ChiWorks Technical Report

Project Specs: https://www.cs.utexas.edu/users/downing/cs373/projects/IDB.html
Website URL: https://www.chiworks.me/
API: https://api.chiworks.me/
Gitlab Repo: https://gitlab.com/cs-373-group-2/chi-works

Background & Purpose

ChiWorks was created to help the unemployed population within South Side Chicago. The unemployment problem within this community, which is predominantly African-American, has deep and complex historical roots which began back in the mid-20th century during the "White Flight" era. This was when White residents began leaving South Side Chicago which led to a large African-American population, relatively. Deindustrialization during the latter half of the 20th century also led to many factories in the region to shut down and factory workers to lose their jobs. These problems, along with a lack of quality education and systemic racism, have led to a deep-rooted issue of unemployment within this underserved community. Additionally, these issues have led to a lack of resources to find and gain employment within the area. Our website is aimed to aid this community with the necessary resources to not only find a job but also gain skills to land a job.

Models and Instances

The website has three models, all pertaining to helping unemployed people find jobs:

- **Employers** - Employers and companies in and around South Side Chicago that have posted job openings
    - Each employer has many job openings and many resources that it might consider valuable
- **Jobs** - Job postings in and around South Side Chicago
    - Jobs have associated employers, and associated resources that can be useful in getting hired
- **Resources** - Career centers and courses that people can utilize to gain the necessary skills to land jobs
    - Resources are associated with several jobs that they can help with and several employers that might consider them valuable

The website currently has three instances per model. This information has been non-programmatically scraped through single API calls and in future phases, programmatic scraping will be implemented.

Toolchain

Tools and versions used for this project can be found in the package.json files. The project itself runs using Node.JS as its runtime environment. The tools and packages we have used for Phase I include:

- **Axios (1.5.0)** - Used to make HTTP requests from Node.js. In our case, this is used to make GitLab API calls for the About page.
- **TailwindCSS (3.3.3)** - CSS framework which allows for rapid styling of components. Current CSS framework that we are using for the project.
- **shadcn/ui (0.4.0)** - Component library for TailwindCSS. Similar to Bootstrap, this adds UI components that can be populated with custom data (e.g. search bars, dropdown menus)
- **React (18.2.0)** - Open source front-end Javascript library for building UIs. The main framework for the website and project.
- **React Router (6.16.0)** - Allows us to link between pages within the application. Ex: Splash page to the about page, employer instance to job instance, etc.
- **Vite (4.4.9)** - Builds and allows for live reloading of the website during development. Used for easier development and bug catching before deploying to the live site.
- **Flask (4.4.9)** - Builds and allows for live reloading of the website during development. Used for easier development and bug catching before deploying to the live site.
- **Flask-marshmallow (4.4.9)** - Provides validation and request parsing capabilities for our
- **Flask-SQLAlchemy (4.4.9)** - A tool to connect to our MySQL database instance and translate python code into SQL query commands.
- **Flask-Cors (4.4.9)** - Allows our front-end app to communicate and utilize our back-end api and database.
- **Openai(4.4.9)** - Allows us to generate a list of general skills required for each job position

API

Being a website that provides information about job postings, most of the APIs that we have used are from job posting sites such as Glassdoor and LinkedIn.

Employer API:

- Coresignal Company API: https://coresignal.com/solutions/company-data-api/

Job API:

- Job API: https://publicapis.io/jooble-api

Resource APIs:

- Udemy Affiliate API: https://www.udemy.com/developers/affiliate/

We have also created documentation for our REST API which will be used to connect the frontend to the backend. The frontend will call this API to pull instance information from the backend. The API will have six endpoints, two for each model. For each model, there will be an endpoint to retrieve all of the instances of a model (with query filters) and an endpoint to retrieve individual instances. For the endpoints that retrieve all of the instances of a model, you will be able to filter the instances based on their attributes. For example, you can filter all of the job instances based on a minimum annual salary.

The ChiWorks API has been thoroughly documented with Postman and can be found here: https://documenter.getpostman.com/view/29722655/2s9YJXakN2

Searching

Searching is mainly done on the front-end. It is Google-like in the sense that the most relevant results will appear first and the least relevant last. Additionally, the relevant information is highlighted to show the user exactly why each result showed up from the search. There are four search options. There is a main search that searches the full Chiworks database, so you will find employers, jobs, and resources. You can also search model-specific if you are only looking for a specific employer, for example.

Sorting & Filtering

Sorting and filtering are mostly done in the back-end within our API routes. Rather than always returning a full list of all instances within a specific model, users can send in various parameters to apply filters and change what attribute the data is sorted by. If no filters are applied, the api will return all instances within the model. Our implementation allows multiple filters to be applied to the same query and only returns queries that satisfy all of the filters. Additionally, multiple filters can be added to one specific category. For example, you can filter jobs with two different companies. This is done by treating query filters as lists, which we then use to query the database. Each model can be sorted and filtered by any of the attributes listed in their schema as listed in the models.py file.

Sorting is done by setting the sortBy parameter variable and must match an attribute of the respective model. The keywords you can sort by are predefined and depend on the model. The order can be flipped to descending by setting isDesc to true.

GitLab API

The About page of our website is dynamic and makes use of Gitlab's API. Using an access token and the Axios HTTP library, we query metrics on our GitLab project such as commits and issues. Using the response objects, we hydrate the cards in our About page with live data from GitLab on each visit to the page.

ChiWorks API Architecture

The ChiWorks API routes are defined within the "routes.py" file in the /back-end/api directory. We have a function for each route including employer(), employers(), job(), jobs(), resource(), and resources(). The api routes that return all instances are used for the model pages. These routes will only return the necessary information for the instance cards which means no media information. The api routes that return single instances will return all information since they will be used for the instance pages. These routes also return all necessary information for the instances that they are related to.

Frontend Architecture

Our repository is structured into two branches: *main* and *develop:*

- ***main****:* the production branch of the project, which is set up to continuously deploy to our live URL at https://chiworks.me. This branch should be merged whenever pushes need to be made to the production website.
- ***develop****:* the development branch of the project, which is where most work should be done up until production deployments. Changes pushed to this branch will be posted to AWS amplify on a staging URL: https://develop.d3es0i992hangy.amplifyapp.com, allowing for testing across devices.

The core file structure for our react app is as follows. Anything labeled in orange is a configuration file that typically doesn't need to be changed, anything labeled in green is a development file/folder that can be changed:

- **/back-end/ :** Contains the server-side code for the project, including deployment setup

- **/.ebextensions** : sets package configurations for eb environment
- **/api/routes**: defines API routes to be used to access our database
- **/virt/***: our virtual environment used to isolate our dependencies
- **.ebignore** : Mainly used to prevent virt from being included when the project is deployed to the eb environment
- **app.py** : Entry point for our app for local development and testing
- **init.py** : Initializes Flask app and connections to our used plugins
- **requirements.txt** : A list of dependencies required to run the project. Used by elastic beanstalk when deploying the environment
- **Dockerfile-frontend** : File for setting up containerized runtime of application via Docker for back-end

- **/front-end/** : Contains the front-end code for the project that runs on the client side
  - **/components/ui** : Installation directory for shadcn/ui components.
  - **/data/** : Contains JSON files for populating model and instance pages, will be replaced when backend API is implemented
  - **/public/** : Contains assets that will be publicly exposed on the website, (e.g. favicon)
  - **/src/** : Contains React source code for project
    - **/assets/** : Contains assets that will be exposed only if imported by components
    - **/components/** : Contains reusable JSX components, specifically those which are reused *within* pages (e.g. cards)
    - **/lib/** : Contains utils needed by shadcn/ui library
    - **/routes/** : Contains individual pages' JSX files (e.g. about, models)
    - **error-page.jsx** : Page rendered an on site error (e.g. 404)
    - **index.css** : Root stylesheet, with default styles for the website
    - **main.jsx** : Entrypoint for react app, handles all page routing as well
  - **.dockerignore** : Contains list of files to be ignored by Docker daemon
  - **.eslint.cjs** : Configuration file for eslint, the linting tool provided by React
  - **components.json** : Configuration file for shadcn/ui
  - **Dockerfile-frontend** : File for setting up containerized runtime of application via Docker for front-end
  - **index.html** : HTML entrypoint for application, sets favicon
  - **package-lock.json** : Ensures dependencies are correctly added across platforms
  - **package.json** : Contains list of all Node.JS packages needed for site
  - **postcss.config.js** : Configuration file for Tailwind's prerequisite plugin library
  - **tailwind.config.js** : Configuration file for Tailwind
  - **tsconfig.json** : Configuration file for shadcn/ui that allows for import aliases

    - **vite.config.js :** Configuration file for vite
- **.gitignore :** Contains list of files to be ignored by git when pushing to repository
- **.gitlab-ci.yml :** Contains CI/CD pipeline via gitlab
- **Makefile :** Contains useful development, testing, and production commands
- **README.md :** Contains useful information about program, how to run, etc.

On the frontend side, reusable components should be created in the front-end/src/components directory. New pages should be created in the front-end/src/routes/ directory. The routes for new pages should be added via react-router, in front-end/src/main.jsx.

Backend Architecture

Our back-end is all organized within the “back-end” directory. Within this root directory, we have our main backend files including app.py (the application), init.py (the initialization), and tests.py (unit tests). There is then two subdirectories: “api” and “database”. The “api” directory only has one file called “routes.py” which defines the behavior of the api routes. The “database” directory has scraping and database files. For scraping, there is “fetch.py” which defines the methods for fetching api information, “populate.py” which calls the fetch methods and populates the database, and “sklill_finder.py” which calls the OpenAI API to populate skills for each instance of jobs and resources. For the database, there is “models.py” which defines the models in the database and what information goes in the models.

Hosting

Site hosting is done through Amazon’s AWS Amplify service. Front-end hosting environments include a “develop” environment and a main environment. The “develop” environment is linked to a default Amplify URL (here). This is where we will push and test new features and bug fixes to our site. The main environment (www.chiworks.me) acts as our production environment, where final changes are deployed only after they have been thoroughly tested in our “develop” environment. A backend environment named ‘staging’ is deployed however unused and unrequired in phase 1 of the project.

For each change that is committed and pushed via git, Amplify automatically attempts to build and deploy the project based on which branch was committed. Results and logs of the deployment attempt can be seen online on the AWS Amplify Console page.

For the back-end, our Flask app is hosted on AWS’ Elastic Beanstalk service. It creates its own virtual environment with the dependencies defined in the ‘requirements.txt’ file. The

environment runs on python 3.11. Elastic Beanstalk pulls the most recent commit from the project repository and uses that as the source code for the virtual environment. The environment can be accessed via our subdomain "api.chiworks.me." This acts as the root route for our API.

Namecheap acts as our domain provider, but we host and manage our domain on AWS Route 53's DNS. Within Route 53, all of the records associated with our name are listed there, as well as the records required to connect the api subdomain to our backend environment instance.

Makefile

The makefile contains various ways to streamline certain tasks and commands. Most of the makefile targets are self-explanatory in nature and have comments associated with each one, but here are some notable targets to mention:

- ***make install:*** Installs project dependencies. Ensure that Node.js (https://nodejs.org/en/download/current) is installed before using this command.
- ***make dockerstart:*** Build the project and create a docker container for both front-end and back-end. Creates a local preview server to see how the project currently compiles on a browser. Exposes port 5173 by default.
- ***make dockerstop:*** Stops the preview and removes containers after they are no longer in use.
- ***make versions:*** Print the current version of all tools used in the project.

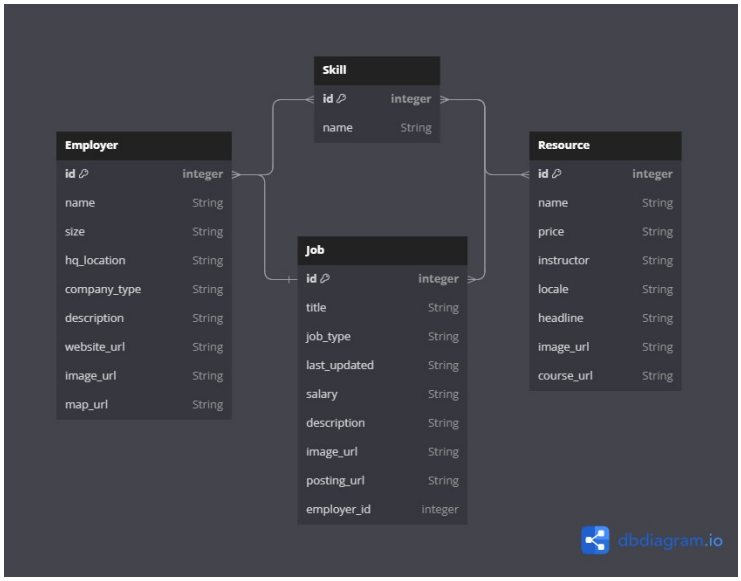
CI/CD

Continuous integration is used to ensure that our website compiles after any push to the repository. Any failed CI jobs should be addressed immediately as this indicates an error in our code base. Continuous integration is split into 3 main stages: build, test, and deploy.

- ***Build:*** Runs the build command for the project and checks if all dependencies can be properly installed.
- ***Test:*** The testing stage is split into two jobs, unit tests and integration tests. Unit tests will test individual functions and items while integration tests will test the overall functionality of the website.
- ***Deploy:*** The final stage of CI. This publishes our project to our "develop" branch on AWS Amplify. Pushes to the main production branch will be done manually to ensure features still being tested are not uploaded.

Database

Our database was created using MySQL and is hosted through Amazon RDS. We have been visualizing our database using SQL Workbench. The ChiWorks database has four models: Employer, Job, Resource, and Skill. The Skill model is what connects resources to the other two models. For example, if an employer and resource have the same skill, then a connection will be made between the two. Between employers and jobs, employer skills are populated by pooling all skills that the employer's jobs have. There is a one-to-many relationship from employers to jobs and a many-to-one the other way. There is a many-to-many relationship from employers to skills and the other way around. There is then a many-to-many relationship between skills and resources. There is a many-to-many relationship between jobs and resources.



API Scraping

When scraping data, we took the "Company" field from the job API, which was "Jooble". This company was then queried in the employer API, which was "Coresignal". If the company is in coresignal, then both the job and employer are added to the database. For resources, we simply scraped from the "Udemy" API and then made all of the connections with skills.

To add the skills to each model, we first started by creating a list of general skills that would be both found in job postings and taught in courses. We then utilized OpenAI's API to pick 5 skills from that list for each resource and job.

Pagination

For pagination, we implemented a “Pagination” component to split up the modelView data into pages of 21 instances. The pages are navigable using the first, previous, next, and last buttons as well as the page indexes. These page indexes are shown five at a time, with the current page in the middle, the previous two on the left, and the next two on the right. In practice, the user can go up to +/- 2 pages and can also jump to the first or last page.

User Stories

Through Phase I, we received 5 user stories with requests and things they would like to see on our website:

Request	Status	Implementation or Reasoning
“I want to know what jobs each company is looking for.”	Solved	Each company instance is linked to all job instances that they are looking for. These instances will be shown within the individual instance card.
“I want to know the salary of the job.”	Solved	The annual salary is an attribute of all job instances.
“How do I know where a career center is?”	Solved	For career center instances specifically, a map embed has been added on the instance page to show where the building is.
“I want to be able to search for a specific employer. ”	Solved	Searching has been implemented. You can search both through a total search page and also a model-specific employer search page. Searching is Google-like and results are sorted by relevancy.
“I want to know how to gain the skills that a job or company is looking for.”	Solved	Both job and employer instances are connected to resource instances which are career centers and online courses. These links are created by matching the skills the resources teach with the skills that the jobs and employers seek.
“We want to see the API running”	Solved	You can access the API from api.chiworks.me. Full documentation of the api found in the link at the top of this document.
“Make sure to add Postman API tests”	Solved	We have added Postman API tests to our CI pipeline.

“We want to see two instances of media for each model”	Solved	For the employer, you have a map and image. For job, you have an image and embedded link. For resources, you have a course link and image.
“Make sure you put testing in your CI pipeline”	Solved	We have added frontend and backend unit tests as well as Postman tests for the API.
“Add unit tests for your frontend”	Solved	We have unit tests for all components of our frontend.
“Make sure to have a database diagram”	Solved	We have added a database diagram which thoroughly shows all of the connections and tables. You can find this in our repo at the base directory. It is titled “ChiWorks_DB_Diagram.png”
“We want to have a thorough tech report”	Solved	This technical report thoroughly outlines all parts of the project and ensures future engineers can pick up right where we left off. This tech report will continue to be updated as more changes are made.
“We want to have API documentation updated”	Solved	API Documentation has been updated to correctly describe and define all api routes and their parameters with the implementation of sorting and filtering.
“Please implement filtering so it is easier to find what I am looking for”	Solved	Filtering has been implemented by updating our backend API routes. You can now add parameters to your api queries which will filter the results that return.
“Please implement sorting so that I can easily find what I need”	Solved	Sorting has been implemented by updating our backend API routes. We now have two new parameters in each route “sortBy” and “isDesc”. Our frontend then utilizes these parameters to implement sorting on the website.

Challenges

Throughout the development process, we ran into a couple of challenges:

- **Hosting**: Hosting was probably the most difficult part of the phase. Kaiden was able to take charge in that realm and get hosting up and running. After many hours of videos and tutorials, we were finally able to get ChiWorks hosted using AWS Amplify. Backend and database hosting was also a struggle that Kaiden was able to figure out after determining the cause of the issue was our dependencies.
- **Design/layout**: Figuring out how we want the website to look was another challenge. We knew that we wanted a sleek and simplistic look, and Tailwind was able to give us the necessary tools to do so. We also wanted to have an identity for the website, so we went for a bean. This, of course, ties into the Chicago bean but also symbolizes a seed and how through this site, users will grow in their careers.
- **Finding an idea / getting approval**: Finding an idea was a huge challenge due to the constraining nature of the prompt. We actually scrapped our initial proposal and went with this one due to it being much more specific to a community. In the end, we are happy with the outcome of our project.
- **Time constraints**: The time constraint of the project was definitely challenging. With so much to do, we were concerned with fulfilling all the parts of the rubric. We were able to finish on time by delegating tasks based on strengths and communicating very efficiently. For example, Kaiden was mainly focused on hosting, Caiden was focused on API documentation, Akram was focused on the GitLab API, and Sarvesh on UI.
- **Data scraping:** Finding the best APIs for data and determining how we were going to scrape them was definitely a challenge, but we were able to design a well thought out plan and ensured our plan was full-proof before beginning.
- **Type Changing:** The database has some information that is in the incorrect form. For example, salaries are strings when we need them to be ints. This is especially challenging for filtering and sorting. A lot of ChatGPT was needed to figure this out.