js application.
- Optimize performance and state management with Vuex.
- Handle API requests with Axios.
- Manage routing efficiently with Vue Router.

Ensure production readiness with Webpack optimizations.

Fetching Data Functions in InsightStream

In the InsightStream project, data fetching is handled primarily using Axios and the built-in Fetch API. Below are key functions used to fetch and manage data from APIs.

6. Submitting Data in Vue with Vuex (components/vuejs/articles/create.vue)

This allows users to submit a new article using Vuex.

```
<template>
  <div>
    <h2>Create Article</h2>
    <form @submit.prevent="submitArticle">
      <input v-model="title" placeholder="Title" required />
      <input v-model="author" placeholder="Author" required />
      <button type="submit">Submit</button>
    </form>
  </div>
</template>

<script>
import { mapActions } from 'vuex';

export default {
  data() {
    return { title: '', author: '' };
  },
  methods: {
    ...mapActions(['createArticle']),
    submitArticle() {
      this.createArticle({ title: this.title, author: this.author })
        .then(() => {
          this.title = '';
          this.author = '';
          alert('Article Created!');
        })
        .catch(error => console.error('Failed to create article:', error));
    }
  }
};
</script>
```

Breakdown of API Endpoints and Keys in InsightStream

The InsightStream project interacts with a backend API to fetch and store data. Below is a breakdown of the key API endpoints and how they are used in the project.

---

1. API Base URL Configuration

The API base URL is stored in an environment file (config/dev.env.js) to differentiate between development and production environments.

```
module.exports = {
  NODE_ENV: '"development"',
  API_URL: '"http://localhost:5000/api"'
};
```

In production, the API URL is updated in config/prod.env.js:

```
module.exports = {
  NODE_ENV: '"production"',
  API_URL: '"https://api.insightstream.io"'
};
```

---

2. API Endpoint Breakdown

---

3. API Request Functions in core/xhr.js

a) Fetch All Articles

```
import axios from 'axios';
```

```
const API_URL = process.env.API_URL || 'http://localhost:5000/api';

export function getArticles() {
  return axios.get(${API_URL}/articles)
    .then(response => response.data)
    .catch(error => {
      console.error('Error fetching articles:', error);
      throw error;
    });
}
```

b) Fetch a Single Article by ID

```
export function getArticleById(articleId) {
  return axios.get(${API_URL}/articles/${articleId})
    .then(response => response.data)
    .catch(error => {
      console.error('Error fetching article:', error);
      throw error;
    });
}
```

c) Create a New Article

```
export function createArticle(articleData) {
  return axios.post(${API_URL}/articles, articleData)
    .then(response => response.data)
    .catch(error => {
      console.error('Error creating article:', error);
      throw error;
    });
}
```

d) Update an Existing Article

```
export function updateArticle(articleId, updatedData) {
  return axios.put(${API_URL}/articles/${articleId}, updatedData)
    .then(response => response.data)
    .catch(error => {
      console.error('Error updating article:', error);
      throw error;
    });
}
```

e) Delete an Article

```
export function deleteArticle(articleId) {
  return axios.delete(${API_URL}/articles/${articleId})
    .then(response => response.data)
    .catch(error => {
      console.error('Error deleting article:', error);
      throw error;
    });
}
```

---

4. Using API Functions in Vue Components

a) Fetch Articles in view.vue

```
<template>
  <div>
    <h2>Articles</h2>
    <ul v-if="articles.length">
      <li v-for="article in articles" :key="article.id">
        {{ article.title }} - {{ article.author }}
      </li>
    </ul>
    <p v-else>No articles found.</p>
  </div>
</template>

<script>
import { getArticles } from '@/core/xhr.js';

export default {
  data() {
    return { articles: [] };
  },
  created() {
    this.loadArticles();
  },
  methods: {
    loadArticles() {
      getArticles()
```

```
        .then(data => {
          this.articles = data;
        })
        .catch(error => console.error('Failed to load articles:', error));
    }
  }
};
</script>
```

---

b) Submit New Article in create.vue

```
<template>
  <div>
    <h2>Create Article</h2>
    <form @submit.prevent="submitArticle">
      <input v-model="title" placeholder="Title" required />
      <input v-model="author" placeholder="Author" required />
      <button type="submit">Submit</button>
    </form>
  </div>
</template>

<script>
import { createArticle } from '@/core/xhr.js';

export default {
  data() {
    return { title: '', author: '' };
  },
  methods: {
    submitArticle() {
      createArticle({ title: this.title, author: this.author })
        .then(() => {
          this.title = '';
          this.author = '';
          alert('Article Created!');
        })
        .catch(error => console.error('Failed to create article:', error));
    }
  }
};
</script>
```

---

5. API Key Usage

If the API requires an authentication key, it is stored securely in an environment variable (.env file) and accessed dynamically.

Storing the API Key Securely in .env

```
VUE_APP_API_KEY=your_api_key_here
```

Using the API Key in Requests (core/xhr.js)

```
const API_KEY = process.env.VUE_APP_API_KEY;

export function getProtectedData(endpoint) {
  return axios.get(${API_URL}/${endpoint}, {
    headers: { 'Authorization': Bearer ${API_KEY} }
  })
    .then(response => response.data)
    .catch(error => {
      console.error('Error fetching protected data:', error);
      throw error;
    });
}
```

Summary of API Endpoints & Functions

The InsightStream project uses: A modular API structure to manage data.

Axios-based API requests for efficiency.

Vue components that dynamically interact with the API.

Environment-based API configuration for security.

Error Handling in InsightStream

Error handling in the InsightStream project ensures that API failures, validation issues, and unexpected bugs are managed effectively. Below is a breakdown of how errors are handled across different parts of the application.

---

1. Centralized Error Handling (core/errors.js)

A dedicated module for handling API and application errors.

```
export class AppError extends Error {
  constructor(message, status = 500) {
    super(message);
    this.name = 'AppError';
    this.status = status;
  }
}

// Handle API errors globally
export function handleApiError(error) {
  if (error.response) {
    // Server responded with an error status
    console.error(API Error (${error.response.status}):, error.response.data);
    return new AppError(error.response.data.message || 'Server error',
error.response.status);
  } else if (error.request) {
    // Request was made but no response received
    console.error('API Error: No response from server', error.request);
    return new AppError('No response from server', 503);
  } else {
    // Other errors (network issues, wrong configurations, etc.)
    console.error('Unexpected API Error:', error.message);
    return new AppError(error.message, 500);
  }
}
```

---

2. Handling Errors in API Requests (core/xhr.js)

All API calls use try-catch to ensure errors are handled properly.

```
import axios from 'axios';
import { handleApiError } from './errors.js';

const API_URL = process.env.API_URL || 'http://localhost:5000/api';

// GET Request with Error Handling
export async function getData(endpoint) {
  try {
    const response = await axios.get(${API_URL}/${endpoint});
    return response.data;
  } catch (error) {
    throw handleApiError(error);
  }
}
```

```
// POST Request with Error Handling
export async function postData(endpoint, data) {
  try {
    const response = await axios.post(${API_URL}/${endpoint}, data);
    return response.data;
  } catch (error) {
    throw handleApiError(error);
  }
}
```

---

3. Handling API Errors in Vue Components (view.vue)

When fetching articles, error handling ensures that users get a meaningful message.

```
<template>
  <div>
    <h2>Articles</h2>
    <p v-if="errorMessage" class="error">{{ errorMessage }}</p>
    <ul v-else-if="articles.length">
      <li v-for="article in articles" :key="article.id">
        {{ article.title }} - {{ article.author }}
      </li>
    </ul>
    <p v-else>Loading articles...</p>
  </div>
</template>

<script>
import { getData } from '@/core/xhr.js';

export default {
  data() {
    return { articles: [], errorMessage: '' };
  },
  created() {
    this.fetchArticles();
  },
  methods: {
    async fetchArticles() {
      try {
        this.articles = await getData('articles');
      } catch (error) {
        this.errorMessage = error.message;
      }
    }
  }
};
</script>

<style>
.error {
  color: red;
  font-weight: bold;
}
</style>
```

---

4. Form Validation & Handling Errors in create.vue

Ensures user input is properly validated before submitting to the server.

```
<template>
  <div>
    <h2>Create Article</h2>
    <form @submit.prevent="submitArticle">
      <input v-model="title" placeholder="Title" required />
      <input v-model="author" placeholder="Author" required />
      <button type="submit">Submit</button>
      <p v-if="errorMessage" class="error">{{ errorMessage }}</p>
    </form>
  </div>
</template>

<script>
import { postData } from '@/core/xhr.js';

export default {
  data() {
    return { title: '', author: '', errorMessage: '' };
  },
  methods: {
    async submitArticle() {
      if (!this.title || !this.author) {
        this.errorMessage = 'All fields are required!';
        return;
      }

      try {
        await postData('articles', { title: this.title, author: this.author });
        alert('Article Created!');
        this.title = '';
        this.author = '';
        this.errorMessage = '';
      } catch (error) {
        this.errorMessage = error.message;
      }
    }
  }
};
</script>
```
---

5. Handling Global Vue Errors (main.js)

To catch and log unexpected errors across the application.

```
Vue.config.errorHandler = (err, vm, info) => {
  console.error('Vue Global Error:', err, info);
  alert('An unexpected error occurred. Please try again later.');
};
```

---

6. Handling Errors in Vuex (store/index.js)

Vuex actions should catch API errors to avoid breaking the UI.

```
import Vue from 'vue';
import Vuex from 'vuex';
import { getData, postData } from '@/core/xhr.js';

Vue.use(Vuex);

export default new Vuex.Store({
  state: {
    articles: [],
    error: null
  },
  mutations: {
    setArticles(state, articles) {
      state.articles = articles;
    },
    setError(state, error) {
      state.error = error;
    }
  },
  actions: {
    async fetchArticles({ commit }) {
      try {
        const data = await getData('articles');
        commit('setArticles', data);
      } catch (error) {
        commit('setError', error.message);
      }
    },
    async createArticle({ commit }, article) {
      try {
        await postData('articles', article);
      } catch (error) {
        commit('setError', error.message);
      }
    }
  }
});
```

7. Displaying Errors in the UI (App.vue)

Show error messages from Vuex across all pages.

```
<template>
  <div>
    <h1>InsightStream</h1>
    <p v-if="error" class="error">{{ error }}</p>
    <router-view />
  </div>
</template>
```

```
<script>
import { mapState } from 'vuex';

export default {
  computed: mapState(['error'])
};
</script>

<style>
.error {
  color: red;
  background: #ffe0e0;
  padding: 10px;
  border: 1px solid red;
  margin: 10px 0;
}
</style>
```

Fetching Related Videos Functions in InsightStream

If your InsightStream project includes a feature for fetching related videos, you would typically retrieve video recommendations from an API. Below is a breakdown of how to implement related video fetching functions.

4. Fetching Related Videos in Vuex (store/index.js)

If you want to manage related videos in Vuex instead of fetching them in the component directly:

```
import Vue from 'vue';
import Vuex from 'vuex';
import { getRelatedVideos } from '@/core/xhr.js';

Vue.use(Vuex);

export default new Vuex.Store({
  state: {
    relatedVideos: []
  },
  mutations: {
    setRelatedVideos(state, videos) {
      state.relatedVideos = videos;
    }
  },
  actions: {
    async fetchRelatedVideos({ commit }, videoId) {
      try {
        const data = await getRelatedVideos(videoId);
        commit('setRelatedVideos', data);
      } catch (error) {
        console.error('Error fetching related videos:', error);
      }
    }
  }
});
```

---

5. Displaying Related Videos Using main.json

Instead of fetching data inside the component, you can map Vuex state and actions:

```
<template>
  <div>
    <h3>Related Videos</h3>
    <ul v-if="relatedVideos.length">
      <li v-for="video in relatedVideos" :key="video.id">
        <a :href="'/videos/' + video.id">
          <img :src="video.thumbnail" :alt="video.title" />
          <p>{{ video.title }}</p>
        </a>
      </li>
    </ul>
    <p v-else>No related videos found.</p>
  </div>
</template>

<script>
import { mapState, mapActions } from 'vuex';

export default {
  props: ['videoId'],
  computed: mapState(['relatedVideos']),
  watch: {
    videoId: 'fetchRelatedVideos'
  },
  methods: {
    ...mapActions(['fetchRelatedVideos'])
  },
  created() {
    this.fetchRelatedVideos(this.videoId);
  }
};
</script>
```

---

6. Handling Errors in Related Video Fetching

To ensure smooth user experience, error handling can be added:

```
export function handleApiError(error) {
  if (error.response) {
    console.error(API Error (${error.response.status}):, error.response.data);
    return new Error(error.response.data.message || 'Server error');
  } else if (error.request) {
    console.error('API Error: No response from server', error.request);
    return new Error('No response from server');
  } else {
    console.error('Unexpected API Error:', error.message);
    return new Error(error.message);
  }
}
```

---

7. Summary of Related Video Fetching

The InsightStream project efficiently fetches and displays related videos using:

- Axios-based API calls for fetching video recommendations.
- Vue components that dynamically update based on the current video.
- Vuex for centralized state management if needed.
- Error handling for smooth user experience.

API Configuration in InsightStream

The InsightStream project follows a structured API configuration to manage environment-specific settings, authentication, and error handling efficiently.

---

1. API Base URL Configuration

The API endpoint varies depending on the environment (development, testing, or production).

Environment Configurations (config/index.js)

```
module.exports = {
  development: {
    API_URL: 'http://localhost:5000/api',
    API_KEY: 'your_dev_api_key'
  },
  production: {
    API_URL: 'https://api.insightstream.io',
    API_KEY: 'your_prod_api_key'
  }
};
```

---

2. Accessing API Configurations in Code

API settings can be dynamically imported based on the current environment.

```
const env = process.env.NODE_ENV || 'development';
const config = require('../config/index.js')[env];

export const API_URL = config.API_URL;
export const API_KEY = config.API_KEY;
```

---

3. API Request Setup with (core/xhr.js)

This file centralizes all API requests and ensures proper configuration.

```
import axios from 'axios';
import { API_URL, API_KEY } from '../config/apiConfig.js';

const apiClient = axios.create({
  baseURL: API_URL,
  headers: {
    'Content-Type': 'application/json',
    'Authorization': Bearer ${API_KEY}
  }
});

// Generic GET request
export async function getData(endpoint) {
  try {
    const response = await apiClient.get(endpoint);
    return response.data;
  } catch (error) {
    throw handleApiError(error);
  }
}

// Generic POST request
export async function postData(endpoint, data) {
  try {
    const response = await apiClient.post(endpoint, data);
    return response.data;
  } catch (error) {
    throw handleApiError(error);
  }
}

// Error handling function
function handleApiError(error) {
  console.error('API Error:', error);
  throw new Error(error.response?.data?.message || 'API request failed');
}
```

---

4. Using API in Vue Components

Vue components import API functions from xhr.js to fetch or send data.

Fetching Data in a Component (components/articles/view.vue)

```
<template>
  <div>
    <h2>Articles</h2>
    <ul v-if="articles.length">
      <li v-for="article in articles" :key="article.id">
        {{ article.title }} - {{ article.author }}
      </li>
    </ul>
    <p v-else>No articles found.</p>
  </div>
</template>

<script>
import { getData } from '@/core/xhr.js';

export default {
  data() {
    return { articles: [] };
  },
  created() {
    this.fetchArticles();
  },
  methods: {
    async fetchArticles() {
      try {
        this.articles = await getData('articles');
      } catch (error) {
        console.error(error.message);
      }
    }
  }
};
</script>
```

5. Authentication Configuration

If the API requires authentication, tokens can be stored and passed in requests.

Storing API Tokens in Local Storage (core/auth.js)

```
export function setAuthToken(token) {
  localStorage.setItem('auth_token', token);
}

export function getAuthToken() {
  return localStorage.getItem('auth_token');
}

export function removeAuthToken() {
  localStorage.removeItem('auth_token');
}
```

Adding Auth Token to Requests (core/xhr.js)

```
import { getAuthToken } from './auth.js';

const apiClient = axios.create({
  baseURL: API_URL,
  headers: {
    'Content-Type': 'application/json'
  }
});

// Add auth token dynamically to every request
apiClient.interceptors.request.use((config) => {
  const token = getAuthToken();
  if (token) {
    config.headers['Authorization'] = Bearer ${token};
  }
  return config;
}, (error) => Promise.reject(error));
```

---

6. Summary of API Configuration

The InsightStream project has:

Centralized API configuration for flexibility.

Secure authentication handling via local storage.

Dynamic API requests using Axios.

Environment-based settings for smooth deployment.

Project Execution for InsightStream

To successfully execute the InsightStream project, follow these steps for setup, running, and deployment.

---

1. Prerequisites

Before running the project, ensure you have the following installed:

- Node.js & npm – Download Node.js
- Vue CLI – Install globally:

```
npm install -g @vue/cli
```

- Code Editor – Recommended: VS Code
- Git (Optional) – For version control (Download Git)

---

2. Cloning or Downloading the Project

Option 1: Clone from GitHub

```
git clone https://github.com/your-repo/insightstream.io-dev.git
cd insightstream.io-dev
```

Option 2: Download ZIP from Google Drive

Extract the ZIP file.

Open the folder in your terminal or VS Code.

---

3. Installing Dependencies

Run the following command inside the project folder to install required packages:

```
npm install
```

---

4. Configuring API & Environment Variables

Edit the .env file or update config/index.js with API settings:

```
VUE_APP_API_URL=http://localhost:5000/api
VUE_APP_API_KEY=your_api_key_here
```

---

5. Running the Development Server

Start the Vue.js development server:

```
npm run serve
```

The application will be available at:
http://localhost:8080/

If the port is occupied, specify a different port:

```
npm run serve -- --port 3000
```

---

6. Building for Production

To generate an optimized build, run:

```
npm run build
```

This creates a dist/ folder containing the production-ready files.

--

7. Deploying the Project

Option 1: Deploy to Firebase Hosting

1. Install Firebase CLI:

```
npm install -g firebase-tools
```

2. Login to Firebase:

```
firebase login
```

3. Initialize Firebase in the project:

```
firebase init
```

4. Deploy the project:

```
firebase deploy
```

Option 2: Deploy to Netlify

1. Install Netlify CLI:

```
npm install -g netlify-cli
```

2. Deploy:

```
netlify deploy --prod
```

8. Testing the Project

Open http://localhost:8080/ (development mode)

Check Console & Network tab in the browser for errors

Test API connections in Postman or cURL

9. Common Issues & Fixes

10. Summary of Project Execution

The InsightStream project is now ready to run and deploy.

Hero Component in InsightStream

The Hero Component is a visually appealing section typically used on the homepage to grab user attention. It often includes a title, subtitle, background image, and a call-to-action (CTA) button.

1. Hero Component Structure (components/shared/Hero.vue)

```
<template>
  <section class="hero">
    <div class="hero-content">
      <h1>{{ title }}</h1>
      <p>{{ subtitle }}</p>
      <button @click="ctaAction">{{ buttonText }}</button>
    </div>
  </section>
</template>

<script>
export default {
  props: {
    title: {
      type: String,
      default: 'Welcome to InsightStream'
    },
    subtitle: {
      type: String,
      default: 'Your source for insightful content and videos'
    },
    buttonText: {
      type: String,
      default: 'Get Started'
    }
  },
  methods: {
    ctaAction() {
      this.$router.push('/explore'); // Redirects to the explore page
    }
  }
};
</script>

<style scoped>
.hero {
  display: flex;
  align-items: center;
  justify-content: center;
  height: 60vh;
  background: url('@/assets/hero-bg.jpg') no-repeat center center/cover;
  text-align: center;
  color: white;
}
.hero-content {
  background: rgba(0, 0, 0, 0.6);
  padding: 20px;
  border-radius: 10px;
}
h1 {
  font-size: 3rem;
  margin-bottom: 10px;
}
p {
  font-size: 1.5rem;
  margin-bottom: 20px;
```

```
}
button {
  background: #ff6600;
  color: white;
  padding: 10px 20px;
  border: none;
  cursor: pointer;
  font-size: 1.2rem;
  border-radius: 5px;
}
button:hover {
  background: #cc5200;
}
</style>
```

2. Using the Hero Component in a Page (pages/Home.vue)

```
<template>
  <div>
    <Hero title="Discover Amazing Insights" subtitle="Explore articles and videos curated just for you" buttonText="Browse Now" />
  </div>
</template>

<script>
import Hero from '@/components/shared/Hero.vue';

export default {
  components: {
    Hero
  }
};
</script>
```

3. Features of the Hero Component

- Dynamic props – Customizable title, subtitle, and button text.
- Background image – Set using CSS.
- Call-to-Action (CTA) button – Redirects to another page.
- Reusable – Can be placed on different pages.