INDIAN INSTITUTE OF TECHNOLOGY INDORE

NAUKRI WALLAH

A Decentralized Job Marketplace

CS 218 Programmable and Interoperable Blockchains

PRESENTED BY: *BUG_WRITERS*
ANMOL JAIN
NIDARSANA M
NANDINI KUMARI
MITANSHU KUMAWAT
PRIYANSHU PATEL
TRIPTI ANAND

Contents

Vision

Connecting employers and freelancers without centralized intermediaries.

Key Features

01 **Escrowed Payments:** Employers lock funds in smart contracts ensuring payment security

02 **Transparent Applications:** Direct peer-to-peer freelancer applications

03 **Automated Workflows:** Smart contracts manage the entire job lifecycle

04 **Trustless Architecture:** No central authority - code-governed interactions

05 **User-Friendly:** Intuitive React interface with MetaMask integration

Core Benefits

Removing intermediaries from freelance marketplace.

For Employers

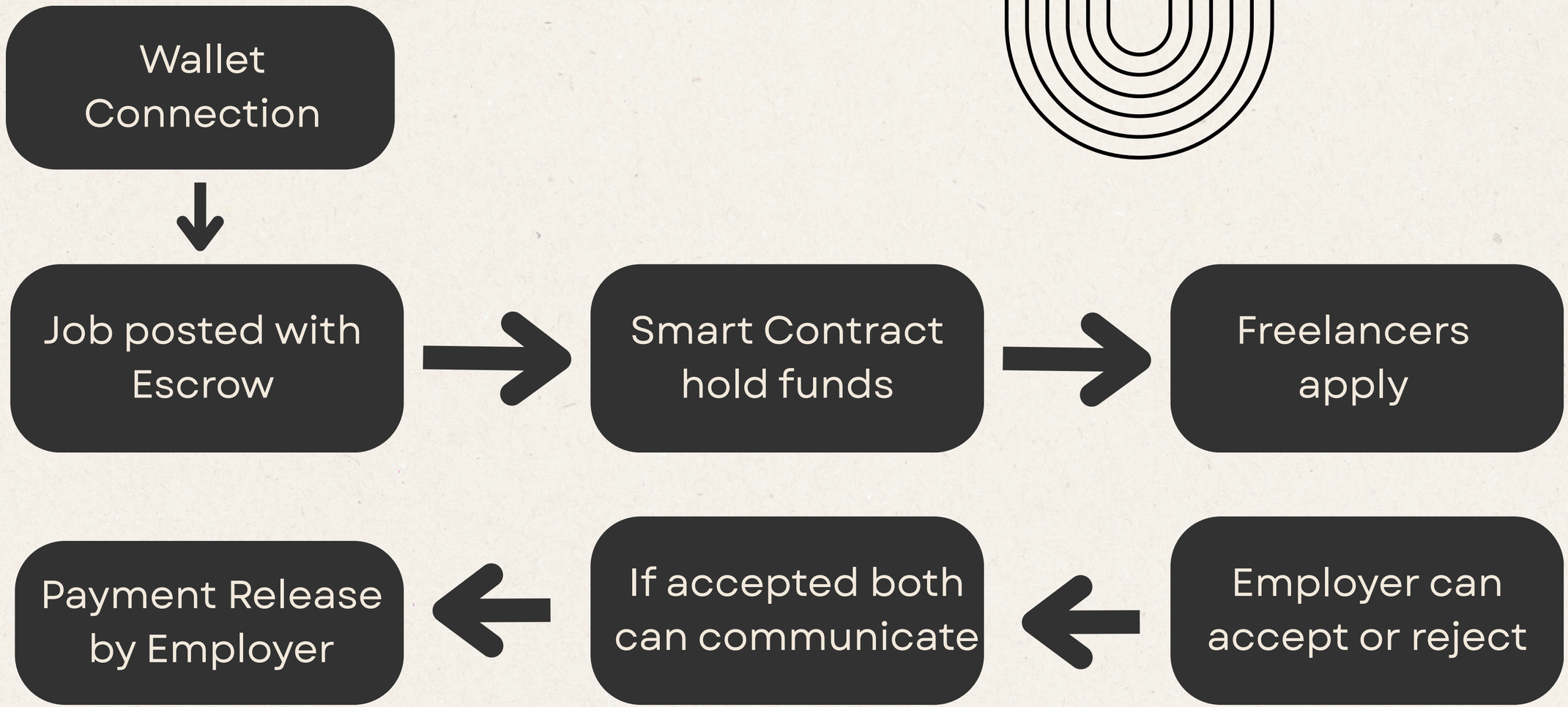
1. Guaranteed work quality before releasing funds.
2. Access to global talent pool.
3. Reduced fees compared to traditional platforms.

For Freelancers

1. Payment security through escrow contracts
2. Fair dispute resolution mechanism
3. Direct client communication

Live Demonstration

Workflow


Wallet Connection
Job posted with Escrow
Smart Contract hold funds
Freelancers apply
Employer can accept or reject
If accepted both can communicate
Payment Release by Employer

Tech Stacks

01 **Frontend:** React.js with Tailwind CSS

02 **Blockchain Integration:** Ethers.js

03 **Smart Contracts:** Solidity (Hardhat local testnet)

04 **Contract Libraries:** OpenZeppelin for security

05 **Backend Services:** Node.js for auxiliary functionality, CometChat API for real-time messaging

06 **Developer Tools:** MetaMask, GitHub

Project Structure

Smart Contract Layer

- JobBoard.sol - Core marketplace functionality
- OpenZeppelin integrations for security

Application Layer

- React components for different user flows
- Services for blockchain & chat integration
- Global state management

Service Layer

- Chat functionality via CometChat
- Blockchain transaction handling

Project Setup

01 git clone https://github.com/Anmoljain2005/NaukriWallah_By_BugWriters.git

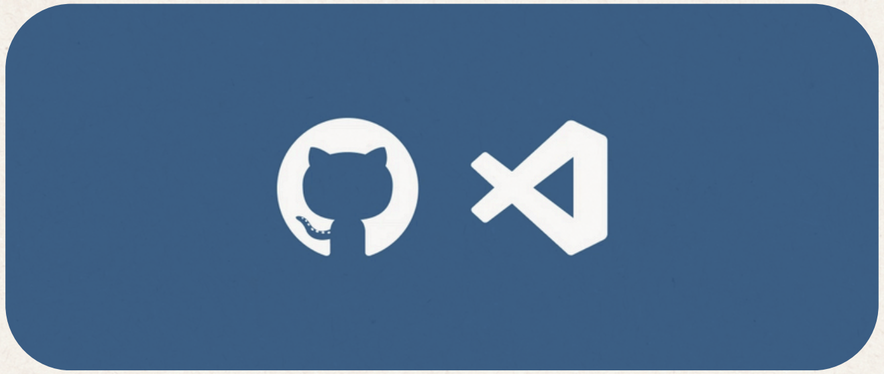
02 cd NaukriWallah_By_BugWriters

03 yarn install

04 yarn hardhat node

05 yarn hardhat run scripts/deploy.js

06 yarn start



Project Structure

NaukriWallah_By_BugWriters combines:

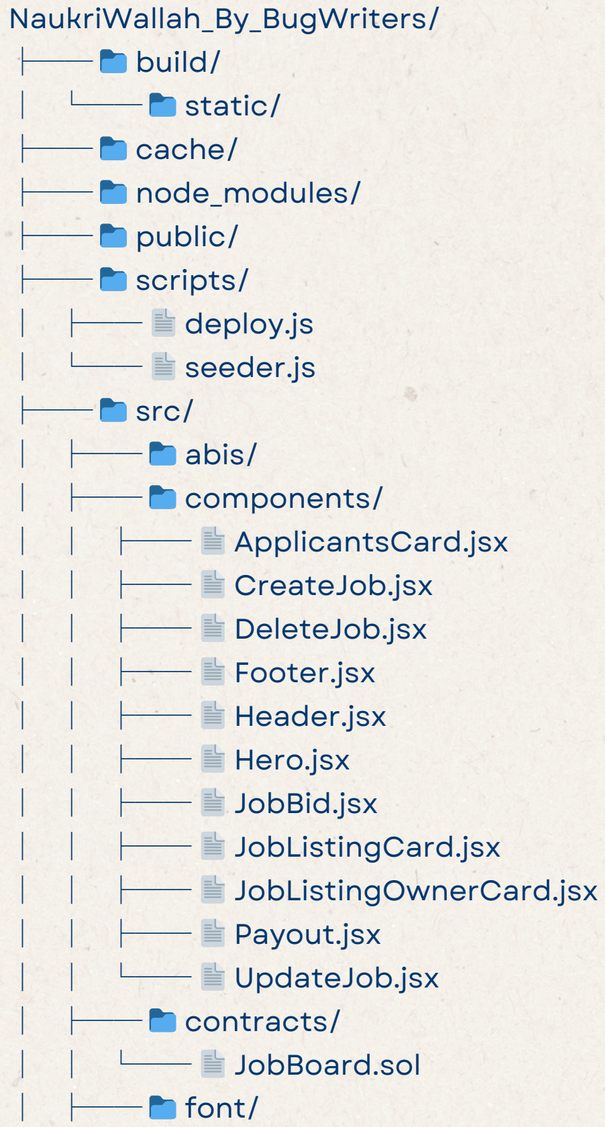
- A smart contract in Solidity
- A modern frontend in React
- Integration through Ethers
- Testing with Hardhat

Top-Level Folders

- **build/** :Contains the production-ready frontend after build
- **cache/**: Temporary files used by Hardhat to speed up compilation.
- **node_modules/**: Auto-generated folder that stores all project dependencies.
- **public/**: Holds static assets like images used in frontend rendering.

Frontend Architecture

- **Components/** – Reusable UI elements
  - Application flow: CreateJob, JobBid, Payout
  - Job handling: JobListingCard, JobListingOwnerCard, ApplicantsCard



- UI elements: Header, Footer, Hero
- Job management: UpdateJob, DeleteJob

- **Pages/** – Main application views
  - Home.jsx – Landing page
  - JobListing.jsx – Browse available jobs
  - MyJobs.jsx – Employer job management
  - etc...
- **App.jsx and index.jsx:** Main entry points for rendering the React application.

Blockchain Integration

- **Contracts/**
  - JobBoard.sol – Core smart contract handling job listings, bids, escrow
- **Services/**
  - blockchain.jsx – Web3 connection and contract interaction
  - chat.jsx – Messaging functionality
- **ABIs/**
  - Application Binary Interfaces for smart contract interaction

Project Infrastructure

- **Store/**
  - index.jsx – Store configuration
  - data.js – Global state definitions
- **Scripts/**
  - deploy.js: Automates the deployment of the smart contract.
  - seeder.js: Seeds initial data (optional for testing/demo purposes).
- **test/**
  - Contains unit tests for smart contract logic.
- **Configuration**
  - Environment variables and build settings
  - Public assets and static resources

Detailed Analysis

Overview

This is a decentralized job marketplace built on Ethereum that connects job posters with freelancers. The contract facilitates the entire workflow from job posting to payment, with built-in dispute resolution mechanisms.

Core Features

- ***Job Creation and Management***
  - Job posters can create listings with a title, description, and tags
  - Job poster calls ***addJobListing()*** with job details and ETH payment
  - Contract validates inputs and stores the job with a unique ID
  - Jobs must be funded upfront when created (ETH is locked in the contract)
  - Job posters maintain control using ***deleteJob()*** or ***updateJob()*** function

- ***Bidding System***
  - Freelancers call ***bidForJob()*** to express interest
  - Job owner can view all bids via ***getBidders()*** and review them
  - Creates a new BidStruct and adds it to jobBidders[id].

- Owner selects a bid with ***acceptBid()***, which assigns the freelancer
- Links the accepted freelancer to the job (jobListings[jobId].freelancer).
- Once assigned, the job is no longer listed as available

- ***Dispute Resolution***
  - Job owner can call ***dispute()*** to flag issues
  - Prevents payout while the dispute is active
  - Disputed jobs can be resolved via ***resolved()*** which clears dispute flag
  - Jobs can be revoked via ***revoke()*** which returns funds back to the owner and marks back the availability of job
- ***Payment Process***
  - Upon work completion, job owner calls ***payout()***
  - Contract takes the platform fee and transfers remaining funds to freelancer
  - Job is marked as paid out

Technical Components

- **Data Structures**
  - **JobStruct**: Stores complete job details including owner, freelancer, payment amount, and status
  - **FreelancerStruct**: Manages freelancer assignments to jobs
  - **BidStruct**: Tracks all bids placed on jobs

- **Access Control**
  - Uses **OpenZeppelin's** Ownable for platform administration
  - Custom modifiers like **onlyJobOwner** ensure only authorized addresses can perform certain actions
  - **Separate permissions** for job owners, freelancers, and platform administrators

- **Security Features**
  - **ReentrancyGuard** prevents reentrancy attacks during payments
  - **Multiple** state **validations** before critical operations
  - Safe transfer patterns with verification of success

Struct

```
struct JobStruct {
    uint id;
    address owner;
    address freelancer;
    string jobTitle;
    string description;
    string tags;
    uint prize;
    bool paidOut;
    uint timestamp;
    bool listed;
    bool disputed;
    address[] bidders;
}
```

```
struct FreelancerStruct {
    uint id;
    uint jId;
    address account;
    bool isAssigned;
}
```

```
struct BidStruct {
    uint id;
    uint jId;
    address account;
}
```

Gas Consumption

Action	Gas Used	Explanation
`dispute`	33426 gas	Filing a dispute costs a small amount because you're writing to blockchain.
`resolved`	33361 gas	Resolving a dispute similarly costs writing gas.
`addJobListing`	367329 gas	Big gas — you're creating a new job and transferring 0.3 ETH along with it.
`bidForJob`	186266 gas	Medium gas — you're adding a bid entry.
`acceptBid`	128415 gas (capped to 138446)	Accepting someone's bid, updating job status.
`payout`	80886 gas	Sending payout after job complete.
`getJob`	almost free (just eth_call)	Only reading from the blockchain (no state change).
Contract Deployment	3251214 gas	Huge because deploying a new smart contract costs a lot!

Smart Contract Design & Efficiency

1. Gas Optimization

- **Proper Data Types Usage**: The contract uses uint for job prizes and IDs, which is more gas-efficient than strings for numerical values.
- **Counters Implementation**: used **OpenZeppelin's Counters** for job IDs (_jobCounter), which is gas-efficient for incremental IDs.
- **Efficient Storage**: minimizes storage operations by **structuring data** appropriately:
  - Uses **mappings** (jobListingExists, hasPlacedBid) for **quick lookups** rather than iterating through arrays
  - Stores only essential job information **on-chain**
- **Avoiding Redundant Storage:** For example, in the bidForJob function, we **update** the hasPlacedBid mapping just **once** rather than checking and updating repeatedly.

Smart Contract Design & Efficiency

2. Security & Access Control

- **Escrow Logic**
  - **Fund Locking**: Funds are **locked** in the **contract** upon job creation via the addJobListing() function, which requires a non-zero ETH value to be sent.
  - **Controlled Release**: The payout() function ensures that funds are only released when **conditions** are **met** (job is assigned, not disputed, and not already paid out).
  - **Payment Splitting:** The contract automatically **calculates platform fees** and transfers the remaining funds to the freelancer, handling the economic model transparently.
  - **Dispute Handling:** dispute resolution mechanism **prevents payouts** while disputes are active.

Smart Contract Design & Efficiency

3. Privacy & Data Minimization

- **On-Chain**
  - All **core job data**, including descriptions and prices
  - Bid history and freelancer assignments
  - Payment and dispute status
- **No Personal Data On-Chain**
  - Uses Ethereum **addresses** for identity rather than personal information
  - Stores only essential job-related information
  - Keeps descriptions minimal and focused on requirements
- **Off-Chain Storage**
  - Communication handled off-chain through **CometChat**
  - **UI state management** and address based filtering.

Challenges & Solutions

Problem: Unauthorized Payment Releases

- Anyone could call the payout() function and steal escrowed ETH.

- **Solution**
  - Used a modifier ***onlyJobOwner(jobId)*** to ensure only the employer **who posted** the job can **release payment**.
  - Added ***nonReentrant*** to prevent reentrancy attacks.

Problem: One Freelancer Applying Multiple Times

- Freelancers might **spam** the **same job** by applying repeatedly.

- Solution:
  - Used a mapping ***hasPlacedBid[jobId][freelancer]*** to track bids.
  - Prevents double-bidding with a ***require(...)*** check.

Problem: Unauthorized Payment Releases

- What if a freelancer is hired but doesn't complete the job?

- Solution:
  - Implemented ***dispute()*** function (employer raises it).
  - Admin (***onlyOwner***) can then:
    - Use ***revoke()*** to cancel the job and refund the employer.
    - Use ***resolved()*** to reset dispute status.

Problem: Funds Must Be Escrowed Securely Until Job Is Done

- ETH must not be paid upfront or withdrawn by mistake.

- Solution:
  - ETH is escrowed in the contract on job creation via ***msg.value***.
  - Stored internally as ***jobListing.prize*** and only released through ***payout()*** after checks.

Future Aspects – Scope for Improvement

- **Decentralize Dispute Resolution**
  - Current Limitation: Only the contract owner can resolve disputes.
  - Future Solution: Use a DAO or community voting mechanism for decentralized, trustless dispute handling.

- **Multi-signature Wallet Support**
  - Current Limitation: Jobs can only be posted and managed by individual accounts.
  - Future Solution: Implement team-based job postings where multiple stakeholders must approve key actions like hiring freelancers and releasing payments, enabling organizational hiring with proper governance controls.

- **Implement a Reputation System**
  - Current Limitation: No feedback mechanism for users.
  - Future Solution: Add a rating and review system to track freelancer/employer credibility across jobs.

Deliverable Checklist

1. **Smart Contract** ✅
   - postJob
     - Employers can list jobs with title, description, and budget.
   - applyForJob
     - Freelancers can apply (employer must escrow funds first).
   - releasePayment
     - Payments are released correctly upon completion.
   - Escrow Logic
     - Funds are locked until job completion.
   - Events
     - Emits JobPosted, JobApplied, PaymentReleased.

2. **Testing** ✅
   - postJob tests
     - Employers can post jobs.
   - applyForJob tests
     - Freelancers can apply.
   - releasePayment tests
     - Payments are released only when conditions are met.

Deliverable Checklist

3. **Optimization & Gas** ✅
   - Uses uint256 for ETH
     - Avoids expensive types like string for budgets.
   - Minimal on-chain data
     - No unnecessary data .
   - Efficient functions
     - Avoids loops in critical functions.

4. **Privacy** ✅
   - No personal data
     - Only stores job details, not employer/freelancer identities.
   - Secure payments
     - Escrow logic prevents fund loss.

5. **Frontend** ✅
   - Wallet connection
     - Uses ethers.js to connect to MetaMask.
   - Job listing
     - Displays available jobs dynamically.
   - Application/payment
     - Allows applying and payment release.
   - Error handling
     - Shows transaction status (success/fail).

Bringing together the power of blockchain, modern web development, and real-time communication we tried to create a platform that empowers users and facilitates collaboration in a decentralized manner.

The End