Darwin AI's "Employee Finder" Technical Documentation

Table of Contents

Introduction

Darwin AI *Employee Finder* is a lead generation application that guides potential clients through a multi-step form to select and schedule a consultation with an AI employee (virtual worker) from Darwin AI. The application supports multiple languages (English, Spanish, Portuguese) and offers various AI workers specialized in different roles.

The user flow consists of 5 key steps:

1. Welcome - Collecting website and business email.
2. Form - Gathering company information (name, industry, size).
3. Worker - Selecting an AI worker best suited for their needs.
4. Schedule - Booking a demonstration meeting.
5. Contact - Providing contact details for follow-up.

The application features a responsive design for Desktop veiwports only, animated transitions between steps, and comprehensive form validation. This documentation outlines the tech stack, component structure, and data flow of the application.

Installation and Setup

Prerequisites

- Node.js (v16.0.0 or later)
- npm (v8.0.0 or later) or yarn (v1.22.0 or later)

Installation

1. Clone the repository

   ```
   git clone [repository-url]
   cd employee-finder
   ```

2. Install dependencies

   ```
   npm install
   # or
   yarn install
   ```

Running the Application

- For development:

  ```
  npm run dev
  # or
  yarn dev
  ```

- For production build:

  ```
  npm run build
  # or
  yarn build
  ```

- To preview the production build:

  ```
  npm run preview
  # or
  yarn preview
  ```

Tech Stack

Core Technologies

- **React**: v18.3.1
- **TypeScript**: v5.5.3
- **Vite**: v5.4.2
- **Tailwind CSS**: v3.4.1

Animation Libraries

- **Framer Motion**: v11.0.8 - Used for page transitions and animations

Internationalization

- **i18next**: v24.2.3
- **i18next-browser-languagedetector**: v8.0.4
- **react-i18next**: v15.4.1

Project Structure

The project follows a modular structure with path aliases configured in `tsconfig.json`:

```
src/
  assets/          # Images, icons, logos
  components/      # UI components organized by purpose
    common/      # Shared components
    layout/      # Layout components (Header, Footer, Background)
    steps/       # Step-specific components for the multi-step form
  constants/       # Constant values and configuration
  contexts/        # React context providers
  i18n/            # Internationalization setup and translations
  types/           # TypeScript type definitions
  utils/           # Utility functions
```

Key Components

Step Components

1. **WelcomeStep**: Initial step collecting website and email. Props:

   ```
   interface WelcomeStepProps {
     onSubmit: (data: Form) => void;
   }
   ```

2. **FormStep**: Collects company information. Props:

   ```
   interface FormPageProps {
     onSubmit: (data: { companyName: string; industry: string;
     companySize: string }) => void;
   }
   ```

3. **AIWorkerStep**: Allows selection of AI worker. Props:

   ```
   interface AIWorkerStepProps {
     onSubmit: (data: { worker: string; industry: string }) => void;
   }
   ```

   3.1. Helper Component: **ChatMediaComponents** - Provides visual elements for the chat interface including:

   - **Audio**: Audio message representation. Props:

     ```
     interface AudioProps {
        className?: string;
        isMan?: boolean;
     }
     ```

   - **ImageMessage**: Image display in chat bubbles. Props:

```
interface ImageMessageProps {
   src: string;
   alt?: string;
   className?: string;
}
```

- **Pdf**: PDF document representation. Props:

```
interface PdfProps {
   description: string;
   className?: string;
   worker?: string;
   industry?: string;
}
```

- **ContactCard**: Contact information cards. Props:

```
interface ContactCardProps {
   name: string;
   time: string;
   imageUrl?: string;
   className?: string;
   type?: 'sent' | 'received';
   isMan?: boolean;
}
```

4. **ScheduleMeetingStep**: Schedules meeting time. Props:

```
interface ScheduleMeetingStepProps {
  onSubmit: (data: { meetingTime: string }) => void;
}
```

5. **ContactStep**: Final step collecting contact information. Props:

```
interface ContactStepProps {
  onSubmit: (data: { contactMethod: 'whatsapp' | 'phone',
  phoneNumber: string }) => void;
}
```

Layout Components

1. **Header**: Navigation header with optional back button. Props:

```
interface HeaderProps {
  onBack: () => void;
  showBackButton: boolean;
}
```

2. **Footer**: Contains logo and other footer elements. Props:

```tsx
interface FooterProps {
  showLogo: boolean;
}
```

3. **Background**: Wrapper component for page background. Props:

```tsx
interface BackgroundProps {
  children: ReactNode;
}
```

Common Components

1. **TransitionContainer**: Wrapper for animated transitions. Props:

```tsx
interface TransitionContainerProps {
  variants: Variants;
  className?: string;
  exit?: string;
  children: ReactNode;
  delay?: number;
}
```

2. **PurpleButtonWithArrow**: Primary action button with arrow. Props:

```tsx
interface PurpleButtonWithArrowProps {
  label: string;
  onClick: (e: React.MouseEvent) => void;
  disabled?: boolean;
}
```

3. **IconPurpleButton**: Button with icon for selection components. Props:

```tsx
interface IconPurpleButtonProps {
  icon: string;
  label: string;
  isActive: boolean;
  onClick: () => void;
}
```

4. **BackButton**: Navigation button with arrow icon. Props:

```tsx
interface BackButtonProps {
  onBack?: () => void;
}
```

5. **Dropdown**: Generic dropdown selector component. Props:

```tsx
interface DropdownProps {
  options: DropdownOptions;
  value: string;
  onChange: (value: string) => void;
```

```
  label: string;
  getIcon?: (value: string) => string;
  error?: string;
}
```

6. **PurpleDropdown**: Styled dropdown with purple theme. Props:

```
interface PurpleDropdownProps {
  options: PurpleDropdownOption[];
  value: string;
  label: string;
  icon?: string;
  onChange: (value: string) => void;
  buttonClassName?: string;
  optionClassName?: string;
  iconSize?: string;
  dropdownWidth?: string;
  isClickToOpen?: boolean;
}
```

7. **IndustrySelector**: Industry selection dropdown. Props:

```
interface IndustrySelectorProps {
  handleIndustryChange: (value: string) => void;
}
```

8. **LanguageSelector**: Language selection dropdown. No props, uses `LanguageContext` internally.

9. **WorkerSelector**: AI worker selection dropdown. Props:

```
interface WorkerSelectorProps {
  handleWorkerChange: (value: string) => void;
}
```

Data Management

Context Providers

1. **StepContext**: Manages the current step in the multi-step form.

```
interface StepContextType {
  step: string;
  setStep: (step: Step) => void;
}
```

2. **FormContext**: Manages form data across all steps.

```
interface FormContextType {
  form: Form;
  setForm: (form: Form) => void;
```

```
  errors: FormErrors;
  setErrors: (errors: FormErrors) => void;
  handleFieldChange: (field: keyof Form, value: string,
  step: string, error?: string) => void;
}
```

3. **LanguageContext**: Manages the application's language selection.

```
interface LanguageContextType {
  language: Language;
  setLanguage: (lang: Language) => void;
}
```

Type Definitions

1. **Step**: Enum defining the steps in the application flow.

```
enum Step {
  WELCOME = 'welcome',
  FORM = 'form',
  WORKER = 'worker',
  SCHEDULE = 'schedule',
  CONTACT = 'contact'
}
```

2. **Form**: Type for the form data collected throughout the application.

```
type Form = {
  website: string;
  email: string;
  companyName: string;
  industry: string;
  companySize: string;
  worker: string;
  meetingTime: string;
  phoneNumber: string;
  contactMethod: string;
}
```

3. **ChatMessage**: Used for chat message representation in the application, supporting both text and React components as content.

```
type ChatMessage = {
   sender: string;
   content: string | React.ReactNode;
   time: string;
   type: 'sent' | 'received';
   isContactCard?: boolean;
}
```

4. **Industry**: Defines industry options with display label and associated icon.

```
type Industry = {
   name: string;
   label: string;
   icon: string;
}
```

5. **Industries**

```
enum Industries {
   AUTOMOTIVE = 'automotive',
   EDUCATION = 'education',
   REAL_ESTATE = 'realEstate',
   RETAIL = 'retail',
   HEALTHCARE = 'healthcare',
   INSURANCE = 'insurance',
   SERVICES = 'services'
}
```

6. **CompanySize**

```
enum CompanySize {
   "1_TO_10" = "1-10",
   "11_TO_50" = "11-50",
   "51_TO_200" = "51-200",
   "201_TO_500" = "201-500",
   "MORE_THAN_500" = ">500"
}
```

7. **Worker**: Represents AI worker profiles with various media assets and capabilities.

```
type Worker = {
   key: string;
   name: string;
   roleKey: string;
   imageChat: string;
   imageCard: string;
   imageSchedule: string;
   videoLandscape: string;
   gender: string;
   quote: string;
   abilities: string[];
   integrations: string[];
   metrics: WorkerMetric[];
}
```

8. **WorkerMetric**: Defines metrics/statistics for AI workers with descriptive text and icon.

```
type WorkerMetric = {
   value: string;
   description: string;
   icon: string;
}
```

9. **WorkerRoleKey**

```
enum WorkerRoleKey {
   LEADS_REACTIVATION = "leadsReactivation",
   LEADS_QUALIFICATION = "leadsQualification",
   COLLECTIONS = "collections",
   POST_SALES = "postSales",
   CUSTOMER_EXPERIENCE = "customerExperience"
}
```

10. **Language**

```
enum Language {
    EN = 'en',
    ES = 'es',
    PT = 'pt'
}
```

11. **FormErrors**: The `FormErrors` type mirrors the structure of the `Form` type but contains string values that indicate validation errors for each corresponding field.

Utility Functions

chatTranslations.ts

- `getTranslatedChatContent`: Manages dynamic content translation in chat bubbles, handling different React element types including text, fragments, and anchor tags.

formValidations.ts

- `validateWebsite`: Validates website URL format
- `validateBusinessEmail`: Checks if email is from a business domain
- `validateEmailFormat`: Validates general email format
- `validateCompanyName`: Ensures company name contains valid characters
- `validateIndustry`: Validates industry selection against available options
- `validateCompanySize`: Validates company size selection against available options

iconManipulation.ts

- `getIconFilter`: Generates CSS filter values for icons based on input state (focused, error, or default)

Application Flow

1. The app follows a multi-step form process using the `StepContext` to manage navigation.
2. Each step collects specific information that is stored in the `FormContext`.
3. After completing all steps, form data is collected but not submitted to any specific vendor.
4. As noted in the *TODO* comment in `App.tsx` (line 46), the final submission handling needs to be implemented based on client requirements. The client can connect to any vendor they want to submit the data to.

Internationalization

The application uses `i18next` for translation support, with language detection capabilities for multi-language support.

Responsiveness Testing

The application has been thoroughly tested across multiple viewport sizes to ensure optimal display and functionality on various devices. Testing was conducted on the following common screen resolutions as reported by StatCounter Global Stats [1]:

- 1920×1080
- 1536×864
- 1366×768
- 1280×720
- 1440×900
- 1600×900
- 1280×800
- 1280×1024
- 1024×768

With an addition of extra-large viewports: 2100x1200 and 2560x1440.

The application was tested in the following browsers to ensure cross-browser compatibility:

- Google Chrome
- Mozilla Firefox
- Microsoft Edge
- Safari

Accessibility

The application has been developed with accessibility as a priority, following WCAG 2.1 guidelines to ensure inclusivity for all users:

Keyboard Navigation

- All interactive elements (buttons, form fields, dropdowns) are fully accessible via keyboard navigation.
- Proper focus states are implemented for all interactive elements.
- Tab order follows a logical sequence through the form steps.

Screen Readers

- All images include descriptive alt text attributes for screen reader compatibility.
- ARIA labels are provided for interactive elements without visible text.
- Semantic HTML5 elements are used throughout to provide meaningful structure.
- Form fields include appropriate labels and error messages are announced to screen readers.

License

This project is proprietary software developed as a work for hire.

Ownership and Rights:

- Full ownership of this codebase has been transferred to the client upon payment.
- The client has exclusive rights to use, modify, and distribute this software.
- The client may choose to make the project public or keep it private.
- No part of this codebase may be reproduced or distributed without the client's permission.

This is not an open-source project. Any use, reproduction, or distribution of this code without proper authorization from the client is strictly prohibited.

Credits

This project was developed by NiksonDev.

Contact: contact.nikson.dev@gmail.com