JuaJobs API Design Documentation

Overview

JuaJobs is a gig economy platform designed to connect skilled workers across Africa (plumbers, electricians, artisans, etc.) with clients seeking their services. This API documentation outlines the platform's core functionality, focusing on job postings, worker profiles, and job applications.

Business Context

In many African markets, skilled workers face challenges in finding consistent work, while clients struggle to find reliable, vetted professionals for their needs. JuaJobs bridges this gap by providing:

1. **Digital presence** for skilled workers who traditionally rely on word-of-mouth
2. **Verification and review systems** to build trust
3. **Location-based matching** to connect local workers with nearby opportunities
4. **Secure payment options** adapted to local preferences
5. **Mobile-first experience** to account for the mobile-dominant African internet landscape

Core Resources

1. Users

Users are divided into two primary roles:

- **Clients**: Post jobs and hire workers
- **Workers**: Create profiles and apply for jobs

Both types share common user attributes but have role-specific functionalities.

2. Job Postings

Job postings are created by clients and represent work opportunities.

3. Worker Profiles

Worker profiles showcase skills, experience, and availability of service providers.

4. Applications

Applications represent a worker's interest in a specific job posting and the subsequent hiring workflow.

5. Reviews

Reviews provide feedback on both workers and clients after job completion.

Phase 1: Resource Modelling

Delivered by: All Team Members

1. Resource Identification & Analysis
    - Extend the initial resource list (Users, Job Postings, Worker Profiles, Applications, Reviews)
    - Add at least 3 additional resources that would enhance the platform
    - Justify the need for each resource within the business context
    - Analyse resource ownership and lifecycle considerations

2. Resource Attribute Design
    - Create detailed data models for each resource with complete attribute lists
    - Classify attributes (required vs. optional, system-generated vs. user-provided)
    - Define data types and validation rules for each attribute
    - Document any computed or derived attributes
3. Resource Relationships
    - Map relationships between all resources with proper cardinality
    - Create an entity-relationship diagram showing the complete resource ecosystem
    - Identify parent-child relationships and containment hierarchies
    - Document resource dependencies and referential integrity requirements

Additional resources

1. **Payments/Transactions**: This resource enables secure and efficient payment processing for freelancers and job seekers. It will help to build trust within the platform, ensuring that clients and workers can rely on the system

for financial transactions.

2. **Message:** The message resource provides a platform for users to communicate directly with each other, facilitating collaboration, feedback, and relationship-building. This feature will enhance the overall user experience and encourage more effective communication.

3. **Products/ Projects**: The product resource allows service providers to showcase their skills and offer services, enabling clients to browse and select from a variety of options. This feature will improve the discovery process and increase the likelihood of successful matches between clients and workers.

Resource Ownership and Lifecycle Considerations for All Core Resources

Fundamental Resources

Users:
Ownership: Users (clients and workers)
Lifecycle:
Creation: When a user signs up for the platform
Update: When a user updates their profile information
Deletion: When a user chooses to delete their account

Job Postings:
Ownership: Clients
Lifecycle:
Creation: When a client creates a new job posting
Update: When a client updates a job posting
Deletion: When a client deletes a job posting

Profiles:
Ownership: Workers
Lifecycle:
Creation: When a worker creates a profile
Update: When a worker updates their profile information
Deletion: When a worker chooses to delete their profile

Applications:
Ownership: Workers
Lifecycle:
Creation: When a worker applies for a job posting
Update: When a worker updates their application status
Deletion: When a worker chooses to withdraw their application

Reviews:

Ownership: Clients and workers
Lifecycle:
Creation: When a client or worker leaves a review
Update: When a review is updated or corrected
Deletion: When a review is removed due to inactivity or other reasons

Additional Resources

Payments/Transactions:

Ownership: The platform (for transaction processing) and users (for payment details)
Lifecycle:
Creation: When a payment is initiated
Update: When a payment status changes (e.g., from pending to successful)
Deletion: When a payment is cancelled or refunded

Message:

Ownership: Users (senders and recipients)
Lifecycle:
Creation: When a message is sent
Update: When a message is updated or replied to
Deletion: When a message is deleted or marked as read

Products:

Ownership: Workers (service providers)
Lifecycle:
Creation: When a worker creates a product listing
Update: When a worker updates their product listing
Deletion: When a worker chooses to delete their product listing

Phase 2: API Design Excellence
Delivered by: Yusuf Molumo

1. Endpoint Design (Led by Endpoint Designer)
We adopted a RESTful approach to API endpoint design, aligning each endpoint with a resource or a sub-resource. Resources were identified based on our domain model, with a focus on user-centric design. Each endpoint serves a specific purpose and was crafted to reflect the data it represents, ensuring predictability and ease of use for API consumers.

2. Create a comprehensive endpoint list following RESTful conventions

1. User

Endpoint	Description
GET /v1/users	Retrieve all users
GET /v1/users/{id}	Retrieve a specific user
POST /v1/users	Create a new user
PUT /v1/users/{id}	Fully update a user
PATCH /v1/users/{id}	Partially update a user
DELETE /v1/users/{id}	Delete a user

2. Job Postings

Endpoint	Description
GET /v1/jobs	Retrieve all job postings
GET /v1/jobs/{id}	Retrieve a specific job posting
POST /v1/jobs	Create a new job posting
PUT /v1/jobs/{id}	Fully update a job posting
PATCH /v1/jobs/{id}	Partially update a job posting
DELETE /v1/jobs/{id}	Delete a job posting
GET /users/{id}/jobs	Retrieve job postings created by a client

3. Worker Profiles

Endpoint	Description
GET /v1/profiles	Retrieve all worker profiles

GET /v1/profiles/{id}	Retrieve a specific worker profile
POST /v1/profiles	Create a new worker profile
PUT /v1/profiles/{id}	Fully update a worker profile
PATCH /v1/profiles/{id}	Partially update a worker profile
DELETE /v1/profiles/{id}	Delete a worker profile
GET /v1/users/{id}/profile	Retrieve profile for a specific worker

4. Applications

Endpoint	Description
GET /v1/applications	Retrieve all job applications
GET /v1/applications/{id}	Retrieve a specific application
POST /v1/applications	Submit a new application
PUT /v1/applications/{id}	Fully update an application
PATCH /v1/applications/{id}	Partially update application status
DELETE /v1/applications/{id}	Withdraw/delete an application
GET /v1/jobs/{id}/applications	Get all applications for a specific job
GET /v1/users/{id}/applications	Get all applications by a specific worker

5. Reviews

Endpoint	Description
GET /v1/reviews	Retrieve all reviews
GET /v1/reviews/{id}	Retrieve a specific review
POST /v1/reviews	Create a new review
PUT /v1/reviews/{id}	Fully update a review
PATCH /v1/reviews/{id}	Partially update a review
DELETE /v1/reviews/{id}	Delete a review

GET /v1/users/{id}/reviews	Retrieve reviews made by or about a user

6. Payments / Transactions

Endpoint	Description
GET /v1/payments	Retrieve all payments
GET /v1/payments/{id}	Retrieve a specific payment transaction
POST /v1/payments	Initiate a new payment
PATCH /v1/payments/{id}	Update payment status (e.g., success/failure)
DELETE /v1/payments/{id}	Cancel a payment
GET /v1/users/{id}/payments	Get payment history for a specific user

7. Messages (Inbox / Chat)

Endpoint	Description
GET /v1/messages	Retrieve all messages
GET /v1/messages/{id}	Retrieve a specific message
POST /v1/messages	Send a new message
PATCH /v1/messages/{id}	Update a message (e.g., mark as read)
DELETE /v1/messages/{id}	Delete a message
GET /v1/users/{id}/messages	Get messages sent to or from a user

8. Products

Endpoint	Description
GET /v1/products	Retrieve all worker products
GET /v1/products/{id}	Retrieve a specific product
POST /v1/products	Create a new product listing
PUT /v1/products/{id}	Fully update a product
PATCH /v1/products/{id}	Partially update a product
DELETE /v1/products/{id}	Delete a product listing
GET /v1/users/{id}/products	Get all products posted by a specific worker

3. Specify HTTP methods (GET, POST, PUT, PATCH, DELETE) with clear justification

- GET: Used for read-only operations without side effects.
- POST: Used to create new resources.
- PUT: Used for full updates of an existing resource, replacing it entirely.
- PATCH: Used for partial updates to existing resources.
- DELETE: Used to remove a resource permanently.

4. Define URL patterns that reflect resource hierarchies and relationships
URL patterns follow a nested structure to show clear ownership and relationships:

- /v1/users/{userId}/projects – Projects that belong to a specific user

5. Design consistent naming conventions for endpoints and parameters

- Resource names are in plural form: /users, /projects, /donations
- Parameters use camelCase: sortBy, filterStatus, userId
- URLs are lowercase with hyphens for readability (if needed): /project-updates
- Consistency is enforced through a centralised style guide

6. Query Parameters & Filtering (Led by User Experience Analyst)
Filtering, sorting, and pagination follow a standardised convention to allow flexible and efficient data access.
Example:

- GET /v1/projects?category=education&status=active&sortBy=createdAt&order=desc&page=2&limit=10

7. Design a standardised approach to filtering, sorting, and pagination

- Filtering: Achieved via query parameters (?status=active)
- Sorting: sortBy and order query parameters
- Pagination: page and limit parameters with default values for each

8. Define field selection parameters (sparse fieldsets)

To reduce payload size and optimise performance, sparse fieldsets are used:
Example:

- GET /v1/users?fields=id,name,email
- GET /v1/projects?fields=id,title,location

This allows API consumers to request only the data they need.

9. Create patterns for complex data queries (full-text search, geo-proximity, etc.)

- Full-text search: /v1/projects?search=clean+water+initiative
- Geo-proximity: /v1/projects?lat=1.95&lng=30.05&radius=10km
- Date filters: /v1/projects?from=2023-01-01&to=2023-12-31

These patterns support advanced use cases and improve user relevance.

10. Document parameter validation rules and error handling

Each endpoint includes validation logic to ensure:

- Required parameters are provided
- Parameters match expected types (e.g., strings, integers, dates)
- Invalid inputs return structured error responses with clear messaging

Example error:

```
{
  "error": "InvalidParameter",
  "message": "The parameter 'userId' must be a valid UUID.",
  "status": 400
}
```

11. HTTP Status Codes & Error Handling (Led by API Architect)

Standard HTTP status codes were mapped to API responses:

- 200 OK – Successful GET/PUT/PATCH
- 201 Created – Successful POST
- 204 No Content – Successful DELETE
- 400 Bad Request – Validation error
- 401 Unauthorised – Authentication failure
- 403 Forbidden – Authorisation error
- 404 Not Found – Resource not found
- 500 Internal Server Error – Unexpected failure

12. Define appropriate HTTP status codes for all possible API responses

Each endpoint maps to a specific set of status codes based on the action and context. For example:

- GET /v1/users/{id}: 200 OK, 404 Not Found
- POST /v1/users: 201 Created, 400 Bad Request
- DELETE /v1/projects/{id}: 204 No Content, 404 Not Found

13. Create a standardised error response format
All error responses follow this JSON structure:

```
{
  "error": "ResourceNotFound",
  "message": "Project with ID 123 not found.",
  "status": 404
}
```

This ensures consistency and easier debugging for API consumers.

14. Develop error categorisation (validation errors, business logic errors, system errors)
We categorised errors as:
- Validation Errors: Missing or malformed inputs (400 Bad Request)
- Business Logic Errors: Violations of domain rules (e.g., "Cannot donate to inactive project")
- System Errors: Server-side failures (500 Internal Server Error)

Each category helps in tracing and debugging issues efficiently.

15. Design informative error messages that guide API consumers
Error messages are written in plain language and provide actionable details:
- "The field 'email' must be a valid email address."
- "Cannot delete a project that has active donations."
- "Authentication token is missing or expired."

This minimises confusion and accelerates development.

16. API Versioning Strategy (Led by API Architect)
We selected URL-based versioning as our primary strategy:
- Example: /v1/users, /v1/projects
- This approach makes versioning explicit and is easy to manage in routing.

17. Select and justify a versioning approach (URL, header, parameter)
URL versioning was chosen due to its transparency and wide adoption. It simplifies client implementation and allows side-by-side support of multiple versions without requiring custom headers.

18. Document deprecation policy and backwards compatibility guidelines
We maintain backwards compatibility for each API version until an official deprecation notice is issued. Deprecated features:
- Will be marked clearly in the documentation
- Will remain accessible for at least 6 months post-deprecation

- Will include recommended alternatives

19. Create a change management framework for future API evolution
All breaking changes follow this process:
1. Announce deprecation via release notes
2. Provide migration documentation
3. Maintain the old version while rolling out the new one
4. Monitor adoption and gradually sunset older versions

Non-breaking changes (e.g., new fields) are added without affecting current consumers.

20. Design feature toggles or capability discovery mechanisms
We designed a GET /capabilities endpoint that returns a JSON payload describing available features:

```
{
  "features": {
    "projectSearch": true,
    "geoFilter": false
  },
  "apiVersion": "v1"
}
```

Phase 3: Documentation Excellence
Delivered by: Vestine Pendo

Overview

This documentation outlines the third phase of the JuaJobs API Design Project, with a focus on ensuring high-quality, comprehensive, and user-friendly API documentation. It builds upon the resource modelling and endpoint design established in Phases 1 and 2.

1. OpenAPI Specification (Narrative-Style Documentation)

A complete **OpenAPI 3.0** specification is included as part of this phase. This machine-readable Swagger file documents all endpoints, request/response structures, parameters, and authentication mechanisms.

1.1. Resources & Endpoints

Each of the following resources will have documented endpoints using RESTful conventions:

- Users: /v1/users/{id}
- Job Postings: /v1/jobs/{id}
- Profiles: /v1/profiles/{id}
- Applications: /v1/applications/{id}
- Reviews: /v1/reviews/{id}
- Payments: /v1/payments/{id}
- Messages: /v1/messages/{id}
- Products: /v1/products/{id}

Each endpoint includes:

- Operation summary and description
- Method (GET, POST, PUT, PATCH, DELETE)
- Parameters (query, path, body)
- Success and error responses
- Authentication requirements

1.2. Schemas & Attributes

Each resource has a documented schema covering:

- Field names
- Data types (e.g., string, integer, boolean)
- Required vs. optional

- System-generated vs. user-provided
- Validation rules (e.g., email format, min password length)

Example (for Job):

- title (string, required)
- description (string, required)
- location (string, required)
- status (enum: open, closed, completed)
- createdAt (timestamp, system-generated)

1.3. Request & Response Examples

For each operation, a clear example is provided:

- **Request Body:** key-value pairs expected in POST/PATCH/PUT
- **Success Response:** 200/201 status, structure of returned object

Error Response: 400/401/403/404 codes with consistent format:

```
 {
  "error": {
    "code": 400,
    "type": "ValidationError",
    "message": "Title is required",
    "field": "title"
  }
}
```

1.4. Security

- **Authentication:** Token-based via Authorisation: Bearer <token>
- **Sensitive endpoints** require authorisation based on role (client/worker/admin)

- **Error codes:** 401 Unauthorised, 403 Forbidden,404 Not Found.

2. API Style Guide

2.1. Naming Conventions

- **Endpoints:** lowercase, plural nouns (/users, /jobs)
- **Parameters:** camelCase (sortBy, pageNumber)
- **Relationships:** nested where appropriate (/users/{userId}/applications)

2.2. Formatting Standards

- **Dates:** ISO 8601 (e.g., 2025-05-25T10:00:00Z)
- **Currency:** decimal with currency code (e.g., 1500.00 RWF)
- **Language & Locale:** Optional query param ?lang=fr for localisation

2.3. Documentation Templates

All endpoint documentation will follow this structure:

- Summary
- Description
- Method & Endpoint
- Parameters
- Request Body Example
- Response Example
- Status Codes
- Authentication Required?

2.4. Guidelines for API Evolution

- **Versioning:** Use URI versioning (e.g., /v1/jobs)
- **Backwards Compatibility:** Maintain support for each version for 24 months
- **Deprecation:** Mark deprecated fields/endpoints with notes and return Deprecation headers
- **Future-Proofing:** Avoid tight coupling between resource formats and client logic

3. Developer Experience (DX) Documentation

3.1. Getting Started Guide

- Register at /v1/users
- Authenticate at /auth/login to receive a token
- Use the token in the Authorisation header for all requests
- Access sample sandbox data using /v1/jobs?demo=true

3.2. Authentication Flows

- **Sign Up → Login → Token Generation**
- **Token Expiration and Refresh Strategy**
- Example flow:
    - POST /auth/login → returns token
    - Use the token for /v1/jobs or /v1/applications

3.3. Interactive Use Case Examples

- **Client Journey:**
    1. POST /v1/users (create account)
    2. POST /v1/auth/login (get token)

3. POST /v1/jobs (create job)
4. GET /v1/applications (view all applications)
5. PATCH /v1/applications/{id} (view specific apllicants)

- **Worker Journey:**
    1. POST /v1/users
    2. POST /v1/auth/login
    3. POST /v1/workers/{id} (create profile)
    4. GET /v1/jobs (browse jobs)
    5. POST /v1/applications (create new job application)

3.4. Support for API Consumers

- FAQ section
- Error Message Reference
- Rate Limit Policy (e.g., 100 requests/minute)
- Sandbox Environment URL for safe testing

Phase 4: Security & Compliance Design (Led by Security Designer)
Authentication Design
Delivered by: Samuel Komaiya

Overview
To ensure secure and scalable access to the JuaJobs API, we will implement a token-based authentication system tailored for mobile-first use in Africa. Our solution balances industry-standard security protocols with real-world connectivity constraints experienced across the continent.

Supported Authentication Methods

1. OAuth 2.0: Industry-standard framework for delegated access.
2. JSON Web Tokens (JWT): Compact, stateless tokens for securely transmitting user claims.
3. API Keys: Used for internal services and low-sensitivity integrations (e.g., public listings API).

Token-Based Authentication Flow
Authorisation Code Flow with Refresh Tokens:

1. Client Registration
   API consumers (e.g., mobile apps) register to receive required credentials.

2. User Authorization
   Users are redirected to the authorisation server to grant consent to client applications.

3. Token Exchange
   After successful authorisation, the client exchanges the authorisation code for:

   An access token (expires in 1 hour), or after a long inactivity period.

   A refresh token (long-lived, securely stored)

Token Refresh
Clients use refresh tokens to renew access without requiring the user to log in again.

Why this matters for Africa:
Intermittent connectivity and high mobile usage make short-lived access tokens combined with refresh tokens the most secure and practical approa**ch.**

Security Requirements for API Consumers

Requirement	Description
TLS Enforcement	All API traffic must use HTTPS to prevent eavesdropping.
API Key Validation	Each client must include a valid API key (in headers or params).

Rate Limiting	Prevent brute-force and abuse using IP/user-based request limits.
Replay Protection	JWTs include timestamps and nonces to prevent replay attacks.

Authorization Framework
Role-Based Access Control (RBAC):

Role	Description	Access Level
Admin	Platform staff	Full access (CRUD on all resources)
Client	Service requester	Create/view job postings, submit reviews
Worker	Skilled service provider	Manage profile, view/apply to jobs

Permission Mapping
Access to endpoints will be tightly scoped per role, e.g.:

POST /jobs → Clients only

GET /applications/{username} → Workers only

Resource Ownership Model
Every job, review, and application is owned by a specific user ID. Only the owner or an admin can modify or delete it.

Delegated Permissions (Future Consideration)
We may support clients delegating limited access to assistants or team members for multi-user organisations.

Data Privacy & Regional Compliance
Personally Identifiable Information (PII) Handling

PII (names, phone numbers, payment details) will be stored securely with field-level encryption. Access to sensitive fields will be role-restricted and logged.

Consent Management

Users will explicitly consent to data collection during onboarding and can revoke consent from their account settings. Consent will be versioned and timestamped.

Data Minimization

Only essential fields will be collected during registration.
API responses will return minimal fields unless more data is explicitly requested (e.g., via the fields= parameter).

Compliance with Key Frameworks:

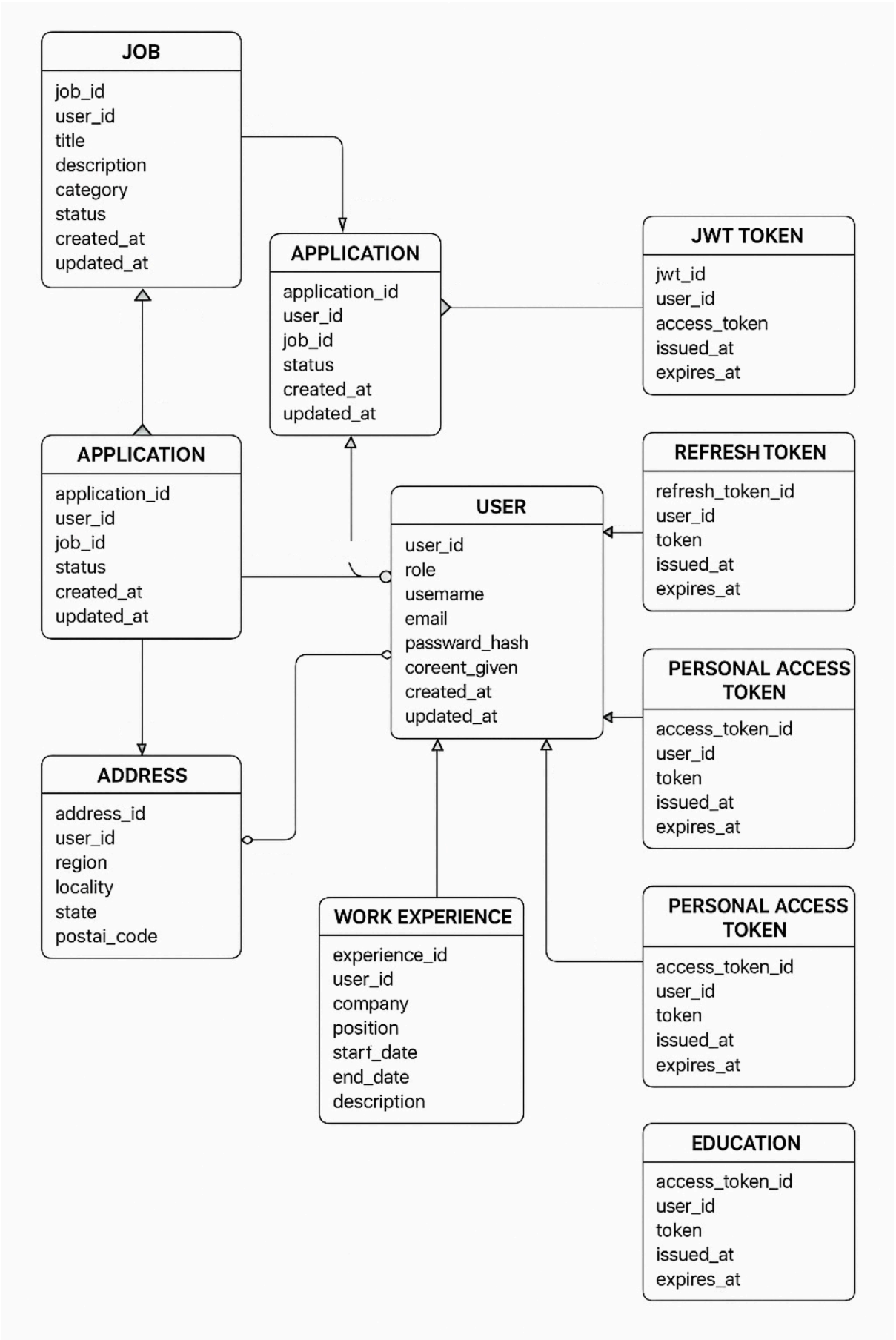
Law/Regulation	**Applicability to JuaJobs**
GDPR	Required for users in or interacting with the EU. Covers consent, right to erasure, and data portability.
CCPA/CPRA	Applies to any US-based interactions or clients from California. Includes opt-out and disclosure mandates
APPIA (African Personal Data Protection Act)	Provides a continent-wide standard under the African Union, including data localisation, consent-first principles, and government data oversight.

References

OAuth 2.0 Spec
JWT Spec
GDPR Overview
APPIA - AU Data Protection Act

This security and compliance design ensures that the JuaJobs API:

1. It is secure for users in both urban and rural areas
2. Complies with international and African data privacy regulations
3. Supports modern app authentication with refresh capabilities
4. Protects user roles and resource ownership through robust access control


JOB
job_id
user_id
title
description
category
status
created_at
updated_at
APPLICATION
application_id
user_id
job_id
status
created_at
updated_at
JWT TOKEN
jwt_id
user_id
access_token
issued_at
expires_at
APPLICATION
application_id
user_id
job_id
status
created_at
updated_at
REFRESH TOKEN
refresh_token_id
user_id
token
issued_at
expires_at
USER
user_id
role
usemame
email
passward_hash
coreent_given
created_at
updated_at
PERSONAL ACCESS TOKEN
access_token_id
user_id
token
issued_at
expires_at
ADDRESS
address_id
user_id
region
locality
state
postai_code
WORK EXPERIENCE
experience_id
user_id
company
position
start_date
end_date
description
PERSONAL ACCESS TOKEN
access_token_id
user_id
token
issued_at
expires_at
EDUCATION
access_token_id
user_id
token
issued_at
expires_at

ERD Summary

The JuaJobs ERD models the core entities and relationships that power the platform's gig economy functionality. At the centre of the design is the User entity, representing both clients and workers, with role-based behaviour determined logically through relationships rather than hardcoded fields. A user can create multiple JobPostings (as a client), and each job posting can receive multiple Applications from other users acting as workers. Workers are represented through a WorkerProfile, which has a one-to-one relationship with a user, capturing professional skills, experience, and availability. The Application entity links a job and a worker, capturing the application status and related communication.

To support worker-client interaction, the system includes a Review entity where users can review one another, modelled with self-referencing foreign keys (reviewerId and revieweeId) to the User entity. Similarly, the Message entity enables private messaging between users, also using sender and receiver foreign keys to the User. The Payment entity handles financial transactions, linking a payer and receiver (both users) to a job where applicable, enabling tracking of completed services. Finally, the Product entity allows workers to list services or offerings, forming a one-to-many relationship with the User entity. These entities collectively form a modular, scalable, and user-centred data model that mirrors real-world gig interactions while maintaining clean API endpoint alignment.