CRISP

for

Cloud Computing (INFS 803)

by

GROUP 8

Juan Herbst (13840146)

Mike Chene (19070279)

Tommaso Cammelli (23215488)

Auckland University of Technology, Auckland, New Zealand

Table of Contents

[Introduction 4](#_Toc166788682)

[Project Planning 4](#_Toc166788683)

[Project Design 5](#_Toc166788684)

[User Experience (UX) & Screen Mock-ups 5](#_Toc166788685)

[Cloud Service Architecture 9](#_Toc166788686)

[Technology Stack 11](#_Toc166788687)

[Interactions and Sequence Diagrams 12](#_Toc166788688)

[Schema Design 13](#_Toc166788689)

[REST API Endpoints. 18](#_Toc166788690)

[User Manual 19](#_Toc166788691)

[Summary 20](#_Toc166788692)

Table of Figures

[Figure 1 Crisp Kanban Board for Project Planning. 4](#_Toc166788697)

[Figure 2 Mock-up: Sign-up Screen. 5](#_Toc166788698)

[Figure 3 Email Verification Code sent during the Sign-up Process. 5](#_Toc166788699)

[Figure 4 Mock-up: Sign-in Screen. 6](#_Toc166788700)

[Figure 5 Mock-up: Recruiter Profile Screen. 6](#_Toc166788701)

[Figure 6 Mock-up: Jobseeker Profile Screen. 7](#_Toc166788702)

[Figure 7 Mock-up: Job Posting Screen. 8](#_Toc166788703)

[Figure 8 Mock-up: Job Application Flow and Screens. 8](#_Toc166788704)

[Figure 7 Mock-up: Find Job Seekers by Category Flow and Screens. 9](#_Toc166788705)

[Figure 10 Cloud Services used by Crisp. 10](#_Toc166788706)

[Figure 11 Sequence Diagram depicting interaction between the User, Cognito, API Gateway, Lambda and DynamoDB. 12](#_Toc166788707)

[Figure 12 Sequence Diagram depicting interaction between the User, Cognito and S3. 12](#_Toc166788708)

[Figure 13 Entity Relationship Diagram (ERD) 13](#_Toc166788709)

TABLE OF TABLES

[Table 1 JSON Schema: Recruiter Profile. 14](#_Toc166788710)

[Table 2 JSON Schema: Jobseeker Profile. 15](#_Toc166788711)

[Table 3 JSON Schema: Notes. 16](#_Toc166788712)

[Table 4 JSON Schema: Categories. 17](#_Toc166788713)

[Table 5 JSON Schema: Jobs. 17](#_Toc166788714)

[Table 6 REST Endpoints: Recruiter Profiles. 18](#_Toc166788715)

[Table 7 REST Endpoints: Jobseeker Profiles. 18](#_Toc166788716)

[Table 8 REST Endpoints: Notes. 19](#_Toc166788717)

[Table 9 REST Endpoints: Categories. 19](#_Toc166788718)

[Table 10 REST Endpoints: Jobs. 19](#_Toc166788719)

## Introduction

Crisp presents a complete solution aimed at enabling both jobseekers and recruiters. For job seekers, Crisp offers an intuitive platform that facilitates the creation of personalised profiles, enhanced by the capability to upload introductory videos and traditional CVs. This multimedia integration exceeds the limitations of conventional resumes, encouraging jobseekers to highlight their competencies, professional experiences, and individuality. Conversely, on the recruiter's end, Crisp provides hiring practitioners with tools for effortless job listing posting, jobseeker evaluation, and interaction. Moreover, Crisp establishes smooth communication channels, nurturing substantive engagements between recruiters and jobseekers throughout the recruitment.

## Project Planning

Scrum, a widely adopted agile framework, was used during the project design, development, and testing phases. Founded on transparency, inspection, and adaptation principles, Scrum empowers cross-functional teams to deliver high-quality products incrementally within short time frames, known as sprints. Scrum fosters a culture of continuous improvement and responsiveness to changing requirements through its structured ceremonies, including daily stand-ups, sprint planning, reviews, and retrospectives. Trello effectively tracked and managed the product's roadmap and maintained thorough oversight of the fortnightly sprint backlog.

Figure 1  
Crisp Kanban Board for Project Planning.

A screenshot of a computer

Description automatically generated

## Project Design

### User Experience (UX) & Screen Mock-ups

Screen mock-ups visually represent the user interface, giving stakeholders a tangible preview of the final product's layout and functionality. These mock-ups facilitate collaborative discussions, enabling teams to iterate on design concepts, gather feedback, and refine user interactions before implementation.

The mock-up screens may differ from the final product, reflecting the ideation phases during the design and development.

Figure 2  
Mock-up: Sign-up Screen.

A screenshot of a login form

Description automatically generated

The user initiates the sign-up process by providing their email address and creating a password, which they confirm to ensure accuracy. Additionally, during the registration process, users are prompted to select their intended role as a recruiter or job seeker.

Figure 3  
Email Verification Code sent during the Sign-up Process.

A screenshot of a computer

Description automatically generated

The user will undergo an authentication step throughout the registration process, whereby a verification code is sent to their provided email address. This verification code serves to validate the authenticity of the user's submitted email address. Upon receipt of the verification code, the user is prompted to enter it into the designated field to confirm the validity of their sign-up email address. This measure ensures the integrity and security of the sign-up process by verifying the validity of the user's contact information.

Figure 4   
Mock-up: Sign-in Screen.

A screenshot of a login form

Description automatically generated

Once the user is registered, they can sign in to Crisp using the email and password created during the sign-up process.

Figure 5  
Mock-up: Recruiter Profile Screen.

A screenshot of a computer

Description automatically generated

If registered users sign up as a recruiter, they must complete their profile, including their name, last name, and the organisation they work for. The read-only email field will be populated with the email provided during sign-up.

Figure 6  
Mock-up: Jobseeker Profile Screen.

A screenshot of a computer

Description automatically generated

Upon registering as jobseekers, users must complete their profile by providing their first name, last name, and date of birth. The read-only email field will automatically be populated with the email address submitted during registration. Users can also upload a digital or a traditional curriculum vitae (CV) and a profile picture. Initially, the list of job applications will be empty; however, as users apply for positions, this list will be populated accordingly.

Figure 7  
Mock-up: Job Posting Screen.

A screenshot of a computer

Description automatically generated

To create a job posting, the recruiter navigates to the job posting URL, selects a job category from the pre-populated list, enters a job title, provides a job description, and specifies the offered salary.

Figure 8   
Mock-up: Job Application Flow and Screens.

A screenshot of a computer

Description automatically generated

The jobseeker initiates navigation to the available jobs page, where they are prompted to select a category of interest. Upon selecting a specific category, the system retrieves and displays a list of jobs corresponding to the chosen category. To view the details of a particular job, the jobseeker double-clicks on the respective job listing, thereby accessing comprehensive job information. Within the job detail view, the job seeker can apply.

Figure 7  
Mock-up: Find Job Seekers by Category Flow and Screens.

A screenshot of a computer screen

Description automatically generated

The recruiter navigates to the job category screen to find job seekers who have applied for a specific role. Here, they select a job category from a pre-populated drop-down list, which will subsequently populate a list of jobs within that category. A pop-up window will appear when double-clicking on a job, displaying a list of candidates who have applied for that position. Double-clicking on a particular candidate will open their profile, displaying their first name, last name, date of birth, and email (the latter being read-only). The only editable field is a text box where the recruiter can enter notes regarding the job seeker. Additionally, the recruiter can download both the digital and traditional CVs of the candidate.

### Cloud Service Architecture

This paper introduces Amazon Web Services (AWS) as the cloud platform of choice for the CRISP project. Leveraging AWS's robust, cost-effective infrastructure, extensive service offerings, and proven reliability, the CRISP project aims to enhance data privacy, ensure high availability, and facilitate seamless scalability. The following sections will provide an in-depth exploration of the specific AWS services utilized in the CRISP project, demonstrating how each service contributes to the project's overall objectives and technical requirements.

*Cognito* is an identity management service offered by AWS. It facilitates secure user authentication and authorisation for web and mobile applications. Amazon Cognito provides users with various features, including user sign-up, sign-in, account recovery, profile management, and data synchronisation across multiple devices. Furthermore, it supports multi-factor authentication (MFA), social identity provider integration, and fine-grained access control policies.

*Route 53* is a highly scalable and reliable Domain Name System (DNS) provided by AWS. Route 53 effectively translates human-readable domain names into corresponding numeric IP addresses by routing user requests across the Internet. Route 53 offers a range of functions, including domain registration, DNS health monitoring, traffic routing policies, and domain name system security extensions (DNSSEC).

*Simple Storage Service (S3)* provides a secure and reliable platform for storing and retrieving objects such as large files. With its distributed architecture spanning multiple geographic regions, Amazon S3 ensures high durability and availability, making it suitable for various applications, from data backup and archival to content delivery and big data analytics.

Figure 10  
Cloud Services used by Crisp.

A diagram of a computer server

Description automatically generated

*API Gateway* is a fully managed service provided by AWS that allows developers to create, publish, maintain, monitor, and secure APIs at any scale. As a front door for applications to access data, business logic, or functionality from backend services, API Gateway simplifies the process of building and managing APIs. Leveraging API Gateway, developers can define RESTful APIs or WebSocket APIs, configure endpoints, handle authentication and authorisation, and enforce usage policies.

*Lambda* is a serverless computing service that enables developers to run code without provisioning or managing servers. Lambda automatically scales and manages the compute resources needed to execute the code in response to incoming requests. Lambda supports various programming languages, including Node.js, Python, Java, and Go, allowing developers to write functions that respond to events triggered by AWS services or custom events.

*DynamoDB* is a fully managed NoSQL database service designed to provide high performance, scalability, and reliability for applications requiring low-latency data access.

### Technology Stack

The Lambda functions were developed using Node.js version 18 and Express, a lightweight web framework characterised by its rapid processing capabilities and minimalistic design philosophy. Furthermore, TypeScript and React were selected to facilitate the development of dynamic user interfaces for front-end applications. Vite was used to optimise the build process, ensuring efficient compilation of project assets. In addition, Tailwind CSS was adopted to streamline the styling of the application, promoting swift responsiveness and usability. Testing frameworks such as Vitest for unit testing and Playwright for end-to-end testing were integrated into the development workflow to maintain code integrity.

Continuous Integration and Deployment Pipelines (CI/CD) were established to streamline the deployment of the back and front-end components triggered by successful pull requests to the main branch, maintaining a consistent update cycle. GitHub served as the designated platform for source control management, facilitating versioning and collaborative efforts among team members.

### Interactions and Sequence Diagrams

Figure 11   
Sequence Diagram depicting interaction between the User, Cognito, API Gateway, Lambda and DynamoDB.

A diagram of a software company

Description automatically generated

The user initiates the authentication process by validating their email address and password against Cognito. Upon successful authentication, a security token is issued to the user, allowing them to perform authenticated actions within AWS. Requests are made by providing the security token and URL to the desired REST resource, facilitated by API Gateway. This service routes requests to a serverless backend implemented using Lambda, where business logic is executed. Lambda functions interact with DynamoDB to retrieve and process data. Retrieved data is transformed and subjected to further business logic within Lambda functions before being delivered back to the user's browser through AWS API Gateway, completing the transaction cycle.

Figure 12   
Sequence Diagram depicting interaction between the User, Cognito and S3.

A diagram of a diagram

Description automatically generated

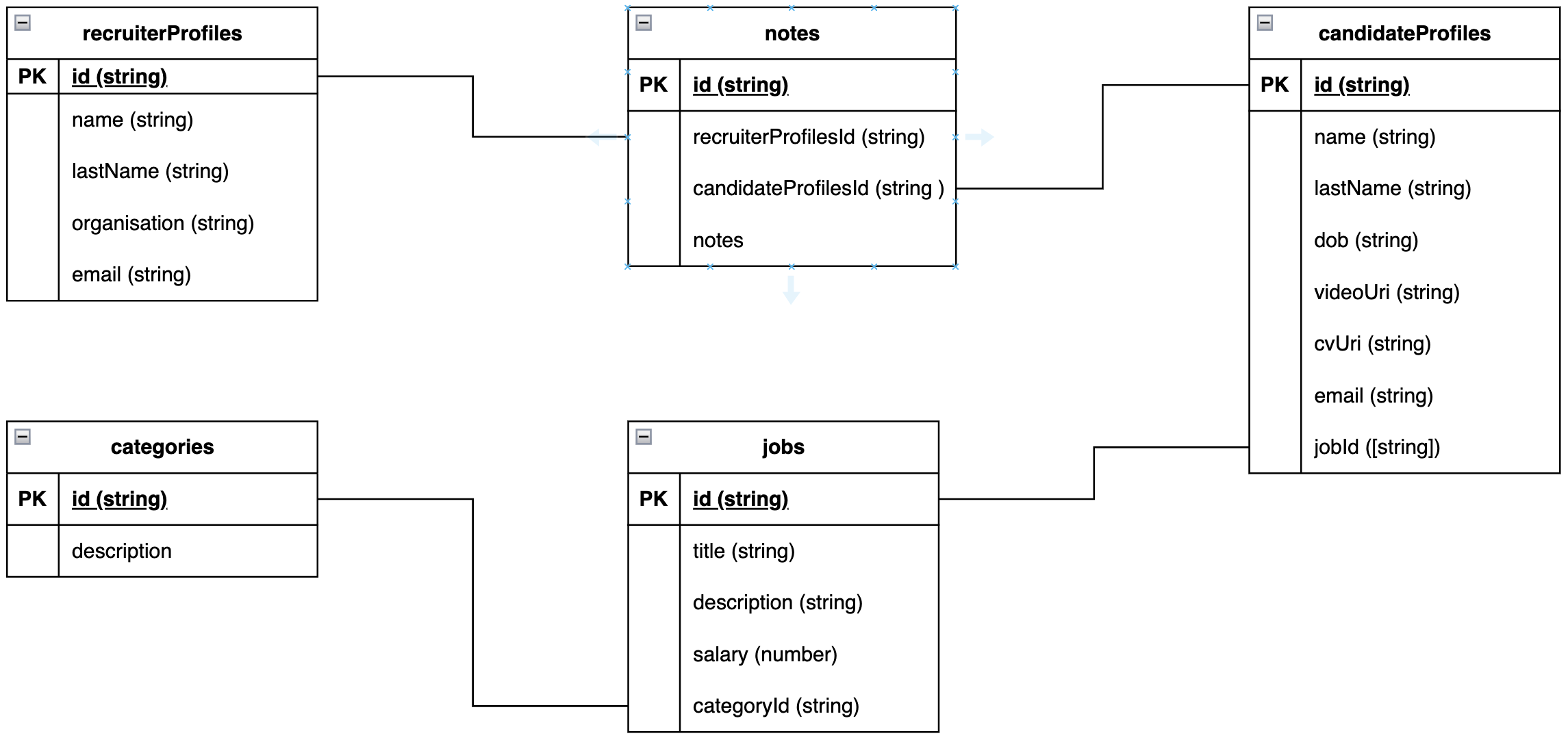
The authentication process begins as the user submits their email address and password for validation against Cognito. Upon validation, an authentication token is provided to the user, granting access to authenticated actions within AWS. Objects like files, videos, and images are also uploaded to S3. Conversely, files, videos, and images are retrieved from the S3 storage repository through a download process.

### Schema Design

Schema design is an essential component of database management. It encompasses the arrangement and organisation of data within a database system and involves creating data models that delineate the structure, relationships, and constraints of the data stored in the database.

In the context of NoSQL databases like DynamoDB, it's noteworthy that conventional relationships between tables, as seen in relational databases, are absent. Instead, NoSQL databases often employ informal relationships or data denormalisation to optimise data access patterns and enhance performance.

Figure 13  
Entity Relationship Diagram (ERD)

****

The JSON schema for a recruiter profile delineates the structure and organisation of personal data stored for each recruiter within the system. This schema includes several key functions. The "id" field, represented as a string, serves as a unique identifier for each recruiter profile, enabling efficient retrieval and management of individual records. Additionally, the schema incorporates fields such as "name" and "lastName," both string types, to capture the recruiter's first and last names, respectively. The "organisation" field, also of string type, facilitates the storage of information regarding the recruiter's affiliated organisation or company. Furthermore, the schema includes an "email" field, represented as a string, designed to store the recruiter's email address, providing a means for communication and correspondence.

Table 1  
JSON Schema: Recruiter Profile.

|  |
| --- |
| 1. {  2. "$schema": "http://json-schema.org/draft-04/schema#",  3. "type": "object",  4. "properties": {  5. "id": {  6. "type": "string"  7. },  8. "name": {  9. "type": "string"  10. },  11. "lastName": {  12. "type": "string"  13. },  14. "organisation": {  15. "type": "string"  16. },  17. "email": {  18. "type": "string"  19. }  20. },  21. "required": [  22. "id",  23. "name",  24. "lastName",  25. "organisation",  26. "email"  27. ]  28. } |

The JSON schema for a jobseeker profile encompasses essential fields intended to capture relevant personal data for jobseekers within the Crisp project. These fields include "id" as a string, serving as a unique identifier for each jobseeker profile; "name" and "lastName" as strings to record the jobseeker's given and family names, respectively; and "dob" as a string to denote the jobseeker's date of birth. Additionally, the schema features "email" as a string, facilitating communication with jobseekers via their designated email addresses. Notably, the schema incorporates "videoUri" and "cvUri" fields to store Uniform Resource Identifiers (URIs) pointing to videos and curriculum vitae (CVs) stored in S, three respectively. Furthermore, the schema accommodates an array of "jobIds" to record the identifiers of jobs to which jobseekers have applied, facilitating tracking, and managing jobseeker applications.

Table 2  
JSON Schema: Jobseeker Profile.

|  |
| --- |
| 1. {  2. "$schema": "http://json-schema.org/draft-04/schema#",  3. "type": "object",  4. "properties": {  5. "id": {  6. "type": "string"  7. },  8. "name": {  9. "type": "string"  10. },  11. "lastName": {  12. "type": "string"  13. },  14. "email": {  15. "type": "string"  16. },  17. "dob": {  18. "type": "string"  19. },  20. "videoUri": {  21. "type": "string"  22. },  23. "cvUri": {  24. "type": "string"  25. },  26. "jobId": {  27. "type": "array",  28. "items": [  29. {  30. "type": "string"  31. }  32. ]  33. }  34. },  35. "required": [  36. "id",  37. "name",  38. "lastName",  39. "email",  40. "dob",  41. "videoUri",  42. "cvUri",  43. "jobId"  44. ]  45. }  The JSON schema for a note encapsulates key fields necessary for recording and managing notes associated with interactions between recruiting jobseekers. This schema includes "id" as a string, a unique identifier for each note entry. Additionally, the schema features "recruiterProfileId" and "jobseekerProfileId" as strings, facilitating the association of notes with the specific recruiter and jobseeker profiles, respectively. |

Table 3  
JSON Schema: Notes.

|  |
| --- |
| 1. {  2. “$schema”: “http://json-schema.org/draft-04/schema#”,  3. "type": "object",  4. "properties": {  5. "id": {  6. "type": "string"  7. },  8. “recruiterProfileId”: {  9. "type": "string"  10. },  11. “jobseekerProfileId”: {  12. "type": "string"  13. },  14. "note": {  15. "type": "string"  16. }  17. },  18. "required": [  19. "id",  20. “recruiterProfileId”,  21. “jobseekerProfileId”,  22. "note"  23. ]  24. } |

The JSON schema for a category is a foundational structure for organising and managing job categories. This schema encompasses essential fields for effectively categorising and describing different job types. The schema includes “id” as a string and functions as a unique identifier for each category entry, ensuring distinct identification and retrieval of specific categories. Additionally, the “description” field, a string, provides a textual representation of the category.

Table 4  
JSON Schema: Categories.

|  |
| --- |
| 1. {  2. “$schema”: “http://json-schema.org/draft-04/schema#”,  3. "type": "object",  4. "properties": {  5. "id": {  6. "type": "string"  7. },  8. "description": {  9. "type": "string"  10. }  11. },  12. "required": [  13. "id",  14. "description"  15. ]  16. } |

The JSON schema for representing job entities involves several fields for job descriptions. Firstly, the “id” attribute, a string, is a unique identifier for each job entry, ensuring its distinct identification and retrieval within the system. Subsequently, the “title” field, also specified as a string, encapsulates a prescriptive title for the job role. Moreover, the “description” attribute, a string, provides a detailed overview of the job’s responsibilities, qualifications, and other pertinent information. Additionally, the “salary” field, represented as a number, denotes the monetary compensation associated with the job. Lastly, the “categoryId” attribute, designated as a string, associates the job with a specific category, enabling efficient categorisation and organisation of job listings based on their respective domains or industries.

Table 5  
JSON Schema: Jobs.

|  |
| --- |
| 1. {  2. “$schema”: “http://json-schema.org/draft-04/schema#”,  3. "type": "object",  4. "properties": {  5. "id": {  6. "type": "string"  7. },  8. "title": {  9. "type": "string"  10. },  11. "description": {  12. "type": "string"  13. },  14. "salary": {  15. "type": "string"  16. },  17. "categoryId": {  18. "type": "string"  19. }  20. },  21. "required": [  22. "id",  23. "title",  24. "description",  25. "salary",  26. "categoryId"  27. ]  28. } |

### REST API Endpoints.

The Representational State Transfer (REST) Application Programming Interfaces (APIs) are deployed utilising the Hypertext Transfer Protocol Secure (HTTPS), a protocol that ensures secure communication over computer networks, particularly the Internet.

Within the context of the Crisp project, the <https://api.crisp.nz> custom domain serves as the deployment endpoint for the REST APIs, facilitating interactions with backend services.

In contrast, the application's front-end components are deployed on the https://crisp.nz domain, which hosts the user interface and facilitates user interactions with the web application's features and content.

Table 6   
REST Endpoints: Recruiter Profiles.

|  |  |  |
| --- | --- | --- |
| Endpoint | Verb | Action |
| <https://api>.crisp.co.nz/recruiters/profiles | POST | Create a Profile. |
| <https://api>.crisp.co.nz/recruiters/profiles | GET | Get all Profiles. |
| <https://api>.crisp.co.nz/recruiters/profiles/{profileId} | GET | Get a Profile by ID. |
| <https://api>.crisp.co.nz/recruiters/profiles/ | PUT | Update a Profile. |
| <https://api>.crisp.co.nz/recruiters/profiles/{profileId} | DELETE | Delete a Profile by ID. |

Table 7   
REST Endpoints: Jobseeker Profiles.

|  |  |  |
| --- | --- | --- |
| Endpoint | Verb | Action |
| <https://api.crisp.co.nz/jobseekers/profiles> | POST | Create a Profile. |
| <https://api.crisp.co.nz/jobseekers/profiles> | GET | Get all Profiles. |
| [https://api.crisp.co.nz/jobseekers/profiles/{profileId}](https://api.crisp.co.nz/jobseekers/profiles/%7bprofileId%7d) | GET | Get a Profile by ID. |
| <https://api.crisp.co.nz/jobseekers/profiles/> | PUT | Update a Profile. |
| [https://api.crisp.co.nz/jobseekers/profiles/{profileId}](https://api.crisp.co.nz/jobseekers/profiles/%7bprofileId%7d) | DELETE | Delete a Profile by ID. |

Table 8  
REST Endpoints: Notes.

|  |  |  |
| --- | --- | --- |
| Endpoint | Verb | Action |
| <https://api.crisp.co.nz/notes/> | POST | Create a Note. |
| [https://api.crisp.co.nz/notes/](https://api.crisp.co.nz/notess) | GET | Get all Notes. |
| [https://api.crisp.co.nz/notes/](https://api.crisp.co.nz/notess){noteId} | GET | Get a Note by ID. |
| [https://api.crisp.co.nz/notes/](https://api.crisp.co.nz/notess) | PUT | Update a Note. |
| [https://api.crisp.co.nz/notes/](https://api.crisp.co.nz/notess){noteId} | DELETE | Delete a Note by ID. |

Table 9  
REST Endpoints: Categories.

|  |  |  |
| --- | --- | --- |
| Endpoint | Verb | Action |
| <https://api.crisp.co.nz/categories/> | POST | Create a Category. |
| <https://api.crisp.co.nz/categories/> | GET | Get all Categories. |
| <https://api.crisp.co.nz/categories/>{categoryId} | GET | Get a Category by ID. |
| <https://api.crisp.co.nz/categories/> | PUT | Update a Category. |
| <https://api.crisp.co.nz/categories/>{categoryId} | DELETE | Delete a Category by ID. |

Table 10   
REST Endpoints: Jobs.

|  |  |  |
| --- | --- | --- |
| Endpoint | Verb | Action |
| <https://api.crisp.co.nz/jobs/> | POST | Create a Job. |
| <https://api.crisp.co.nz/jobs/> | GET | Get all Jobs. |
| <https://api.crisp.co.nz/jobs/>{jobId} | GET | Get a Job by ID. |
| <https://api.crisp.co.nz/jobs/> | PUT | Update a Job. |
| <https://api.crisp.co.nz/jobs/>{jobId} | DELETE | Delete a Job by ID. |

## User Manual

TODO

## Summary

The CRISP project is deployed on AWS, renowned for its cost-effectiveness, scalability, resilience, and availability. CRISP is a single-page application hosted in Amazon S3 and accessed via the custom domain crisp.nz. The backend adheres to the Platform as a Service (PaaS) model and leverages serverless technology. The design, development, and quality assurance processes used Scrum as the project delivery methodology. Standard DevOps practices were employed, including using Amazon CloudWatch for logging and monitoring, with metrics sourced from native AWS services such as DynamoDB. Continuous Integration (CI) practices are implemented using Gitflow and GitHub for source control, whereas Continuous Deployment (CD) is achieved by merging code into the main branch through pull requests, ensuring code quality via the code review process.