# Contents

# 1 Go Resume Design Document

## 1.1 Design Document Contributors

- Jorge Henriquez (contact@jorgehenriquez.dev)

## 1.2 Introduction & Motivation

If you're in the job market you need to have a couple resumes, plural. This is because if you scour the internet for job seeking advice, you'll find that one of the many recommendations is to tailor your resume to each job you apply to. That's hard.

Having to locate your `.docx` (or `.tex` if you're fancy), edit it (which can be annoying if you're using a word processor), and convert it to a `.pdf`, *is a pain.* All I should I have to do is just build my base resume once, then edit a small file to produce a resume. This is where ***Go Resume*** steps in.

## 1.3 Overall Goal

The good folks over github.com/saadq/resumake.io have made an amazing resume system. So amazing that a `pandoc-loving-can't-write-latex-to-save-my-life` nerd can use it and make amazing resumes! However, I just want to be able to use it once to build the initial system, then use a `CLI+yaml` to edit my resumes just how I want it.

The good news is that once we create the resume on their website we can download a `resume.json` file that holds our entire resume! Even better, the website also has an API endpoint that'll produce our `resume.pdf` from our `resume.json`! If we use a configuration file and some good old-fashioned programming, we can create an easy resume system!

While they do have a live website it would be rude to bombard their servers. So we'll just add their repo as a sub-module in our repo! Then we can just build their code and we'll be off to the races.

Also, I want this to be cross-platform so we'll try to avoid any *NIX or Windows specific tools, with the obvious exception, of this `Makefile` generated design document. Whoops…

## 1.4 Design

To motivate the design let's examine the use case of the program.

1. Build the initial resume using the website generator.
2. Download the `resume.json`
3. Configure a yaml file that has the desired-job attributes (i.e. skills).
4. Combine the yaml file with the `resume.json` to yield a `customized-resume.json` file.
5. Send the customized file to the api endpoint.
6. Receive and write the `resume.pdf` to the file system.
7. Submit the application.
8. Find a new job posting
9. Goto 3

The first thing that stands out is having to run a script that start up the `resumake.io` services. Since a goal of this project is to be cross platform we can't use a `.*sh` file to execute the program. But we can use a programming language (we'll use Go for this) to create a CLI (`resume-start`) to automate this for us.

### 1.4.1 Launching a local instance of resumake.io

This task is pretty trivial. In our programming language, we just spawn a process that executes the instructions as specified by `resumake's` `contributing.md`.

To build `resumake.io` we need to run `npm run build` and `npm start`. Building takes a while and only needs to be done once. We'll provide an option to the CLI to skip building the frontend (`-skip-build`) with its default value set to false. To also allow for more developer flexibility we'll also provide a path to the resumake.io directory with the `-resumake-dir` flag. This is needed so that the tool knows where to execute the build instructions.

Additionally the process should be able to handle interrupts gracefully when running the server and client. It should return a success (typically 0) exit code when running them. If an interrupt is received during the build process it should return a failure exit code (typically 1).

Since the GUI is only potentially used once it makes little sense to run it every time the user needs to tailor a resume. Therefore, the `-no-client` flag should exist. This option will not run or build the client. It follows that it will also adhere to the `-skip-build` flag and have a graceful shutdown properties.

The following command (`resume-start`) will encompass the above design.

```
Usage of resume-start:
  -log string
        the file where resume-start logs to (default "./.resume-start.log")
  -no-client
        disable running the client
  -resumake-dir string
        the directory where resumake.io resides (default "./resumake.io")
  -skip-build
        skip building resumake.io dependencies
```

To build `resume-start` it should be as easy as running the following cross-platform shell command:

```
go build ./cmd/resume-start
```

### 1.4.2 Tailoring a Resume

We'll want to a build a binary called `resume-tailor` that tailors a resume. It should be built similar to resume start: `go build ./cmd/resume-tailor`.

To start, let's examine lst. 3 which is a generated sample `resume.json`. In our case we'll want to tailor the `skills` and `selectedTemplate` keys of the object.

Since Go is a strongly typed language we'll have to create a data structure from this definition. This is easily achievable with the help of json-to-go. A notable result of the generation is lst. 1, the generated skills Go data structure.

If we create a file that looks like lst. 2 we could parse it into the Go Skills structure. This file is easy to use and easy to edit, which makes it a perfect for this project. This is because it'll allow us to combine `skills.yaml` with the `resume.json` file. With a simple line (`parseJSONResume.Skills = parsedYAMLSkills`) we can edit the resume, re-marshal it into JSON and send it off to resumake.io server. The server would then yield a customized and tailored `resume.pdf`.

That being said we should have a couple of options for the user to configure.

#### 1.4.2.1 Options

**1.4.2.1.1 Output** The result should output to the operating system's standard output. This will allow the user to call the file whatever they want or pipe it into another program.

**1.4.2.1.2 Required Inputs** As for inputs (`resume.json` and `skills.yaml`) they should be optional position arguments. These will come after any flags we define later and will have the default names of `resume.json` and `skills.yaml`. This will allow the user to use the binary like so:

```
# Method 1, assumes (`resume.json` and `skills.yaml` exists)
resume-tailor > ouput.pdf

# Method 2 uses the provided file names to generate the resume.
resume-tailor otherResume.json otherSkills.yaml > ouput.pdf
```

**1.4.2.1.3 Flags** An obvious flag is to allow the user to select a template. I quite happen to like template number 6, so we'll make that the default one. It can specified with the `-template` flag.

Another obvious flag is to allow the user to configure the API endpoint of resumake.io server. Since this is made to be used in conjunction with `resume-start` we'll use it's default. Its default is `http://localhost:3001/api/generate/resume`. We'll provide this flag just in the case doesn't want to install `Latex` or `Node.js` on their system. It could be changed to `https://resumake.io/api/generate/resume` to use the already running server. This flag will be specified by `-URL`

Another flag is the `-force-single` which forces the generated pdf to be a single page. You will commonly find advice that states that a resume should be no longer than a page, thus we'll provide this option to the user. By default, it will be turned off.

Lastly we'll allow the user to output the customized JSON with a `-JSON` flag. This will not output the pdf data.

Calling `resume-tailor -help` should result in the following:

```
Usage: resume-tailor [OPTIONS] [resume.json skills.yaml] > resume.{pdf,json}

[OPTIONS]
  -JSON
        output JSON instead of PDF data
  -URL string
        the API endpoint of resumake.io (default "http://localhost:3001/api/generate/resume")
  -force-single
        force the resulting resume to be a single page
  -template int
        which template to use [values: 1-9] (default 6)
```

## 1.5 General Usage

To use this system this user will install all the required dependencies (though, in practice Go is the only required dependency since `resume-tailor` has a `-URL` flag). The following steps then ensue:

1. The user builds `resume-start` with `go build ./cmd/resume-start`.
2. The user builds and starts resumake.io with `resume-start`
3. The user builds their base resume on the GUI and downloads their `resume.json`.
4. The user can optionally stop `resume-start`.
5. If `resume-start` was killed, it can now be re-ran with `resume-start -skip-build -no-client`.
6. The user builds their `skills.yaml` based on a job description.
7. The user builds `resume-tailor` with `go build ./cmd/resume-tailor`
8. The user then runs `resume-tailor > resume.pdf` with their options to generate their tailored resumed.

On subsequent runs the user only has to do steps 5,6, and 8. And now the user can easily generate a tailored resume for any job they encounter.

This concludes the design document.

## 2 Appendix

**Listing 1** Definition for skills in Go

```go
// Skills represents a skill that you might have.
// Keywords are usually a subset of that skill.
type Skills struct {
    Level    string   `json:"level,omitempty" yaml:"level,omitempty"`
    Keywords []string `json:"keywords" yaml:"keywords"`
    Name     string   `json:"name" yaml:"name"`
}
```

**Listing 2** Example yaml file

```yaml
- name: Programming Languages
  keywords:
    - Go
    - Python
    - Java
    - C
    - C++
- name: Spoken Languages
  keywords:
    - English (Native)
    - Spanish (Native)
    - French (Conversational)
```

**Listing 3** Sample `resume.json`

```json
{
  "selectedTemplate": 1,
  "headings": {
    "work": "Work Experience",
    "education": "Education",
    "projects": "Projects",
    "awards": "",
    "skills": "Skills"
  },
  "basics": {
    "name": "Jorge Henriquez",
    "email": "contact@jorgehenriquez.dev",
    "phone": "(661) 243-7834",
    "location": {
      "address": "Bakersfield, CA"
    },
    "website": "https://jorgehenriquez.dev"
  },
  "education": [
    {
      "institution": "University of California, Santa Cruz",
      "location": "Santa Cruz, CA",
      "area": "Computer Engineering with Honors",
      "studyType": "BS",
      "startDate": "September 2016",
      "endDate": "December 2020",
      "gpa": "3.35"
    }
  ],
  "work": [],
  "skills": [
    {
      "level": "",
      "keywords": ["C/C++", "Go", "Verilog", "Chisel3"],
      "name": "Programming Languages"
    },
    {
      "keywords": ["React", "Babel"],
      "name": "Frameworks and Tools"
    }
  ],
  "projects": [],
  "awards": [
    {
      "title": "",
      "date": "",
      "awarder": "",
      "summary": ""
    }
  ],
  "sections": [
    "templates",
    "profile",
    "education",
    "work",
    "skills",
    "projects",
    "awards"
  ]
}
```