ChatGPT

Front-End UI Design for Job Seeker Intelligent Agent

Overview:
This plan outlines a Vue3 front-end architecture for the Job Seeker Intelligent Agent project. It is designed from the **job seeker’s perspective**, providing a smooth flow from uploading a resume to viewing job matches and detailed analysis. The implementation uses **Vue 3 + Vite + Pinia + Vue Router + Axios**, with **Tailwind CSS** for styling. The UI will be clean and professional, featuring a white background and grey-blue accent colors (blue for primary buttons/links) for a simple, modern look. Internationalization (i18n) is supported for English and Chinese. Pinia is used for state management (configured in `main.js` with `createPinia()` [1]) to share data (like the uploaded resume and matches) across components. Below we detail the page structure, core components, routing, i18n, and API integration, followed by an example code snippet.

UI Page Structure

We propose four main pages under `src/pages/` to cover the job seeker user journey:

- **`Home.vue`** – **Home Page:** Introduces the platform and guides the user on how to use the service. This landing page has a brief overview of the AI job matching features and a call-to-action (e.g. **“Upload Your Resume”** button) to start the process. It may also include an optional search bar for job keywords (using the knowledge base API) and a language switcher in the header for locale toggle. The style is welcoming and simple, with a brief introduction and a clear next step.

- **`Upload.vue`** – **Resume Upload Page:** Allows the user to select or drag-and-drop a resume file for upload. The page uses a **ResumeUpload** component (see below) to handle the file input. After the file is uploaded (via the `/resume/upload` API), the page displays a **preview of the parsed resume** data (e.g. name, email, key skills, experiences) so the user can verify it [2]. A “Find Matching Jobs” button is provided to proceed. The layout is centered with a file drop area styled with Tailwind (e.g. a dashed border box with instructions). On successful upload, the component emits the structured resume JSON which is stored (via Pinia) and some basic info is shown as confirmation. The user can then click to navigate to the recommendations page.

- **`Recommend.vue`** – **Matching Recommendations Page:** After uploading a resume, this page displays a list of **recommended job postings** that best match the candidate. Each job is shown as a **JobCard** component with key details: job title, company name, location, application deadline, and a match score (or percentage) indicating relevance [3]. A short snippet of the job description or matching rationale is included for context (e.g. from the `snippet` field in the API response) [3]. At the top of the page, a **MatchSummary** component can display an overall summary sentence from the AI about the candidate’s fit for these jobs (using the `summary` field from the `/match/auto` API) [4]. The jobs are typically listed as cards in a grid or list; each card also provides a link or button to view a **detailed report**. The page pulls the recommendations either from a Pinia store (if pre-fetched) or by calling the `/match/auto` API using the stored resume filename. It also handles loading states and error messages (e.g. if no matches found or API error).

- `Report.vue` – **Match Analysis Report Page:** This page shows an in-depth analysis of how the uploaded resume matches a specific job. It is accessed when a user selects a job from the recommendations. The frontend calls the `/match/single` API with the resume file and job ID to get a detailed analysis and an HTML report path [5]. The page then **renders the HTML report** (for example, by fetching the HTML content from the `report_path` or embedding it). The report (generated by the LLM) includes sections like skill comparisons, experience match, and a compatibility score (e.g. similarity score) [5]. The UI may show the job title, company, and similarity score at the top, and then the formatted report content below. We can use a container with scroll if the report is long. There is also a back link ("← Back to Recommendations") to return to the list. The **LanguageSwitcher** remains available to toggle the report language if needed (assuming the report content can be in the chosen language).

Directory Structure: The pages and components are organized clearly for team development. Below is a suggested project structure highlighting key files:

```
src/
 ├── pages/
 │    ├── Home.vue          # Home introduction & guide page
 │    ├── Upload.vue        # Resume upload page
 │    ├── Recommend.vue     # Job recommendations list page
 │    └── Report.vue        # Detailed match report page
 ├── components/
 │    ├── ResumeUpload.vue  # Component for file upload & preview
 │    ├── JobCard.vue       # Component to display a job info card
 │    ├── MatchSummary.vue  # Component for the summary of matches
 │    └── LanguageSwitcher.vue  # Locale toggle dropdown/button
 ├── store/
 │    └── resume.js         # Pinia store (e.g. current resume & matches)
 ├── api/
 │    ├── upload.js         # Axios calls for resume upload
 │    └── match.js          # Axios calls for job matching & reports
 ├── i18n/
 │    ├── en.json           # English language messages
 │    └── zh.json           # Chinese language messages
 └── router/
     └── index.js          # Vue Router configuration
```

(Additional utility modules, assets, etc. are omitted for brevity.)

Core Components Design

To keep the codebase modular and maintainable, we factor common UI pieces into reusable components under `src/components/`. Here are the core components and their roles:

- **ResumeUpload.vue:** This component provides the UI and logic for uploading a resume file. It includes an `<input type="file">` (hidden or styled as needed) and a drag-and-drop dropzone area. Using Tailwind, the dropzone might be a centered box with a gray dashed border (e.g. `border-2 border-dashed border-gray-300 p-6 text-center`) and hover style. The component handles events for file selection or drop. On file selection, it calls the **upload API** (`POST /resume/`

`upload`) via an Axios wrapper and awaits the response. While uploading, it can show a loading indicator. On success, it emits an event (e.g. `@uploaded`) carrying the returned **Resume JSON** (which includes the filename and parsed data) [2]. It can also internally display a brief preview of the parsed data (for example, listing the name, email, and top skills extracted) so that the user gets immediate feedback. The preview could be a simple list or table of the `resume_data` fields [2]. If the file format is unsupported or parsing fails (API returns error 400 or 500), the component should display an error message (using the error detail from the response). This component is used within the Upload page.

- **JobCard.vue:** This component represents a single job recommendation entry in the list. It accepts props such as job title, company, location, deadline, match score, and snippet text. The card is styled with a light background (e.g. white or gray-50) and a subtle border (rounded corners, shadow on hover for emphasis). Key information is displayed prominently: the **job title** and **company** at top (possibly with company logo if available), the **match score** as a percentage or rating (could be a badge or progress bar indicating the score, e.g. 87% match), and the snippet of the job description or AI commentary below. For example, a card might show “**Data Analyst** at ACME (Shanghai) – 87% match” and a snippet like “... looking for Python and SQL experience ...” derived from the `snippet` field [3]. A **“View Details”** button or link on the card directs the user to the Report page for that job. The component should be responsive (stacking content on small screens). It can also highlight the match score in the accent color (blue) for visibility.

- **MatchSummary.vue:** This component displays the overall summary of the matching results, typically shown at the top of the Recommend page. It takes a summary text (from the `/match/auto` API’s `summary` field) which might say, for example, “候选人与岗位高度匹配，具备突出的数据分析技能...” (“The candidate is highly matched with these positions, possessing strong data analysis skills...”) [4]. The component simply renders this text in an emphasized style (italic or a distinct color) or inside a stylized container (e.g. an info alert box). If using multiple languages, the summary text would be returned from the API in the current locale (or the component could handle translating static parts if needed). This component helps separate the summary display logic from the main page and could be reused or expanded in the future (for instance, to show different types of summaries or tips).

- **LanguageSwitcher.vue:** A small component (e.g. a dropdown or toggle button) to switch the interface language between Chinese and English. It likely appears in the header or footer of the app (maybe included in a global NavBar or directly in each page as needed). The simplest implementation is a select box or toggle that binds to the i18n locale. For example, using Vue I18n’s global locale: `<select v-model="$i18n.locale"> ... </select>` [6] to switch between available locales. We will provide two options (e.g. “English” and “中文”) or use `$i18n.availableLocales` to generate them dynamically [6]. When the user selects a language, all text on the UI updates immediately (thanks to Vue I18n reactivity). The LanguageSwitcher ensures that labels like “Resume Upload” or “Match Score” are shown in the chosen language by using localization keys (discussed below). We might also persist the selected language in localStorage or in a cookie so that the preference is saved on revisit.

Each component will be documented and structured for reusability. They will use Pinia stores or emit events to communicate with pages as needed (e.g. ResumeUpload emits the parsed data upward).

Routing Configuration

We define client-side routes using **Vue Router** (in `src/router/index.js`) to map URLs to the above pages. The routing setup uses history mode (for clean URLs) and includes the following routes:

- `/` → **Home:** Displays the Home.vue component. This is the landing page.
- `/upload` → **Upload:** Displays the Upload.vue component for resume upload.
- `/recommend` → **Recommend:** Displays the Recommend.vue component, showing job matches. This page is typically navigated to after a successful upload. We assume one resume at a time (for simplicity), so this route shows matches for the last uploaded resume (stored in state).
- `/report/:resume/:job` → **Report:** Displays the Report.vue component for a specific resume and job combination. The route includes dynamic params: `resume` (which could be a resume identifier or filename) and `job` (job ID). For example, after choosing a job with ID `job_101` for resume file `resume_张三.json`, the app might navigate to `/report/resume_张三.json/job_101`. The Report component will use these `$route.params` to fetch the correct analysis via API. In the router config, this route can be named (e.g. name: 'report') for convenience. We will also ensure that if a user tries to access this route without data, the app can redirect them (e.g. if no resume is loaded, redirect to Home).

Route definitions might look like:

```
// router/index.js
import { createRouter, createWebHistory } from 'vue-router';
import Home from '@/pages/Home.vue';
import Upload from '@/pages/Upload.vue';
import Recommend from '@/pages/Recommend.vue';
import Report from '@/pages/Report.vue';

const routes = [
  { path: '/', name: 'home', component: Home },
  { path: '/upload', name: 'upload', component: Upload },
  { path: '/recommend', name: 'recommend', component: Recommend },
  { path: '/report/:resume/:job', name: 'report', component: Report, props: true }
];

export default createRouter({
  history: createWebHistory(),
  routes
});
```

We use history mode for clean URLs (assuming proper server setup for SPA fallback). During development, Vite’ s dev server can proxy API calls to the FastAPI backend (e.g. configure `server.proxy` in `vite.config.js` for paths like `^/resume` or `^/match` to `http://localhost:8000`) so that API calls don’ t face CORS issues and no manual prefix is needed. In production, the front-end could be served from the same domain or the proxy can be configured similarly.

Internationalization (i18n) Support

We will implement **multi-language support** using Vue I18n (the Intlify Vue 3 version). The app will support English (`en`) and Simplified Chinese (`zh`) locales. The localization setup involves:

- **Messages resources:** We create language files under `src/i18n/` such as `en.json` and `zh.json` (or `en.ts/js` exporting an object). These files contain translation keys for all text in the UI. For example:

`en.json`:

```
{
 "home": {
  "welcome": "Welcome to the AI Job Matching Portal",
  "description": "Upload your resume to get started with intelligent job recommendations."
 },
 "upload": {
  "pageTitle": "Resume Upload",
  "instructions": "Drag & drop your resume file here, or click to select a file.",
  "uploadButton": "Upload Resume",
  "viewRecommendations": "View Matching Jobs"
 },
 "recommend": {
  "pageTitle": "Recommended Jobs",
  "summaryTitle": "Summary",
  "scoreLabel": "Match Score",
  "viewDetail": "View Details"
 },
 "report": {
  "pageTitle": "Match Analysis Report",
  "backButton": "← Back to Recommendations"
 }
}
```

`zh.json`:

```
{
 "home": {
  "welcome": "欢迎使用AI岗位匹配平台",
  "description": "上传您的简历，开始获取智能岗位推荐。"
 },
 "upload": {
  "pageTitle": "简历上传",
  "instructions": "将简历文件拖拽到此处，或点击选择文件。",
  "uploadButton": "上传简历",
  "viewRecommendations": "查看匹配岗位"
 },
 "recommend": {
  "pageTitle": "推荐岗位列表",
    "summaryTitle": "匹配摘要",
    "scoreLabel": "匹配度",
    "viewDetail": "查看详情"
  },
  "report": {
    "pageTitle": "匹配分析报告",
    "backButton": "← 返回推荐列表"
  }
}
```

These keys cover UI text like page titles, button labels, and important terms. We include terms such as "Resume Upload", "Match", "Recommendation", etc., translated appropriately.

- **Vue I18n setup:** In `main.js`, we initialize Vue I18n and load the message files. For example:

  ```
  import { createI18n } from 'vue-i18n';
  import zh from '@/i18n/zh.json';
  import en from '@/i18n/en.json';
  const i18n = createI18n({
    locale: 'zh', // default to Chinese or detect browser preference
    fallbackLocale: 'en',
    messages: { en, zh }
  });
  const app = createApp(App);
  app.use(i18n);
  app.use(pinia);
  app.mount('#app');
  ```

  This makes the `$t` function available in components to fetch translations.

- **Usage in components:** All static text in the pages/components will use `$t('key.path')`. For instance, the Upload page heading might be: `<h1>{{ $t('upload.pageTitle') }}</h1>`. The upload button text would be `$t('upload.uploadButton')`. In the JobCard, labels like "Match Score" can be `$t('recommend.scoreLabel')`. This ensures the UI can switch languages seamlessly.

- **Language switcher:** The **LanguageSwitcher** component (as described) provides a UI to change the locale. We bind the selected value to `$i18n.locale` for global scope switching [6]. For example:

  ```
  <template>
    <select v-model="$i18n.locale" class="language-switcher">
      <option value="en">English</option>
      <option value="zh">中文</option>
    </select>
  </template>
  ```

Vue I18n will reactively update text across the app when this changes. We will position this switcher in a convenient spot (top-right of nav bar or footer). Optionally, use icons or flags for a nicer UI.

By maintaining separate language packs, the team can easily update translations. New UI text must have keys added to both files to keep parity. The system can be extended to other languages in the future if needed.

Axios API Integration

All API calls to the FastAPI backend are handled through a centralized module using **Axios**. We will create an Axios instance with a base URL pointing to the backend (e.g. `http://localhost:8000`) or use relative paths with Vite’s proxy. For example, an `api/axiosInstance.js` could configure Axios:

```
// api/axiosInstance.js
import axios from 'axios';
const instance = axios.create({
  baseURL: import.meta.env.VITE_API_BASE_URL || 'http://localhost:8000',  // backend URL
  timeout: 10000  // optional: request timeout
});
export default instance;
```

(Using a Vite env variable `VITE_API_BASE_URL` allows easy switching between dev/prod URLs [7].)

We then create modules for different API groupings. In particular:

- **Resume Upload API (src/api/upload.js):** Contains the function to upload a resume file.

```
import api from './axiosInstance';
// POST /resume/upload – upload resume and get structured data
export function uploadResume(file) {
  const formData = new FormData();
  formData.append('file', file);
  // call backend to upload and parse resume
  return api.post('/resume/upload', formData);
}
```

  This returns a Promise that resolves to a **ResumeJson** object. The structure of this object (as defined by the backend) includes the original filename, the JSON filename saved on server, and the parsed `resume_data` with fields like basic_info, skills, experience, etc. [2]. For example, the response might look like:

```
{
  "filename": "resume_张三.pdf",
  "json_file": "data/uploads/resume_张三.json",
  "resume_data": { "basic_info": {…}, "skills": […], "experience": […] }
}
```

In the front-end code, we await this call and then use `response.data.resume_data` to get the structured info for preview. (We may also store the `json_file` name, which will be needed for match queries.)

- **Match Recommendation APIs (src/api/match.js):** Contains functions for job matching and reports.

```
import api from './axiosInstance';
// GET /match/auto – get job recommendations for a given resume JSON
export function getRecommendations(resumeFile) {
 // resumeFile is the filename or path returned by upload (e.g. "resume_张三.json")
 return api.get('/match/auto', { params: { resume_file: resumeFile } });
}

// GET /match/single – get detailed match analysis for one job
export function getMatchReport(resumeFile, jobId) {
 return api.get('/match/single', { params: { resume_file: resumeFile, job_id: jobId } });
}
```

The **getRecommendations** call returns a JSON containing an array of recommended jobs and a summary [8] [4]. Each job item includes fields like `job_id`, `title`, `company`, `location`, `deadline`, and a matching `score` and `snippet` [3]. For example, a single recommendation might be:

```
{
 "score": 0.87,
 "job_id": "job_101",
 "title": "数据分析师",
 "company": "ACME",
 "location": "上海",
 "deadline": "2025-01-10",
 "snippet": "公司名称: ACME…"
}
```

and the response also includes a `summary` like `"候选人与岗位高度匹配..."` [4]. We will define a **JobMatch** type/interface to represent these fields for clarity.

The **getMatchReport** call returns a JSON with detailed analysis for the specified job and resume [5]. This includes fields such as `job_title`, `company`, `location`, a `similarity_score` (numerical match metric), an `analysis` text summary, and a `report_path` which points to an HTML file with the full report [5]. The front end can use `analysis` for a quick text summary and then fetch or embed the HTML from `report_path` for the full content. We will define a **MatchReport** type for these results.

All API functions return **Promises** (or could use async/await in components). We will handle errors globally – e.g., use Axios interceptors to catch HTTP errors and show notifications. The API module ensures the front-end uses consistent endpoints exactly as defined by the backend documentation (e.g., passing `resume_file` and `job_id` correctly). By encapsulating in functions, if the backend changes (e.g., different param names), we update in one place.

Additionally, the Axios instance could include default headers or interceptors for things like loading spinners or error messages. Since the current backend doesn’ t require auth, no auth headers are needed (if later added, we could integrate token handling in this layer).

Sample Code – Upload Page & Component Example

Below is an example snippet illustrating the **Resume Upload page (Upload.vue)** and how it uses the **ResumeUpload** component and Pinia store. This shows the structure and integration of various pieces described above (Tailwind classes for styling, i18n for text, event handling, and routing):

```
<!-- src/pages/Upload.vue -->
<template>
  <div class="max-w-2xl mx-auto p-6">
    <!-- Page Title -->
    <h1 class="text-2xl font-bold mb-4">{{ $t('upload.pageTitle') }}</h1>
    <!-- Upload Component -->
    <ResumeUpload @uploaded="handleUploaded" />
    <!-- Preview of parsed resume (shown after upload succeeds) -->
    <div v-if="resumeData" class="mt-6 p-4 bg-gray-50 border border-gray-200 rounded">
      <h2 class="text-xl font-semibold mb-2">{{ resumeData.basic_info.name || 'Unnamed' }}</h2>
      <p class="mb-1"><strong>Email:</strong> {{ resumeData.basic_info.email }}</p>
      <p class="mb-1"><strong>{{ $t('recommend.scoreLabel') }}:</strong>
          <!-- Example: show number of skills for demo -->
          {{ resumeData.skills.length }} Skills Extracted
      </p>
      <!-- ... other preview info ... -->
    </div>
    <!-- Button to view recommendations (visible after upload) -->
    <div v-if="resumeData" class="mt-4">
      <router-link :to="{ name: 'recommend' }" class="bg-blue-600 text-white px-4 py-2 rounded">
        {{ $t('upload.viewRecommendations') }}
      </router-link>
    </div>
  </div>
</template>

<script setup>
import { ref } from 'vue';
import { useRouter } from 'vue-router';
import { useResumeStore } from '@/store/resume';
import ResumeUpload from '@/components/ResumeUpload.vue';

const router = useRouter();
const resumeStore = useResumeStore();
const resumeData = ref(null);

// Handle event when ResumeUpload component finishes uploading and parsing
function handleUploaded(result) {
 // `result` is the parsed resume JSON returned by API (ResumeJson)
 resumeData.value = result.resume_data;
 // Save the full resume info (including file name) to Pinia store for access in other pages
 resumeStore.setResume(result);
 // (Optional) We could auto-fetch recommendations here instead of on the next page.
 // For example:
 // await resumeStore.fetchRecommendations();  (assuming store action calls getRecommendations
API)
 // Then navigate to recommendations page:
 // router.push({ name: 'recommend' });
}
</script>
```

```
<!-- src/components/ResumeUpload.vue (simplified example) -->
<template>
  <div  class="border-2 border-dashed border-gray-300 p-8 text-center">
    <input  type="file"  ref="fileInput"  class="hidden"  accept=".pdf,.docx,.txt"
@change="onFileChange"  />
    <p  class="mb-4">{{ $t('upload.instructions') }}</p>
    <button  class="bg-blue-600 text-white px-4 py-2 rounded"  @click="triggerFileSelect">
      {{ $t('upload.selectFile') || 'Choose File' }}
    </button>
    <!-- Display upload progress or status -->
    <p  v-if="uploading"  class="mt-2 text-gray-500">{{ uploadStatus }}</p>
    <p  v-if="error"  class="mt-2 text-red-600">{{ error }}</p>
  </div>
</template>

<script  setup>
import { ref } from 'vue';
import { uploadResume } from '@/api/upload';  // API function for uploading file

const fileInput = ref(null);
const uploading = ref(false);
const uploadStatus = ref('');
const error = ref('');

function triggerFileSelect() {
 fileInput.value.click();
}

async function onFileChange(event) {
 const files = event.target.files;
 if (!files || files.length === 0) return;
 const file = files[0];
 uploading.value = true;
 uploadStatus.value = 'Uploading...';
 error.value = '';
 try {
  const response = await uploadResume(file);  // call API
  const result = response.data;
  uploadStatus.value = 'Upload successful';
```

```
    uploading.value = false;
    // Emit the parsed resume JSON up to the parent
    emit('uploaded', result);
  } catch (err) {
    uploading.value = false;
    uploadStatus.value = '';
    error.value = err.response?.data?.detail || 'Upload failed';
  }
}
</script>
```

In this example:

- We use Tailwind classes (`max-w-2xl mx-auto` to center the content with a max width, `bg-gray-50` for a light background, etc.) to achieve the clean white/gray look and blue primary buttons.
- The **Upload.vue** page leverages `$t(...)` for all user-facing text, ensuring it’s localized.
- The **ResumeUpload** component encapsulates file input logic and directly calls the backend via the Axios API helper. It provides a smooth UX with status text and error handling.
- The Pinia store (`useResumeStore`) could have actions like `setResume(result)` to store the resume data and perhaps an action to fetch recommendations using the API. By storing the resume JSON (especially the `json_file` name), the Recommend page can retrieve the recommendations. Alternatively, we showed how one might fetch immediately and navigate.
- Navigation is handled by router-link (which is easier than using useRouter in script for this case). After uploading, the user can click the **“View Matching Jobs”** button to go to the recommendations page.

A similar pattern would be followed for the Recommend and Report pages: the Recommend page on mount would call `getRecommendations(resumeFile)` (unless already in store) and populate a list of JobCard components; the Report page would use route params or store to call `getMatchReport` and then render the results (possibly using `v-html` to display the returned HTML report content, after sanitization if needed).

Conclusion: This front-end design provides a clear separation of concerns: pages manage layout and overall flow, components handle specific UI pieces, and an organized store + API layer connects to the backend. The use of Vue3, Pinia, and Tailwind ensures the solution is modern, maintainable, and stylistically consistent with the requirements. All text and data flows are adaptable to Chinese/English audiences, and the architecture can be extended (for example, adding a recruiter view or additional features) by following the same patterns. By adhering closely to the backend API definitions (as documented) and using a proxy for development, the integration with the FastAPI + LangChain backend will be straightforward and testable.

---

[1] [7] Axios + Vue.js 3 + Pinia, a “comfy” configuration you can consider for an API REST | by Israel Díaz Zapata | Medium
https://medium.com/@bugintheconsole/axios-vue-js-3-pinia-a-comfy-configuration-you-can-consider-for-an-api-rest-a6005c356dcd

[2] [3] [4] [5] [8] api_documentation.md
file://file-18MqKDM8xaKqpnzNo5sfpU

6 Scope and Locale Changing | Vue I18n
https://vue-i18n.intlify.dev/guide/essentials/scope