Students

- Abbas Yadollahi - 260680343
- Alexander Bratyshkin - 260684228
- Alexander Harris - 260688155
- Andrei Ungur - 260690364
- Bogdan Dumitru - 260690446
- Camilo Garcia La Rotta - 260657037
- David Ritch - 260685199
- Elias Al Homsi - 260797449
- Erick Zhao - 260687719
- Filip Bernevec - 260689062
- Laurent Chenet - 260614863
- Rami Djema - 260668934
- Sebastian Andrade - 260513637

Motivation and Reason

With the increasing complexity of the job searching process, it has become extremely difficult to keep track of one's job applications. Indeed, it is not rarely that one ends up receiving a call from a company to which they don't even remember applying to. This is largely due to how distributed the recruitment process has become: a job searcher has to find out about openings on different portals (Glassdoor, Linkedin, ...) and then subsequently has to fill out an application in different companies' career sites (Facebook's, Amazon's, ...). Naturally, in such circumstances, a person can become overwhelmed and can quite easily lose track of where one has applied, especially when aggressively applying to a plethora of openings (as it is often the case for students looking for internships or for people in desperate need of a job). Our product's aim is to facilitate the job hunt process by providing an seamless platform which allows users to efficiently manage their applications. Users can, in a click of our Chrome extension, easily add the application to which they've just applied (from Google's career site, Indeed.com, Linkedin, ...) into their own personal backlog of applications, without having to, for example, manually fill out entries inside of an Excel spreadsheet. This backlog, or history of applications, can quickly be accessed from our web portal, in which users can view relevant information related to a particular application in a very expressive way (company, position, title, deadlines, ...) and/or perform grooming tasks such as, for example, update its status (applied, rejected, offer, ...). Additionally, we provide functionalities that allow users to externalize memories instead of worrying about remembering them; an example of this would be our "interview questions" feature in which users can easily associate interview questions that they actually had to a particular application. We believe that through the organizational power that our neat interface provides, users are going to be able to gain meaningful insights into the progress of their search for employment and manage it with much more confidence. Jobhub's ultimate goal is to help its users remain organized and methodical throughout their job hunt in hopes of lightening the anxiety of what is already an extremely burdensome process.

In prediction of the fact that the uniqueness of our tool would attract a large initial user base, we also decided to add recruiters into the platform. The motivation for such an expansion of our scope is the following: given that we amass a large number of users primarily due to the functionalities described in the previous

paragraph, we also want to provide recruiters with an opportunity to reach out to such a large pool of candidates, and, therefore, grow the number of visitors into our website even further. Hence, recruiters are also able to add job postings inside of our platform, and users are able to apply to these with a few clicks. This can also be seen as an attempt to tackle the problem of decentralization of the recruitment process by unifying both, applications and applications, under one shared system.

Project Management

Project Preparation

The project preparation document, created before development began, allowed the team to turn their idea and vision for the project into a concrete delivery plan. This included creating a product backlog to implement features, a testing plan and a done checklist in order to assure that the product delivered was up to coding and business standards, and organizing the sprints in order for the team to benefit from Agile development.

Done Checklist

The done checklist was used to ensure that the product delivered was up to the standards set by the team and those commonly seen within the coding community. It is the contract that binds the team members to producing high quality code that can be maintained and sustained through the development process. As such, the team incorporated all the items that were established in the done checklist. For example, the team chose to adopt the GitFlow standard, a branch management standard that separates code used for development purposes and code delivered to production, as opposed to a trunc based approach where small feature branches would be merged into the master branch. This choice was made to be able to have stable and taggable releases, thus ensuring high quality code and having a clear separation between development, staging, and production environments. Furthermore, to make sure that the entirety of each subteam had an understanding of elements of the code other members were working on and ensure that code quality was respected, pull requests needed to be peer-reviewed by at least two members of the subteam. However, in smaller subteams, this review was done by the entirety of the subteam. Additionally, during the second sprint, seeing that the team was technically talented in all aspects of the project (frontend, backend, authentication), developers that were put in other subteams would also evaluate the pull requests of their previous team, showing also the collaboration between teams.

Product Backlog

The product backlog provided the team with a complete picture of how to put their vision for the product into deliverable features. In creating the product backlog, the team agreed that developing the high priority stories would be sufficient to obtain the "minimum viable product" that could be released, whereas the medium priority stories' implementation would assure that we delivered a complete product. The low priority stories were considered as "nice-to-have" and "quality of life" features. This gave the team a good idea as to what should be included in each of the two sprints, but also give enough flexibility to add more low priority stories in the eventuality that development went better than expected.

Sprint Backlog and Task List

The sprint backlog allowed the team to plan their work for a short period of time, thus allowing them to focus only on the tasks they chose and believed were the most important to include at the time. It focuses on

developing features over writing documentation, and as such only features that have value to the hypothetical customer of the product were included. As such, no documentation tasks were added to the backlog, opting to write it after the development of the product finished. However, documentation was needed for the authentication microservice seeing all teams needed to have an understanding as to how it could be used and implemented. Even so, documentation for this microservice was added after the first sprint when all the endpoints were already fully developed. Additionally, the task list allowed the team and, more specifically, the scrum master, to track the progress of how each story evolved throughout its development. An advantage of have such a tasklist with specific development stages for each story ("backlog", "in progress", "done", "blocked", "not implemented") is that the progress can be tracked and priorities may be shifted to stories that need more attention. It also allows other developers to see the pain points of the sprint and, as caring members of the project, lend a hand to fixing said issues. As such, the continuously evolving nature of the sprint backlog allowed the team to deliver the product that the hypothetical user wants rather than following a set plan, because requirements tend to change very fast in today's world.

Scrum Rituals

Backlog Grooming

At the start of the first sprint, we brainstormed a list of all features we intended to implement. This formed the basis of our product backlog. Each item in the product backlog was represented by a user story however the product backlog also contained all features, functions and requirements. After all requirements and features were defined, we grouped the items into high, medium and low priority user stories. Once the user stories were defined, we broke up the user stories into tasks, addressed the acceptance criteria for each and estimated the effort of each. Because our main priority was to deliver working software frequently, we made sure to pay extra attention to higher priority stories when grooming the backlog.

By the second sprint our backlog grooming process did not evolve much. Besides minor adjustments to some tasks, we followed the same process of looking over each user story and converting them to tasks and their estimated efforts.

Sprint Planning

Our team carried out sprint planning at the beginning of each sprint and during our weekly meetings where we refined our stories (backlog grooming). The scrum master facilitated the meeting with the entire team present where we discussed the stories and associated tasks involved with each feature. After the feature was discussed with the entire team, each member defined their work and effort necessary to complete their assigned task.

From sprint 1 to sprint 2 there was no change in terms of planning. However, we had to adjust the roles of our team members due to one of our peers dropping out of the class at the beginning of the first sprint and only informing us late in the first sprint. Because of his absence we had one less developer on his microservice and had to adjust accordingly. Because he did not inform us of him dropping the course, we operated on the assumption that he was completing his tasks. Additionally, because of his unresponsiveness we only learned late into the first sprint that he had in fact not completed any of his assigned tasks. This lead to an evolution in how we planned the effort of our tasks for sprint 2. For example, we had to be more conservative with story estimation because of having one less team member. Because our objective during each sprint was to maintain a constant pace we had to respond to this change in our team dynamic and deviate from our original plan. We had to adjust the ownership of the microservice he had been assigned to in order to make sure the

tasks had been completed. Given that we had a large pool of talent within our team, the entire microservice that was left without a member was completed by one developer.

Story Estimation

Instead of estimating a story based on points as is done in industry, we estimated the effort involved in fully implementing a story based on the effort involved for each task of the story. This allowed us to have a slightly simpler model for evaluating the importance of a story.

This process did not evolve from the first to second sprint. Firstly, we wanted to keep the same method of story estimation to maintain continuity between the two sprints. Secondly, we did not feel the need to change how we estimated a story's effort because we did not run into any problems in the first sprint and we were able to mark all our tasks as completed.

Daily Standup Meetings

Daily scrums were done informally within subteams, as a daily standup comprised of all fourteen members of the team would be infeasible and counter-productive considering the nature of the project. It would have required a lot of effort to organize the meeting around every member's schedule. Seeing the project required each team member to work on the product for a minimum of three hours a week, a daily scrum meeting does not provide enough content to justify its existence.

As such, to use the time allocated to the project more efficiently, daily scrums were left at the discretion of each subteam, which opted to communicate daily over the corresponding Discord text channel. This also allowed other teams to read what the team had been doing if the matters concerned them. The subteams also organized in person meetings in order to do pair-programming and discuss implementation and design choices. This was more suitable than discussing on Discord, as the subteams could use whiteboards to draw mockup solutions and have richer debates using voice instead of text.

However, to make sure the team was in-sync with each-other and the progress of the product, we held weekly scrum meetings. We opted to hold these meetings remotely over a Discord voice channel seeing the large majority of the team resides far from campus. A few challenges arose using this method. The size of the team made choosing a time where each student was available very complicated. Additionally, there were also exceptional situations where team members simply could not assist the meeting due to personal reasons, and thus needed to still have access to the new information that arose during the weekly scrum. The solution we chose was to hold the weekly scrum on Mondays right before class. This is a time where every member of the team was available and would start the new week of the sprint early enough for subteams to incorporate the information discussed in the meeting. As the concept of a scrum meeting entails, the weekly meeting served to put all the subteams on the same level with the project as a whole and to assess how their work is developing. Three pieces of information were thus needed: what was done in the previous week, what was going to be done in the next week, and identifying major internal or external blockers. Also, the scrum master would also make any new announcements he received from the teacher assistant. To ensure that even the absent members were kept informed of the discussions had in the scrum meeting, minutes were taken and posted to a Google Drive folders and hyperlinked in the appropriate Discord text channel.

With such a robust scrum meeting system put in place early in development, there was not much change in the format of the meeting. However, albeit usually short meetings were held, weeks where larger deliverables were due required longer meetings, and as such the scrum master needed to warn the rest of the team to be available longer than usual.

Sprint Demo

At the end of each of the sprints, we had a release branch which signified it as demo-ready. After the second sprint ended we demoed the full application and its functionalities to the TA. During our weekly meetings, the sub-teams demoed their progress in order to ensure continuously delivering a release-ready project. This obviously was not as important during the first few weeks of the sprint but towards the end was crucial in our meetings.

From the first to second sprint there was no evolution in how we preformed sprint demos.

Sprint Retrospective

Sprint retrospective meetings were held once after every sprint and were incorporated in the weekly scrum meeting. For the first sprint retrospective, our team discussed what went well and what could be improved upon. Because we only had 2 sprints, we only had the opportunity to implement the suggestions from one sprint retrospective. However, in order to ensure the team was as efficient as possible we regularly discussed any blockers and progress during our weekly meetings at great length.

We did not change our approach to the sprint retro from the first to the second sprint. While discussing what could be worked on, we made it a point to ensure that there was continuous attention to technical details as well as following good design practices.

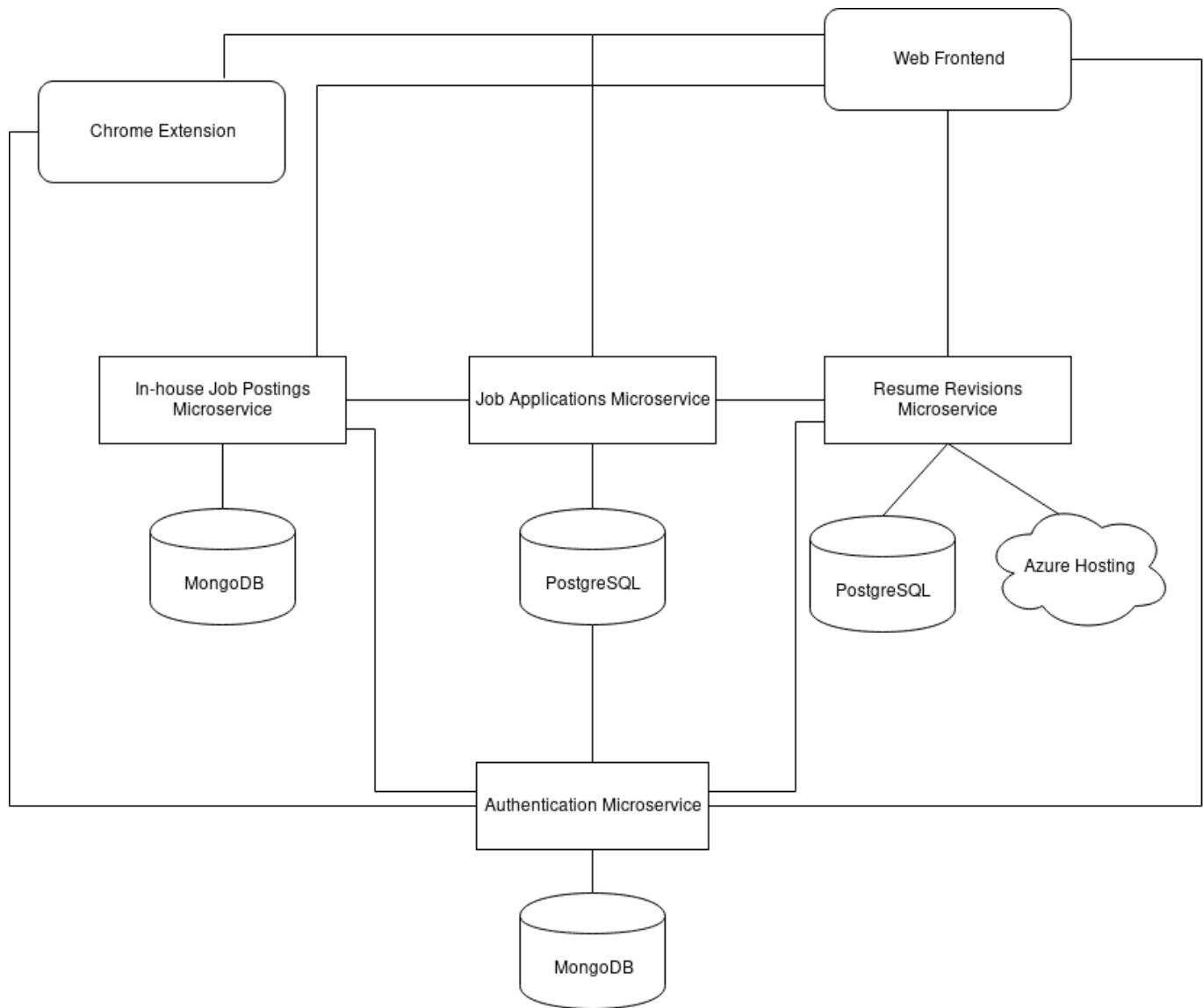## Collaboration Between Team Members

Given that we were a team of 14 members it was not always easy to coordinate between everyone. Because of the scarce facilities McGill offered for meetings of this size and everyone's incompatible schedules, we opted to form sub teams. In order to facilitate collaboration between members we split the application into microservices. Each microservice was assigned to a sub team which made face-to-face collaboration between those members easier. Through the weekly meetings, each respective sub team could then update the whole team on their progress. Additionally, because of the smaller teams working on subset of tasks, we utilized pair programming which lead to less errors and more robust code. Moreover, we felt that breaking up into self-organizing teams would lead to the greatest efficiency, architecture and design. Maintaining smaller teams facilitated collaboration which in turn greatly increased the simplicity of dependence between members.

We did not consider another method of collaboration as we felt this was most effective. As mentioned before, the large team size was a blocker for collaboration so splitting into subteams was the optimal choice for us. From sprint 1 to sprint 2 we did not evolve this strategy. However, because some microservices were finished earlier we did rotate members into other sub-teams to balance the workload.

Architecture

On a high level, we opted for a microservice architecture. There are two logistical reasons for this.

First, it allowed us to utilize the diverse expertise of our team's members. Nobody was restricted to a particular technology, and everyone could use their preferred stack and programming language. This helped us speed up the development process quite significantly, since it spared some of our members the time required to learn an entire new ecosystem that they're not familiar with.

The second reason is distribution of tasks. By keeping ourselves localized to smaller teams corresponding to each microservice, we avoided the overhead of everyone having to deal with the potentially annoying technicalities that are proper to large teams working on the same repository and/or codebase.

As for more technical concerns, we chose microservices because it's an architecture which significantly improves fault isolation (since failures are localized to specific modules and not to the whole application as a whole), because it gave the flexibility (for those who wanted it) to experiment with new technologies without having to worry about dependency concerns or roll back changes, and finally because reasoning about microservices it's extremely easy, meaning that the functionality of our service was very easy to understand.

We also considered a service-oriented architecture, but quickly realized that this would imply setting up a standardized communications protocol over a network, such as for example an enterprise service bus. However, this would've been too extravagant for the purposes of our project without really offering any significant benefits. A microservice architecture is significantly easier to implement and simpler to reason about.

Given our microservice architecture, it was most natural for us to split each microservice/module into its own repository, as opposed to having everything in a monolith repo. First and foremost, this would prevent us from dealing with the clutter that's engendered by having everything in one repo, e.g. the exponential number of commits and issues, most unrelated to each other; we thought that having separate repos would simply be cleaner and would allow us to remain better organized. Secondly, multiple repos establish a clearer separation of concerns: why would the frontend team have to worry about very specific technical changes inside of one of the microservices, for instance? Additionally, having multiple repos lessened our risk of having to deal with undesired conflicts and merges, which naturally sped up the development process. Finally, setting up CI/CD and dependency management when all modules are isolated in their own repos is easier because we can simply use preset configurations without having to waste time on dealing with more intricate operational issues, such as for example manually configuring package managers or CD config files, or having to be constrained by very specific folder structures. Of course, the other advantage is that sub-teams are given the freedom to pick their own preferred CI/CD options, without being limited by a particular choice with which they might feel less comfortable.

The two most popular choices for an application which interfaces with multiple microservices/systems are REST and GraphQL. We opted for the former for all of our microservices simply because it is a well-established technology. A lot of our members are unfamiliar with GraphQL, and we didn't want them to waste time on climbing the learning curve in exchange of no real major benefit for the modest purposes of our project.

The following subsections detail the architectures, design decisions and technology choices of the particular microservices themselves.

Authentication

Description

😎 Provides authentication/user management for all jobhub microservices. Uses JWT for authentication.

Each user has the following required attributes:

- `id`: A unique ID generated for each user.
- `email`: An email address used for login.
- `password`: The users password. All passwords are hashed using bcrypt.
- `type`: The type of user. Can be Applicant or Recruiter.
- `verified`: Whether the user has verified their email after creating their account. Required to be able to login.

Getting Started

Setup

```
git clone https://github.com/scrum-gang/authentication.git
cd authentication
npm install
npm start
```

Typical usage

1. Create user using `/signup`.
2. Verify new user by clicking link in email received.
3. Login using `/login`, keep JWT token.
4. Can get logged in user using `/users/self` and passing token in header.

Running Tests

```
npm test
```

Deployment

Builds are automated using Travis and deployed on Heroku.

There are two Heroku deployments:

- Staging: https://jobhub-authentication-staging.herokuapp.com/
- Production: https://jobhub-authentication.herokuapp.com/

The staging deployment should be used for all development/testing purposes, in order to keep production from being polluted with test data.

Please note that any new builds on the **development** or **master** branches will **wipe** the staging database.

Folder Structure

```
.
├── doc # Contains endpoint documentation
├── favicon.ico
├── LICENSE
├── package.json
├── README.md
├── src
│   ├── auth.js # Password hashing implementation
│   ├── config.js # Contains various constants and environment variables
│   ├── index.html # Default route landing page
│   ├── index.js # Main application
│   ├── models # Contains mongoose models
│   │   ├── InvalidToken.js # Database schema for invalid JWT tokens
│   │   └── User.js # Database schema for user object
│   └── routes # Contains endpoints
│       └── users.js # Contains all authentication endpoints
└── test # Contains all tests for the application
    └── functionality
        └── endpoints.js # Tests for all endpoints
```

Database Info

Details all the fields in the User model.

Field	Type	Required	Allowed Values	Specified By
email	String	true	n/a	user
password	String	true	n/a	user
type	String	true	Applicant, Recruiter, Moderator	user
name	String	false	n/a	user
address	String	false	n/a	user
github	String	false	n/a	user
linkedin	String	false	n/a	user
stackoverflow	String	false	n/a	user
id	String	n/a	n/a	system
verified	Boolean	n/a	n/a	system
created_at	Date	n/a	n/a	system
updated_at	Date	n/a	n/a	system

All `/users` endpoints except for `/users/self` are restricted to moderators only. Moderators have unrestricted access to all endpoints. Only a moderator can promote another user to a moderator role.

Note: Restrictions on endpoints can be bypassed by passing the `secret` header in the request. Ask someone on authentication for the secret or see pinned message on authentication channel on Discord.

Technologies

- Node: Main language used.
- Restify: For creating REST endpoints.
- mLab + MongoDB: For storing user data.
- JSON Web Token: Authentication scheme.
- bcryptjs: Password hashing.
- nodemailer and googleapis: For sending verification emails to new users.
- mongoose: MongoDB modeling for User object.
- mocha, chai: Unit and endpoint testing.
- Istanbul and coveralls: Test coverage and reporting.
- Travis CI: CI/CD
- Heroku: Cloud hosting
- mongodb-memory-server: DB mocking.
- eslint: Linter
- Visual Studio Code: Main IDE used.

Design Decisions

For creating our REST endpoints, we decided to use Restify over a more well known framework like Express due to the fact that it is more lightweight and minimalistic, which made it easier to quickly get our service up and running, since most of the other microservices depended on ours.

For the database we elected to use a cloud-hosted service called mLab, which provides MongoDB databases. Our main motivations behind this decision were the NoSQL-type database, and the cloud hosting which made the deployment process much easier.

When it comes to the actual authentication scheme, we decided to use JSON Web Token (JWT) over something like OAuth due to the fact that it is much simpler to implement and members of our team did not have much experience integrating OAuth into a web application.

In addition to the technologies previously discussed, the following were also used to provide various features:

- bcryptjs: Password hashing.
- nodemailer and googleapis: For sending verification emails to new users.
- mongoose: MongoDB modeling for User object.
- mocha, chai: Unit and endpoint testing.
- Istanbul and coveralls: Test coverage and reporting.

For CI/CD we used Travis CI to automatically build our app on new commits, run tests and coverage, and then deploy to our cloud platform. This is the industry standard CI/CD tool for open-source projects and has very good integration with GitHub. It being cloud-hosted also means we did not need to worry about managing our own CI/CD server like if we had used something like Jenkins. To host our app, we used Heroku since it provides free hosting and integrates well with Travis, and avoids us having to manage our own web server.

We tried to be as thorough as possible when designing tests for our microservice considering existing time constraints. Every endpoint has at least one success flow, with critical endpoints also having an additional error flow. Given more time we would like to add more test cases for each endpoint, at a minimum having at least one success and error flow for each.

We have two configurations for the test suite. One runs the tests on a mocked database created in memory at runtime using mongodb-memory-server. This runs on every Travis build. The other runs only when deploying a new build to Heroku, and uses a staging version of our database on mLab, which lets us verify that data is being replicated properly on the remote database.

We also run our linter before the test cases using eslint, which ensures we keep a consistent code style throughout the codebase.

Task Distribution

Sebastian Andrade: Database schema, JWT implementation, endpoint implementation.

Bogdan Dumitru: Database schema, JWT implementation, endpoint implementation.

David Ritch: IP timeout/blocking feature, input validation for endpoints.

Alexander Harris: Unit/endpoint testing, endpoint implementation, extended User schema.

Laurent Chenet: Endpoint implementation, input validation for endpoints.

Limitations

As it stands, our authentication supports an IP blocking feature for when a certain IP sends more than 20 requests in 1 minute or fails 10 login attempts in 5 minutes. At this time, the amount of time that the IP address times out is hard coded to be 5 minutes, which works in terms of functionality, though an improvement that could be made would be to implement it so that the length of the timeout varies depending on the number of times the IP has been timed out. This is something that would be worth doing to improve the security of the system in the future.

Additionally, we believe that it would be useful to include a feature where a moderator can see which users are currently logged in and which are not. This was originally in our low priority stories, though due to the time constraint of the course, we decided that it wasn't worth implementing in the scope of this project. However, since all other requirements related to the main functionality of our authentication are currently completed, this would be the next item to implement.

Job Applications

Description

This microservice allows authenticated users to track applications to jobs. The microservice handles applications to inhouse postings and external postings differently to ensure we don't create redundancies in the stored data, but most other functionality is generic to any type of application.

Getting Started

Setup

Note: For making contributions, an ideal development environment is Visual Studio Code as it is lightweight and provides syntax highlighting. PyCharm is also a good option, as it is a industry-standard Python development environment.

To run the microservice, you need to...

- ... create a virtual python environment using the following command: `python -m venv venv`
- ... activate the environment: `source venv/bin/activate`
- ... install the dependencies: `pip install -r requirements.txt`
- ... run the API : `python app.py`

Once inside the virtual environment, make sure to have PostgreSQL installed on your computer.

Create a database named `jobapplications` and grant any of your psql users permissions to it.

To do this:

- `sudo -u postgres psql` to connect to the database;
- `CREATE USER potato WITH PASSWORD 'potato_pw';` to create a user;
- `grant all privileges on database jobapplications to potato;` to grant the user privileges;

Next, you'll want to set your environment variables to contain the database log-in credentials:

- export PSQL_USER="potato"
- export PSQL_PW="potato_pw"

Finally, you can create the tables in your database using the setup.py script.

- python setup.py

You now have an instance of the database with the correct tables and columns.

Running Tests

`build passing`

`codecov 73%`

Once you have completed the setup, you can run the tests from the root of the repository via pytest:

```
pytest tests/
```

If you would like to view coverage reports as well, you can run the following command to leverage pytest-cov:

```
pytest --cov --cov-config=.coveragerc --cov-report term tests/
```

We perform thorough unit testing on the controllers of the application. Each function within the controllers has a test for a successful flow, as well as at least one failure flow. It is difficult to test for every possible error, however we still achieve a code coverage of 73%. Note that code coverage here is defined by line coverage. We also have a suite of tests where we execute controller functions directly on the database, to ensure that the database itself is functional and allows us to have the desired controller functions. These tests have Test Driven Development in mind as they were designed before implementing the controllers themselves. In order to run the tests, it is necessary to first setup the microservice as explained in the setup section in the repository readme. Once everything is setup, to run the tests, simply run pytest tests/ from the root of the project.

Folder Structure

```
.
├── alembic # folder with all the alembic configuration files
├── alembic.ini # alembic initialization file
├── applications.py # methods related to applications
├── app.py # The starting point of the flask application
├── docs  # folder with all the docs
│   ├── databasedesign.md # Explanation of the database schemas
│   └── endpoints.md # Explanation of the endpoints
├── external.py # Methods related to external applications
├── internal.py # Methods related to internal applications
├── interview.py # Methods related to the interview questions
├── LICENSE # Project's license
```

```
├── Procfile  # Heroku configuration file
├── README.md # REpository readme file
├── requirements.txt # All the projects requirements
├── setup.py # Script to initialize the database
├── tables.py # Script defining database schemas
├── tests # Folder with all the tests
│   ├── conftest.py # Setup test
│   ├── test_db_creation.py # Test for the database setup
│   ├── test_external_applications.py # Test external applications
│   └── test_internal_applications.py # Test internal applications
└── utils.py # Script with authentication methods
```

Database Info

Below is a description of the tables for the Job Applications database.

Applications

Parent table with generic information about all types of applications. The identifier from this table is used in every API endpoint that operates on specific applications (i.e.: deleting applications or updating information about them).

- `id` : integer, primary key, unique, mandatory
- `date` : date, mandatory
- `user_id`: string, mandatory, comes from auth microservice
- `is_inhouse_posting` : Boolean, true if it's an inhouse posting false if not
- `status` : String, tracks the status of a given application. Can be modified by user for external postings but not
- `resume`: String, handy tool for applying to jobs
- `comment`: String, Optional comment a user might add to his application.
- `resume`: String

Inhouse

For inhouse postings, we only need to store the job identifier which maps to information tracked by the inhouse postings microservice.

- `id` : Integer, primary key
- `application_id` : Integer, foreign key from `Applications`
- `job_id` : String, mandatory, comes from inhouse postings microservice

External

For external postings, we need to track more information on the application as the posting does not come from one of our microservices.

- `id`: Integer, primary key
- `application_id` : Integer, foreign key from `Applications`

- `url`: String, mandatory
- `position`: String
- `company`: String
- `date_posted`: String, but it's just a stringified datetime object
- `deadline`: String, but it's just a stringified datetime object

InterviewQuestion

Users may want to also track what interview questions they received for a given job application. This table enables mapping many interview questions to a given job application, regardless of its type.

- `id`: Integer, primary key
- `application_id`: Integer, foreign key from `Applications`
- `title`: String, title of the interview question
- `question`: String, specific interview question

Technologies

- Python: a programming language that lets you work quickly and integrate systems more effectively.
- Flask: a micro web framework written in Python.
- SQLAlchemy: a Python SQL toolkit and Object Relational Mapper.
- Alembic: very useful tool for allowing consistent and speedy data migrations.
- PostgreSQL: an open source object-relational database management system.
- Heroku: a cloud platform as a service (PaaS) supporting several programming languages.
- TravisCI: a hosted, distributed continuous integration service used to build and test software projects hosted at GitHub.

Design Decisions

We chose to build the microservice using Python given its simplicity: when we follow the PEP8 styling practices, well-written Python can be read like prose. This microservice was built using the Flask framework once again for simplicity. We've opted for a SQL database to enable scalability, as we expect to have too many applications for an in-memory database. We opted to use PostgreSQL because the team was already familiar with it. We interact with this database through an Object Relational Mapper (ORM) called SQLAlchemy provided by Flask, once again for simplicity: it is much easier for other teammates to read code with ORM calls than it is to parse SQL queries within Python code. The choice to include Alembic was with a long-term vision: as we want to make modifications to our database, having migration scripts will be quicker and more efficient.

Although we've considered alternate Python frameworks, all the ones we picked are simply the most reliable due to having reliable or large amounts of contributors, and most commonly used.

The database schema was also designed with simplicity in mind: we knew that this microservice would interact with every other microservice given that every other component of our project is meant to enable users to track job applications. We decided to use inheritance between tables to be able to make modifications that affect all types of applications, while also being able to make modifications which affect only specific types of applications (i.e.: inhouse or external).

The architecture of our repository also had modularity in mind to enable quick modifications: we have one central file with the "View", which is the routes for our endpoints, and separate files for each controller. The view simply takes the JSON body of a user's request and sends it to a controller. This makes testing and localization of faults easier.

The application is deployed through Heroku due to its seamless integration with Github, and ease-of-use. The deployed version can be found here: Heroku app link. More information regarding the many end-points themselves can be found in the microservices' official documentation.

Task Distribution

Originally, there were 2 developers assigned to this team, but one of them dropped out of the class during the first sprint. Consequently, Andrei Ungur, the only remaining developer in the team, was in charge of this repository. By the end of the first sprint, there was not much else to do for this microservice, so it was not necessary to add more people to this team. Andrei was in constant contact with members from the team working on the Chrome extension as well as the web application for help with this microservices' development as well as ensuring consistency between the implementation of our components.

Limitations

The biggest limitations of this microservice stem from our interactions with its database. Namely, the problems are:

- Nothing to handle concurrent `INSERT`, `UPDATE` or `DELETE` statements,
- No indexing on frequently queried attributes in the tables.

Consequently, the system could fail, have inconsistencies in the data it stores or become increasingly slow as we scale up and the amount of transactions with the databases increases.

Moreover, the project is being tested using only unit tests and no integration tests. Most of the integration testing was done manually when connecting this microservice to the others (Chrome Extension and Front-End). We found that this type of test was not very relevant because the application is mostly handled by controllers (which themselves are tested). It would, however, make our microservice more robust to include integration testing.

Regarding scalability, we also have no setup for stress testing our application, which would be crucial given how central this microservice is. In the future, we would also add concurrency testing as well as handling for concurrent `INSERT` operations, which we believed to be out of the scope of this course.

Although we have included Alembic, we did not adjust our data migration scripts very diligently as we developed the microservice, which means that currently it is not fully prepared to handle data migrations.

Miscellaneous

Future Work

Addressing the limitations related to scaling, namely including indexes to frequently queried attributes and testing for potential concurrency issues to ensure our application is safe from many concurrent calls. Adding mutual exclusion around our database operations might be required if concurrency testing reveals potential issues.

For speeding up our microservice, we can also create a second database for "archived" applications, where we'd move very old applications (or those of inactive members) through an administrative endpoint. This would ensure only very important information is kept in the database for active users, to maximize its speed.

Finally, we should update the Alembic data migration scripts.

Resume Revision

Description

📙 Microservice that provides an endpoint to manage and store resumes.

Getting Started

Setup

```
git clone git@github.com:scrum-gang/resume-revision.git
cd resume-revision
bundle install
systemd start postgresql
rails server
```

A typical use case:

1. Create a new resume using POST on `/resumes`.
2. List all users resumes using GET on `/resumes/:user_id`
3. Edit a specific resume meta data using PATCH on `/resumes/:id`
4. Get a resume using GET `/resumes/:user_id/:title/:revision`
5. Delete a resume using DELETE `/resumes/:user_id/:title/:revision`

Builds are automated using Travis and deployed on Heroku.

The deployment is on this url:

- Production: https://resume-revision.herokuapp.com//

The tech framework is using rails for 3 different environments:

- `test` which is used for running test cases
- `dev` which is used for developing the app locally
- `production` which is deployed on heroku and uses azure active storage to store resumes.

Running Tests

In order to run the tests, the code needs to be pushed onto github to trigger the travis file to run the test folder. Also you can run tests locally by running `rails test`

Folder Structure

The folder structure of the resume-revision microservice is as follows:

```
├── app  # It organizes application components. It's got subdirectories
├── controllers # has the resumes_controller.rb
│   │   ├── application_controller.rb
│   │   └── resumes_controller.rb # has all the functionalities for resumes
├── models
│   │   ├── application_record.rb # base class for a component
│   │   └── resume.rb #the resume entity
│   ├── service
│   │   └── authentication_token_verifier.rb # this service calls the auth
endpoint
├── config
│   ├── application.rb #general rails configuration
│   ├── database.yml # database configurations like connection string`
│   ├── environments
│   │   ├── development.rb # configuration for development environment
│   │   ├── production.rb # configuration for production environment
│   │   └── test.rb # configuration for test environment
│   ├── routes.rb # has routes defined on this project
│   └── storage.yml # has active storage configuration and azure connection
├── db
│   ├── migrate # migrations are used to introduce changes in the schema of
db
├── Gemfile # the file that includes packages used for rails
```

- app/controllers: The controllers subdirectory is where Rails looks to find the controller classes including resume controller.
- app/helpers: The helpers subdirectory holds any helper classes used to assist the model, view, and controller classes.
- app/models: The models subdirectory holds the classes that model and wrap the data stored in our application's database. In most frameworks, this part of the application can grow pretty messy, tedious, verbose, and error-prone. Rails makes it dead simple!
- components: This directory holds components, tiny self-contained applications that bundle model, view, and controller.
- config: This directory contains the small amount of configuration code.
- db: this has the database configuration for the rails application
- License: which includes the license of the repository

The other folders are generated by by Rails and they are not very relevant.

Database Info

Details all the fields in the Resume model.

Field	Type	Required	Allowed Values	Specified By
:data	Blob	true	Base64 of the resume pdf	user

Field	Type	Required	Allowed Values	Specified By
`:title`	String	true	the title given to the resume	user
`:revision`	String	true	numeric versioning	user
`:user_id`	String	true	foreign key on the auth service	user
`:user_name`	String	false	n/a	user
`:resume_data`	String	false	The parsed raw :data	user
`id`	String	n/a	id auto generated by Rails	system

Technologies

- **Heroku** here
- **Postgresql** here
- **Active Storage on Azure with Rails** here
- **Httparty** as an http client here
- **Webmock** here
- **Rails** here

Design Decisions

The dev environment uses Ruby on Rails for the app and the backend side and to manage the connections with the database.

Ruby on Rails is a fast framework used to develop apps that are convenient and reliable. On the other hand we needed some technologies to upload resumes and save them permanently which Heroku did not provide under its free plan. This is why we used Azure Storage to save resume files as a blob and attach them to the resume object and their fields. We choose Azure storage due to their convenience regarding the student plan which gives us 300$ free trial to access resources. Also, the team was familiar with it and we did not want to waste time with searching for alternatives.

We have also used webmock to do integration testing of the backend to ensure that the authentication endpoint is being called when passing a resume. We choose webmock because it is one of the well known libraries for simulating calls to other services. It was also a great fit since the team members already had experience with it.

The design of the resume-controller backend service follows a model-route-controller architecture. We chose this model since Rails by default supports MVC architecture and we did not want to change the default behavior as Rails apps works best with MVC. Also the team is familiar with this pattern.

The endpoints are described in the /routes directory and also when using `rails routes` command

Each endpoint is linked to a function in the controller under /controllers inhouse-job-postings.controller.js. The controller functions handle the business logic associated with each endpoint.

The resume-revision service is hosted on Heroku. Using Travis CI, each time code is pushed into our repository on Github, a build will be spawned in Heroku.The build first runs our unit tests and then executes the rails server command. The database chosen for this microservice is PostgreSQL which is the default

database. The rationale behind is that Rails comes with its own database and sets up its own table. Rails apps are known to manage their own database schema for fast prototyping.

Heroku was used as the deployment platform for several reasons. First of all, it integrates easily with other platform used such as Github, Travis CI, and mLab for the database. Moreover, it is free of charge for a project of this scale, and most importantly reliable when looking at down times and deployment speeds.

Task Distribution

The project was separated into two sprints.

In the first sprint, Elias Al Homsi defined worked on making a working prototype that connects to the Azure blob storage to save resumes. Abbas Yadollahi worked on creating the pipeline and the deployment CI/CD with the intention of integration with Heroku.

In the second sprint, Abbas Yadollahi worked on fixing bugs, making sure the code base is readable and appropriate for the other teams in order to link our microservice with ours. Elias Al Homsi worked on integrating the authentication service with the current resume endpoint and make sure it that all requests should authenticate first before accessing resources.

The history of commits is available on Github.

Limitations

The resume-revision micro service requires all requests to follow the exact schema provided in order to attach a good resume pdf file.The current design does not check that the passed binary file is actually a pdf resume, an image or a virus. Therefore, All field must be populated and a good resume must be passed to the resume-revision in pdf format. We also don't check the files uploaded so they could be infected.

Another limitation is that the Heroku deployment is using a free tier plan. This means that the amount of computational resources is limited (512 MB RAM). The same holds for the PostgreSQL database, which only provides the application with a limited amount of storage due to the free tier plan. If this application were to scale up and the amount of concurrent users were to increase, this limited amount of resources would be insufficient.

Each resume has the following required attributes:

- `user_id`: A unique ID which is a foreign key from the authentication service.
- `user_name`: The name of the user that owns the resume (could be null).
- `title`: The title of the resume ex: my tech-resume, design resume.
- `revision`: The version number of the resume. ex: tech-resume v2.
- `resume-data`: The blob data **Rails Active Storage** that has the resume pdf.

The main reason of having a title and a revision is that sometimes users would like to customize their resumes depending on the job to focus on different skill set that might be more relevant to some postings.

Inhouse Postings

Description

The Inhouse Posting module is a back-end microservice providing API endpoints to create job postings. It hosts the functionality and logic allowing recruiters to post job applications on the web app. When recruiters want to display job postings on our website through the user interface, the web app will connect to the inhouse-posting micro service and take care of storing, formatting and retrieving data. This is all achieved using a Node.js architecture following a restful design. The specifics on the technology & architecture are highlighted in the *Technologies & Design Decisions* section.

Getting Started

Setup

In order to setup the microservice, follow the steps below:

```
git clone https://github.com/scrum-gang/inhouse-postings.git
cd inhouse-postings
npm install
npm start
```

Builds are automated using Travis and deployed on Heroku.

There is one Heroku deployment:

- https://inhouse-jobpostings.herokuapp.com/

Running Tests

In order to run the tests, the code needs to be pushed onto github to trigger the travis file to run the test folder.

Folder Structure

```
├── authentication  # folder that contains authentication
│   ├── inhouse-job-posting.auth.js # file that contains authentication
functions
├── controllers  # folder that contains controllers
│   ├── inhouse-job-posting.controller.js # file that handles HTTP request
logic
├── models  # folder that contains models
│   ├── inhouse-job-posting.model.js # file that defines job posting model
├── routes  # folder that contains routes
│   ├── inhouse-job-posting.route.js # file that defines routes
├── test  # folder that contains tests
│   ├── test.js # file that tests endpoints
├── .gitignore # file to ignore some files from pushing to github
├── .travis.yml # file for travis CI
├── app.js
├── aap.json
├── LICENSE
├── package-lock.json
├── package.json
```

```
├──  README.md
```

The folder structure of the inhouse-postings microservice is as follows:

In the root, there are 4 folders; authentication, controllers, models, routes, test. The root also contains file needed for Node.js such as app.js, LICENSE, package.json.

`authentication` folder contains the javascript file that has the functions that perform the logic of getting the JWT token from the header and sending it to the authentication microservice endpoint to check if the user is a recruiter or not.

`controllers` folder contains the javascript file that deals with data processing after reaching the endpoint.

`models` folder contains a javascript file with the MongoDB schema structure for a job posting.

`route` folder contains a javascript file of all the HTTP routes the microservice can offer.

`test` folder contains a javascript file with the test environment and the tests on each endpoint.

Database Info

Each job posting has the following attributes:

- `id` : A unique ID generated for each user.
  Type: `String`,
- `recruiter` : A Global Unique Identifier for the recruiter.
  Type: `String`, Required: `true`
- `title` : The title of the position of the job.
  Type: `String`, Required: `true`
- `description` : The description of the job position.
  Type: `String`, Required: `true`
- `location` : The location of the job.
  Type: `String`, Required: `true`
- `salary` : The yearly salary for the position.
  Type: `Number`, Required: `true`
- `requirements` : All the requirements needed for the job.
  Type: `String`, Required: `true`
- `company` : The company name.
  Type: `String`, Required: `true`
- `start date` : The date the applicant will start working.
  Type: `String`, Required: `true`
- `end date` : The date the applicant will finish working if it is a contract position.
  Type: `String`
- `posting date` : Generated date of the posting.
  Type: `String`, Required: `true`
- `deadline` : The deadline for the applicant to apply.
  Type: `String`, Required: `true`

Technologies

- **npm** is used as our package manager since the code is written in JavaScript. It is the default package manager for the JavaScript runtime environment Node.js, which is at the core of our architecture. npm
- **Node.js** is used to execute our javascript code. It has the advantage of having many tools and framework that significantly facilitate the development of a RESTful backend. nodejs
- **MongoDB** is used as our database. mongodb
- **TravisCI** is used to integrate the builds into github. travisci
- **GitHub** is used as our repository and version control system. You can find our repo here. github
- **Heroku** is used to host the inhouse postings microservice and to integrate the automated tests into our build system. heroku

List of npm (Node.js) packages used:

- **axios** is used to handle authentication. It provides an easy-to-use promise based HTTP client for Node.js.
- **Express** is used to build our APIs. It provides a fast and minimalist way to create endpoints for our Node.js application.
- **Mongoose** is used for our object and schema modelling since we are using MongoDB. It provides simple ways to describe our schema in node.js
- **Chai** and **Mocha** are used to write our unit tests. It provides a BDD/TDD assertion library as well as a framework to run Node.js unit tests that test our api endpoints.

Design Decisions

The design of the inhouse-posting backend service follows a model-route-controller architecture. The reason for choosing this type of architecture is because we needed routes to be seperate from their purpose handled in the controller. This way we can accomodate the frontend with different routes and worry about implementing them in another file.

The endpoints are described in the `/routes` directory in the `inhouse-job-posting.route.js` file.

Each endpoint is linked to a function in the controller under `/controllers/inhouse-job-postings.controller.js`. The controller functions handle the business logic associated with each endpoint.

The model `/models/inhouse-job-postings.model.js` describes the database schema. The controller uses the model to make sure that the incoming requests and outgoing responses follow the schema.

The model-route-controller architecture is used to separate and keep the three modules independent of one another. The advantage of this design is that changing one of the three modules does not impact the others. For example, changing the name of an endpoint from `/getall` to ``/retreiveall` in the routes file is as simple as just changing the route name. This does not impact the rest of the code, since they are independent.

The unit tests are written under the `/test` directory in the `test.js` file. The tests use the `expect()` function for assertion.

The inhouse-postings service is hosted on Heroku. Using Travis CI, each time code is pushed into our repository on Github, a build will be spawned in Heroku. The build first runs our unit tests and then executes the npm run command, which uses Node.js to run our app on Heroku's hosted server.

The microservice was coded using javascript with Node.js for the sole reason that the whole team had experience with the language and the framework. This meant that the team could spend more time implementing high priority features and not worry about understanding the language and the node framework.

The database chosen for this microservice is MongoDB. The rationale behind is that there is no need for a relational database as only job postings are stored. Hence, a document-oriented database such as MongoDB is a perfect fit. MongoDB also is easily deployed using the mLab hosting service.

Heroku was used as the deployment platform for several reasons. First of all, it integrates easily with other platform used such as Github, Travis CI, and mLab for the database. Moreover, it is free of charge for a project of this scale, and most importantly reliable when looking at down times and deployment speeds.

Task Distribution

The project was separated into two sprints.

In the first sprint, Filip Bernevec defined the job posting database schema, as well as implementing it with MongoDB. He setup the continuous integration with Travis CI as well. On the other hand, Rami Djema spent his working hours on creating the backend architecture using Node.js. During this sprint, he created REST API endpoints for creating a job posting, updating it, viewing and deleting.

In the second sprint, Filip Bernevec had to update the schema in order to have the similar entities as the job applications microservice. He also had to configure and deploy the app on Heroku, and implement authentication in the backend. As for Rami, he setup the testing environment using Mocha to test all the endpoints. He also handled crashing issues with the app, and added custom endpoints as requested by the frontend team.

Limitations

The inhouse-postings micro service requires all requests to follow the exact schema provided to the recruiter. All field must be populated. Therefore one limitation is that if some non-conforming information is provided by the recruiter, i.e. a salary range rather than an absolute value for the salary, this information will have to be provided somewhere in the description and the salary must be left null. These small limitations are things that could be fixed as we would get feedback from users of the application in order to make a schema that best fits their needs.

Another limitation is that the Heroku deployment is using a free tier plan. This means that the amount of computational resources is limited (512 MB RAM). The same holds for the MongoDB database, which only provides the application with a limited amount of storage due to the free tier plan. If this application were to scale up and the amount of concurrent users were to increase, this limited amount of resources would be insufficient.

Miscellaneous

Endpoints

The detailed endpoints can be found on in the README of the inhouse-posting repository. In summary, this microservice has endpoints for getting a posting by ID, getting a posting by recruiter, and getting all postings. It allows the recruiter to create a posting, to update by ID, and to delete by ID.

There is a restriction on three endpoints, which require the user to be a recruiter. This authentication feature allows only the recruiter to create, update and delete a posting. In order to accomplish this authentication, those endpoints expect the HTTP request to have a JWT token in the header.

Web Application

Description

The front-end web application communicates with all microservices and serves as the end user's main gateway to the JobHub application. The web application was built to run on any modern web browser (e.g. Firefox, Chrome, Safari, etc.) using the open-source ReactJS view library as the backbone for our project.

Getting Started

Setup

Ensure that you have Node.js installed. Clone the scrum-gang/jobhub-web repository, go into the root directory of the project, and run `npm install` to install all dependencies. To run the project, simply run `npm start`. To maintain code quality, we recommend using the VSCode text editor, and having the TSLint and Prettier extensions installed to have proper code formatting.

Running Tests

We ran unit tests with the Jest test runner and the Enzyme React testing utility. Essentially, the tests consisted of determining if the routes were correctly rendered for each page. We decided to limit the scope of our automated unit testing for the front-end since the core functionality of the APIs was already tested within each microservice. To run the tests, simply run the `npm test` command.

We also had a suite of acceptance tests written in Java and Selenium, which performed the tasks on the live version of the website (see the scrum-gang/jobhub-selenium repository).

Folder Structure

```
public # HTML file that's ultimately rendered
src
├── Features # an individual folder for each feature
├── Shared # shared components across features
├── __mocks__ # mocks for jest tests
├── __tests__ # jest tests
├── api # axios API layer
├── assets # static assets (images)
├── config # shared TypeScript constants and types
├── App.tsx # Main component for the application
├── index.tsx # Uses react-DOM to render the App.tsx component onto the
HTML
├── react-app-env.d.ts # configuration file so React can work with
TypeScript
├── serviceWorker.ts # allow offline running of app
├── setupTests.js # Enzyme testing configuration
```

```
.gitignore # ignores files on git
.travis.yml # build settings for Travis
LICENSE # open-source license
README.md # repository readme
netlify.toml # Netlify deploy settings
package-lock.json # npm lockfile to conserve dependency versions
package.json # package details (name, license, dependencies, etc.)
tsconfig.json # TypeScript transpiling configuration
tslint.json # linter configuration
```

Technologies

- TypeScript: Superset of JavaScript with added Static Types
- ReactJS: view library for JavaScript
- Create React App: starter template for ReactJS
- Material-UI: user interface component library for ReactJS following
- Google's Material Design spec
- MUI-datatables: in-depth tables for Material-UI
- Axios: HTTP client for JavaScript
- Yup: Object schema validation for form validation
- Formik: Form library for ReactJS
- React Router: Client-side routing for ReactJS
- React Toastify: Notifications for ReactJS
- Timeago.js: JavaScript utility for converting a timestamp to plain language (e.g. date X was "2 weeks ago")
- TSLint: Static code linting for TypeScript
- Jest: Test runner for JavaScript
- Enzyme: Testing utility for ReactJS components
- TravisCI: Continuous integration tool
- Netlify: Static website host
- Codacy: Static code analysis
- Codecov: Testing coverage analyzer
- Prettier: Code formatter for VSCode

Design Decisions

We chose React because it was a lightweight UI library that we could extend with different libraries (e.g. react-router for routing, Formik and Yup for form validation, Material-UI for pre-styled visual components, react-toastify for live notifications, etc.)

We bootstrapped the project using create-react-app, a zero-configuration starter project for ReactJS that equips developers with a toolchain allowing for newer JavaScript features (see the ECMAScript standard here) that are not supported across all browser versions. Moreover, we decided to use TypeScript as our development language rather than plain JavaScript. TypeScript is a superset of JavaScript that adds static typing, leading to less error-prone and more scalable code. We have set up create-react-app to transpile the TypeScript down to JavaScript.

Aside from React, we also use the axios HTTP client to interface with all of our microservices. Axios provides many abstractions that are useful for dealing with multiple REST services (individual API instances, configuration options, redirects, etc.) We wrapped each microservice into its own instance of axios and created an API layer that could be queried across the entire application.

To save persistent state, we used a mix of the browser's localStorage API to have persistence across sessions, and ReactJS' Context API to pass state across different components from a single source of truth. This was mostly used to keep the authentication state useable throughout different scenarios (closing your tab, navigating through different pages, etc.) We also considered using a separate global state container library (such as React Redux or Mobx), but we ultimately decided that the scope of our project didn't require that we learn to use another developer tool when a simpler solution could suffice.

The application was deployed using Netlify, a free static site host with many useful features, such as Continuous Deployment, HTTPS, and deployment previews (with each GitHub pull request). Travis CI was also added to have Continuous Integration with our unit tests, as well as Codacy to have continuous integration with static analysis of our code with each pull request.

Task Distribution

Erick Zhao worked on bootstrapping the application, writing the API layer, laying out the UI, handling persistence, developing the Authentication and In-house Job Posting features, and handling unit tests. Suleman Malik worked on a few UI tasks and the Resume Revision integration. Sebastian Andrade worked on the Recruiter dashboard. Alexander Bratyshkin handled a lot of the logic for Job Applications. Laurent Chenet worked on acceptance tests. Bogdan Dumitru worked on editing and deleting profiles. Andrei Ungur worked on integrating the in-house job posting API with the applications API.

Limitations

The web app might run slowly because all of its APIs that it queries are run on Heroku dynos that sleep when inactive. There is no catch-all 404 route for when a user enters an invalid URL. Some of our forms have poor user interface practices
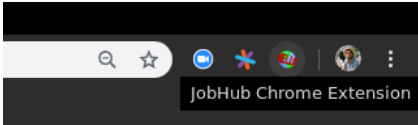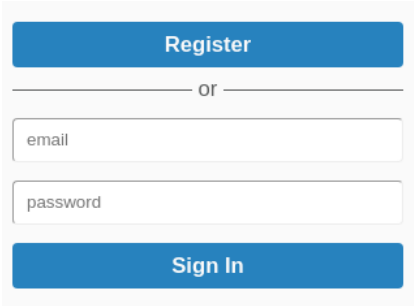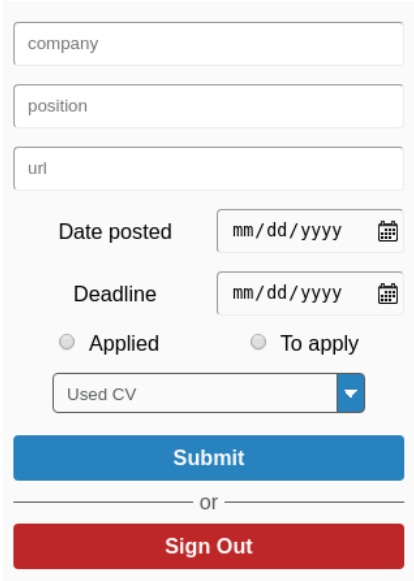
Chrome Extension

Description

The Chrome extension is meant to be an ad-hoc access to the JobHub services. When the users are on a job application website, they don't need to stop browsing the post in order to open the JobHub website on a separate tab. Instead, they can start tracking the given application via the popup submit directly on the job posting website. This component is meant to be as unobtrusive as possible in the user's browsing experience, while nonetheless providing a quick access to the main JobHub use cases.

Any user, both registered or unregistered with the JobHub services, can begin using the extension within a Chrome browser in any website. It suffices to click on the extension's icon on the top left corner of the browser and the login form will popup. The user can register to JobHub by clicking the *register* button, which will redirect him to the Frontend service to complete the registering form. If he already has an account, he can login by entering his credentials. This will log the user and store the JWT token required to perform operations with the JobHub services. Subsequent logins within the same browser session will automatically login the

user. Once the user is authenticated, the Job Posting form will be displayed. If the user opened the extension from within a website containing a job application, the extension will automatically detect the name of the company, the position, the URL, and the date the posting was posted. The user fills the remaining details, such as CV used, and whether or not they already applied or will apply. Note that the CVs must be added from the Frontend service. After the user is done submitting job applications to the JobHu service, he can click anywhere in the browser outside of the extension popup to close the widget.

Getting Started

Extension Logo	Login Popup	Job Posting Form

Setup

All the technologies named here will be further explained in the *Technologies* section.

1. Clone the GitHub repository and install the dependencies found inside `package.json`

```
git clone https://github.com/scrum-gang/jobhub-chrome.git
cd jobhub-chrome
npm install
```

2. Start the Bucklescript watcher to compile ReasonML `./src/*.re` into Javascript `./lib/js/*.js`

```
npm start
```

3. Start the Webpack watcher to bundle `./lib/js/*.js` into `./build/index.js`

```
npm run webpack-dev
```

4. Start the Jest watcher to continuously test your changes

```
npm test
```

5. If you haven't done so previously, unpack the extension locally in your browser.
   Note that you only really need to run this step once.

Testing

As shown in point #4, the NPM test runner will take care of continuously running the test suite everytime the source code is modified. Testing was integrated into the development cycle since the first sprint. The team chose to write tests for features that did not require mocking at the same time that the components were being implemented. To ensure accountability for testing, the issue and the pull request forms had a field specifying how the test coverage of the component under question was modified. TravisCI would also flag any pull requests which reduced the test coverage of the system as a whole.

Folder Structure

```
.
├── .github # contains Code of Conduct & templates for PR/Issues
├── build   # generated by Webpack. Contains the final
js/html/css/resources
├── data    # list of companies to scrape
├── lib     # generated by Bucklescript. ReasonML -> Javascript
├── src     # ReasonReact components & global stylesheet
├── .editorconfig      # linter
├── .travis.yml        # CI
├── bsconfig.json      # Bucklescript config
├── jsconfig.json      # VSCode intellisense for Chrome API
├── manifest.json      # Chrome Extension config
├── package.json       # Node config
├── popup.html         # html of the extension
└── webpack.config.js  # Webpack config
```

Contributor Guidelines

Code of Conduct

We chose to adopt the standard Code Of Conduct specified by https://www.contributor-covenant.org Because the Chrome Extension component of JobHub was a fully open source project, we wanted to render contribution to the project inclusive to all. The code of conduct can be summarized as follows:

> In the interest of fostering an open and welcoming environment, we as contributors and maintainers pledge to making participation in our project and our community a harassment-free experience for everyone, regardless of age, body size, disability, ethnicity, sex characteristics, gender identity and expression, level of experience, education, socio-economic status, nationality, personal appearance, race, religion, or sexual identity and orientation.

Issues and Bug Forms

So that any member of the ECSE428 team could request features or flag issues with the system, we created an Issue template. It contains the minimum information required for the Chrome Extension team to debug the issue or to plan the requested feature.

Form for bugs

- Expected Behavior
- Actual Behavior
- Steps to Reproduce the Problem

Form for enhancement

- Reason for the enhancement
- Explanation of how the enhancement may implemented
- Impact on existent system

Pull Request Forms

Because teams were shuffled after the first sprint, Pull Requests could potentially be opened by any member of the team. To homogenize the requests and facilitate the code review process a template was provided to be filled when opening a Pull Request. The minimum information the Pull Request must have is:

- an issue attached to it so that the team can discuss before a solution is implemented
- a set of tests that validate the bug is fixed or the enhancement is functional

We were keen on developing a safe system, hence we were open and explicit in the GitHub page of the project to request any security bug that would be found to be reported directly to the team.

Form for Pull Requests

- Description of changes made
- Checklist
    - Code compiles correctly
    - All tests passing
    - There is an open issue linked to this PR
    - Created tests which fail without the change (if possible)
    - Extended the README / documentation (if necessary)

Technologies

- ReasonML: syntax extension for the OCaml language that resembles JavaScript. Used as main development language for the Chrome Extension.
- Bucklescript: it is a backend for the OCaml compiler which emits JavaScript. Used to generate JavaScript files from the ReasonML source code.
- JavaScript: it is a programming language that is characterized as dynamic, weakly typed, prototype-based and multi-paradigm. It is the only supported language by the Chrome Extension API.
- NPM: package manager for JavaScript. Used to specify and lock dependencies of the project such as Bucklescript, Webpack, Chrome API.

- Webpack: it is a module, asset, and dependency bundler. Used to combine all the JavaScript files generated by BuckleScript into a single index.js file.
- Chrome Extension API: Chrome Extension API and development kits. Used to interact with the storage and network components of the Chrome browser.
- React: a library to create user interfaces. Used to modularize the Chrome Extension user interfaces.
- Editorconfig: style linter. Used to homogenize the style across all the developers of the Chrome Extension.
- TravisCI: Continuous Integration web service. Used to run the test suite automatically on every branch for every commit.

Design Decisions

From a high level perspective the Chrome Extension has four main React components:

- The App component is the parent component which holds the global state. It keeps track of the JWT and the UserID. It displays the Login or the JobApp form depending on whether the JWT is still valid or has expired.
- The Login component has an HTML form to enter JobHub credentials or a button to register for the service.
- The JobApp component renders the job application form for the user to submit.
- The ScrapingInput components are HTML input fields which scrape information form the website currently open by the user. They scrape URL, company name, position, and date posted.

```
        popup.html
            |
          App.re
      {token, userID}
            |
   +-----------+-----------+
   |                       |
 Login.re              JobApp.re
                           |
                   ScrapingInputs.re
```

`token` is the state variable which is initially obtained by `Login` during the authentication process and in subsequent executions of the extension its validity is confirmed by `App` directly. `userID` is also obtained during the initial login process. In case of a deprecated `token`, both the `userID` and the `token` are cleared from the browser's memory. Both variables are passed onto the `JobApp` component to be able to fetch CVs and to submit job applications.

The system leverages the following modules:

- `ScrapingFunctions.re`: functions related to extracting/processing HTML elements
- `Services.re`: functions related to asynchronous actions external to the extension (load files/API calls)
- `SyncStorage.re`: functions related to the Chrome storage management
- `Uilities.re`: general purpose helper functions

While using a language other than JavaScript for the Chrome Extension significantly complexified our technological stack, we decided in favour of using ReasonML because of the powerful and stable type system that OCaml offers. We were able to increase the confidence we had in our code with significantly less tests because of the default safe guards that the Bucklescript compiler provided. Additionally, the functional paradigm that ReasonML offered, allowed for more elegant and maintainable components across the code base.

We chose to use React as our user interface library over vanilla HTML/JS/CSS, Angular, or VueJS because the developers already had experience with the library. Hence, the sprint speed of the team for user interface tasks was significantly increased. An additional factor was that ReasonML offered out of the box support for React (both libraries are developed by Facebook). React allowed us to modularize the interfaces of the extension into the above mentioned components. This allowed for a decoupled structure with its own internal logic and state.

We chose to provide an extension only for Chrome and not for Internet Explorer or Firefox due to the constraint in time and resources. We decided to focus on the most widely used browser and the one with the most updated API. An added advantage of using the Chrome browser for the extension was that it has synchronized storage across all browsers the user is currently logged in. This means that logging in the JobHub extension in a laptop will also allow the user to remain logged in (given that the JWT does not expire) in his desktop. A drawback of all current browsers is that none support extensions in mobile platforms. Hence the JobHub Extension remains a non-mobile component for the foreseeable future. We chose not to publish the Chrome Extension in the Chrome Marketplace as it costs $40 USD to create a developer account with Google to be able to publish an extension. Information on how to setup the Chrome Extension from source has been given in the *setup* section.

We chose editorconfig for the linting of the code base to homogenize developers' code. While the Bucklescript already performs major linting of the code before compilation, minor details such as maximum line length and required documentation are checked by editorconfig.

Task Distribution

The first sprint team was comprised of Alexander Bratyshkin and Camilo Garcia La Rotta.

- **Alexander Bratyshkin**: he was mainly in charge of setting up the project structure and developing the scrappers. He wired up the Bucklescript compiler with the Webpack bundler, effectively completing the pipeline from ReasonML code to valid Chrome API Javascript. He created the components required to scrape company names and job positions.
- **Camilo Garcia La Rotta**: he was mainly in charge of creating the Login and Job Posting forms as well as connecting them to the appropriate backend service APIs. During the first sprint he completed the Login sequence Use Case as well as implementing the date posted scraper.

The second sprint team was solely comprised of Camilo Garcia La Rotta.

- **Camilo Garcia La Rotta**: he focused on connecting the JobApp form to the CV revision and Job postings API. He added the test suite for all external facing methods of the application as well as adding features specified in the use cases such as avoid submitting an application twice.

Limitations

- Currently, the only supported `posted date` scraper pattern is `"<num> days ago"`
- Currently, the extension does not have a `deadline date` scraper
- Because of the bucklescript-chrome bindings used to build the extension, we are locked with outdated versions of core libraries such as `React` and `Bucklescript`
- For the same reason as above, we can't use Chrome's Content Scripts. Instead we inject a script into the active website to extract DOM information.

Miscellaneous

The team chose to develop the Chrome Extension on a issue based approach. This meant that an issue was to be created for each feature to be developed, so that every pull request had an issue assigned to it and the issue would be closed automatically by GitHub once merged into master. The project also followed semantic commit messages in order to reduce the cognitive load required to understand the changes done to the codebase since a given commit. The naming convention for every branch would hence be `feat/<short_issue_description>-<issue-number>` for a given feature being implemented. This allowed the team to quickly parse the multiple branches being worked on concurrently.