

gig.io

Design Document

Geoff Hackett, Sam Fellers, Rishabh Saxena, Andrew Houvenagle, Austin Wirth

Index

● Purpose	1
○ Requirements	
● Design Outline	2
● Design Issues	3
○ Functional Issues	
○ Non-Functional Issues	
● Design Details	5
○ Database Schema Mockup	
○ Server API	
○ Frontend State Diagram	
○ Sequence Diagrams	
○ UI Mockups	

Purpose

Many people have menial jobs that they need done and lack the time or ability to do them. Our application aims to solve this problem by providing an environment for users to find cheap, reliable help for completing daily tasks in an efficient manner. Gig.io allows users to post certain tasks they need completed and other users can accept these tasks for an agreed upon pay. A key feature of the application will be the ability for users to bid on the wage they are willing to accept for said task. In addition, a review system will also be available to make sure that users receive help from trustworthy and genuine people.

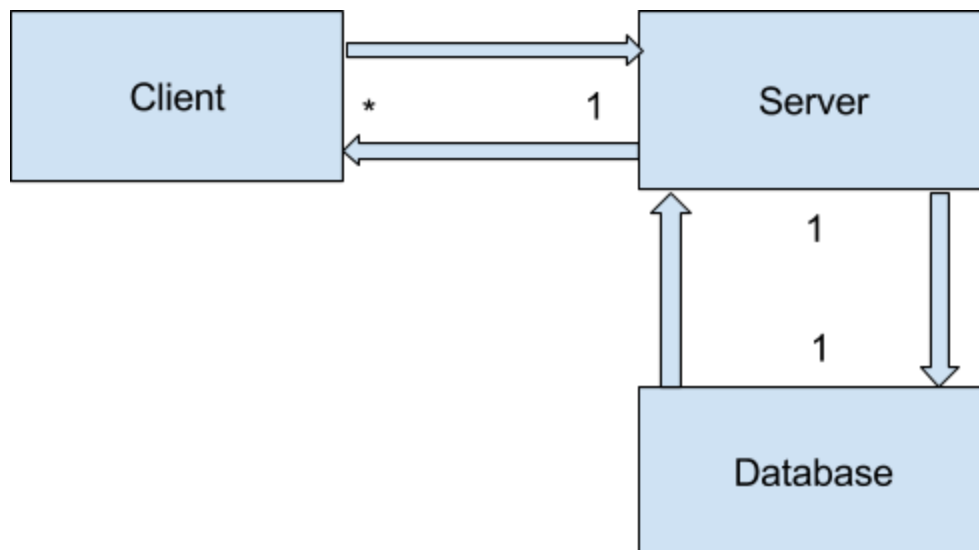
Functional Requirements:

1. User Profile:
 - a. Users can create an account and log in.
 - b. Users can view and change their profile information.
2. Posting and Browsing System:
 - a. Users will have the ability to create a post stating the details of the task they need completed. This may include a description of the task, the profile of the user who created the post, etc.
 - i. This type of user will be known as a *Poster*.
 - b. Users who are looking to bid on posts will be able to browse posts by distance.
 - c. After the bid for a post has been won, the post will not be viewable to users other than the poster and the bidder.
3. Bidding System:
 - a. Once the post is created, a bid would open up for the task and other users could bid on how much they are willing to be paid for that task.
 - b. Each bid will include the proposed price and a link to the bidder's user profile.
 - c. When the Poster closes the bid after a certain time frame, he/she will have the option to choose which user they want to complete the job based on their proposed price, rating, and other profile details.
4. Payment System:
 - a. Users will be able to pay another user for completing a task. The money will be taken from the user who posted the job once the amount is agreed upon, but will not be paid to the user who bid on the job until they complete the task. This prevents users from either not paying people who complete jobs for them or being paid before completing a job.
 - b. Users will be able to use secure protocol when making transactions through the gig.io website.
5. Rating System:
 - a. Users will have separate ratings for hosting and completing jobs. They will be able to view both of these ratings on their profile page.
 - b. After a user completes a job for another user, they can rate that user on how easy they were to work with. The user who posted the job can also rate the user who completed the job based on how well they completed the task.
6. Administration System
 - a. Administrators will be able to ban or manage users within the system.

- b. Jobs with no activity for 30 days will be automatically deleted to minimize the burden of user and job management on administrators.

Design Outline

1. Web Client:
 - a. The web client will represent the front-end of our application and will behave as the interface to our system internals.
 - b. The web client will communicate with the server using HTTP requests in JSON format.
 - c. All information received from the server will be formatted and presented on the web client in an elegant form.
2. Server:
 - a. The server will provide easy access to store and retrieve information from the database.
 - b. The server will interface with an external payments service.
 - c. The server will validate all requests prior to processing them.
 - d. The server will process requests from clients.
3. Database:
 - a. The database will store user and job post information.
 - b. Database Tables:
 - i. Ratings
 - ii. Users
 - iii. Postings
 - iv. Notifications
 - v. Bid
 - vi. Transactions



Design Issues

Functional Issues

1. Should users have profiles?

Option 1: Gig.io will only allow posting of job listings, all other interactions will be handled by users

Option 2: Allow users to build a profile and allow users to briefly introduce themselves and their relevant experience or qualifications

Decision: Allowing user profiles makes for a more personal experience and will encourage users to return to the app. Additionally it allows feedback and ratings to be attached to a user so that other users may evaluate them before choosing to do business with them.

2. How should users receive notifications?

Option 1: Via text message

Option 2: Via email

Option 3: On the gig.io website

Option 4: Some combination of the other options

Decision: We have decided send notifications via email and to make notifications available on the gig.io website. That way, users will be able to see notifications whether or not they are browsing gig.io.

3. How will users who post tasks be differentiated from users who do tasks?

Option 1: Separate types of accounts for the two types of users

Option 2: One type of account with two user ratings

Option 3: One type of account with one aggregate user rating

Decision: We have decided to have one type of account with separate user ratings for posters and task-doers. Having one type of account simplifies development and the user experience, while having two kinds of ratings will increase user confidence in other users.

4. How long should completed job pages be stored?

Option 1: Archive them forever

Option 2: Deleted immediately after a bid is accepted.

Option 3: Allow them to remain for 30 days after job completion

Option 4: Delete pages after both users are satisfied with job completion and payment

Decision: To conserve storage space, old jobs will be deleted immediately after payment.

Non-functional Issues

1. How will we decide whether a job has been completely adequately?

Option 1: Leave all disputes up to the administrator's discretion

Option 2: Allow users to decide for themselves

Decision: Users will be required to present measurable goals before the job begins, but ultimately administrators will have the final say if a job was completely adequately for payment.

2. What platform will our user client initially be developed for?

Option 1: Web Browser

Option 2: Windows Executable

Option 3: Android App

Option 4: iOS App

Decision: We will design gig.io as a website rather than an executable or an application for a specific platform since a website will allow the widest range of access for customers with minimal effort required for cross platform support. This allows us to start with a universal platform and expand into specific markets further on in development.

3. What hosting will we use for our web client?

Option 1: Purdue's student hosting

Option 2: Github hosting

Option 3: Firebase

Decision: Github's web hosting, since it is free and convenient and allows us to have a reasonable URL.

4. Which technology will we use for the development of the backend?

Option 1: Node.js

Option 2: Django

Option 3: PHP

Decision: Node.js, as it provides the necessary functionality and is the option that the members of the team are most familiar with.

5. What kind of database will we use?

Option 1: A relational database such as MySQL

Option 2: Firebase

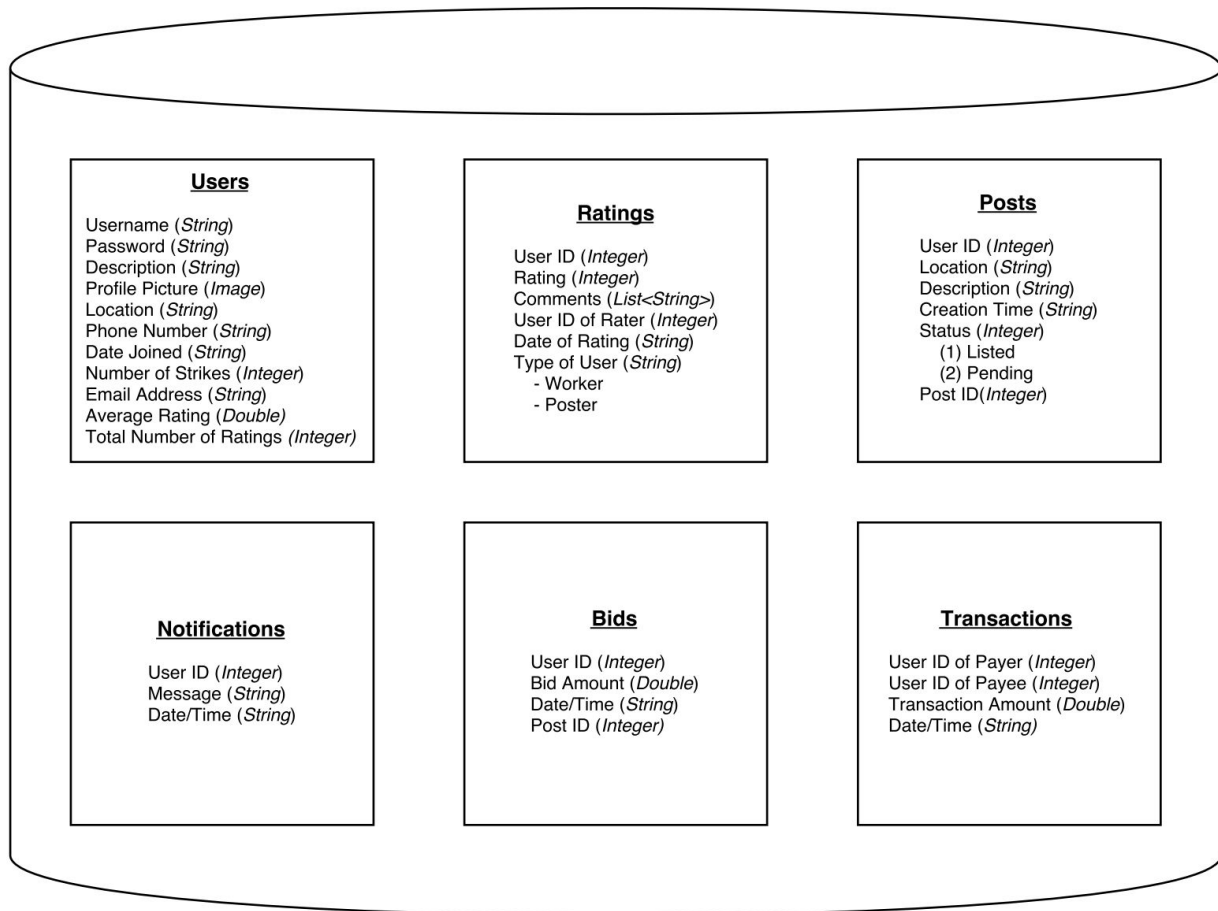
Option 3: A noSQL database such as MongoDB

Decision: We have decided to use a relational database due to ease-of-use and developer familiarity.

Design Details

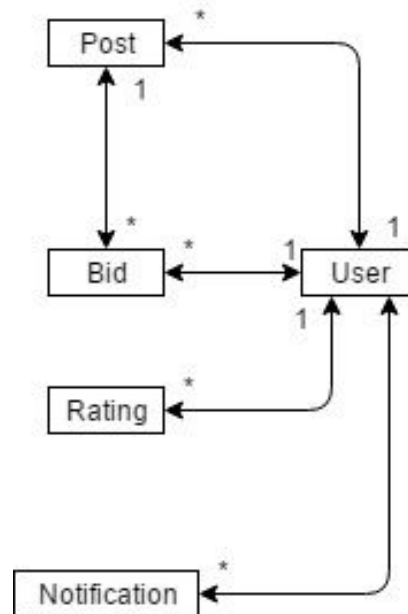
Database Schema Mockup

The database will consist of 6 tables: Users, Ratings, Posts, Notifications, Bids, and Transactions. The diagram below illustrates a mockup of the database schema and the different fields within each table along with the type of each data field.



Description of Database Tables

The mockup above shows the schema for the database and details the various fields that belong to a table. Although the schema may change through development, these tables should contain most of the relevant information needed to meet a majority of the functional requirements. The following section provides a more in-depth description of the primary function of each table in the database:



1. Users
 - a. This table contains all user information, such as email address, username, password, etc.
 - b. Every user who registers is represented by this class and is given no special privileges over other users.
2. Ratings
 - a. This table contains information regarding the user's average rating based on their performance after completing a task and for reliably paying another worker and abiding by the rules and regulations of being a payer.
 - b. Using the "User ID" field, each rating is correlated with a specific user and can be used to update that user's average rating as well.
 - c. In addition, users also have the option to send comments to explain the reasoning behind their rating.
3. Posts
 - a. This class represents a new task that a user posts.

- b. Each post contains a description of the task, where the task needs to be completed, the profile of the user who posted, etc.
 - c. There are possible two states that a post can be in...
 - i. *Listed*: When a task has been posted and is open for bidding.
 - ii. *Pending*: When the bid has been closed, but the task is still incomplete or is in the process of being completed.
- 4. Notifications
 - a. Each user will receive notifications noting any activity, such as whether they win a bid, whether a task they posted has been completed, whether they have been reported by another user, etc.
- 5. Bids
 - a. This is the class that represents the bid that a user puts in for a given task.
 - b. Each bid refers to a post using the "Post ID" field.
 - c. It also contains a link to the bidder's profile page, the bid amount, and when the bid was made.
- 6. Transactions
 - a. This class is used to keep track of any payments/transactions made between users.
 - b. It contains links to the payer and payee's profile page, the transaction amount, and the date and time of the transaction.

ServerAPI

Our server will use a RESTful api written in Node.js to facilitate communication between the clients and the database. We will use JSON to store the information.

For each endpoint, state will have three possible return values, giving general error information:

0 = success

1 = invalid input

2 = other error

CreateAccount:

Accepts

<String> Username
<String> Email
<String> Password

Returns

<Integer> State
<Integer> UserID

Login:

Accepts

<String> Username
<String> Password

Returns

<Integer> State
<Integer> UserID

EditProfile:**Accepts**

<String> Username
<String> Password
<String> Email
<String> Description
<Image> ProfileImage
<Location> Location
<String> PhoneNumber

Returns

<Integer> State

CreatePost:**Accepts**

<Location> Location,
<String> Description

Returns

<Integer> State,
<Integer> PostID

DeletePost:**Accepts**

<Integer> PostID

Returns

<Integer> State

GetPostsByLocation:**Accepts**

<Location> Location
<Integer> Radius

Returns

<Integer> State
List<Post> Posts

GetPostsByUser:**Accepts**

<Integer> UserID

Returns

<Integer> State
List<Post> Posts

Bid:**Accepts**

<Integer> PostID
 <Integer> UserID
 <Double> Amount

Returns

<Integer> State
 <Integer> BidID

GetBids:**Accepts**

<Integer> PostID

Returns

<Integer> State
 List<Bid> Bids

GetUser:**Accepts**

<Integer> UserID

Returns

<Integer> State
 <String> Username
 <String> Description
 <Image> ProfileImage
 <Location> Location
 <String> PhoneNumber
 <Date> DateJoined
 <Double> PosterRating
 <Integer> NumberOfPosterRatings
 <Double> CompletionRating
 <Integer> NumberOfCompletionRatings

GetNotifications:**Accepts**

<Integer> UserID

Returns

<Integer> State
 List<Notification> Notifications

Pay:**Accepts**

<Integer> UserIDFrom
 <Integer> UserIDTo
 <Double> Amount

Returns

<Integer> State

ReportUser:**Accepts**

<Integer> UserID

Returns

<Integer> State

GetRatings:**Accepts**

<Integer> UserID

Returns<Integer> State
List<Rating> Ratings**AddRating:****Accepts**<Integer> UserID
<Integer> Type
<Integer> Rating
<String> Comment**Returns**

<Integer> State

ClosePost:**Accepts**<Integer> PostID
<Integer> BidderUserID**Returns**

<Integer> State

BanUser:**Accepts**

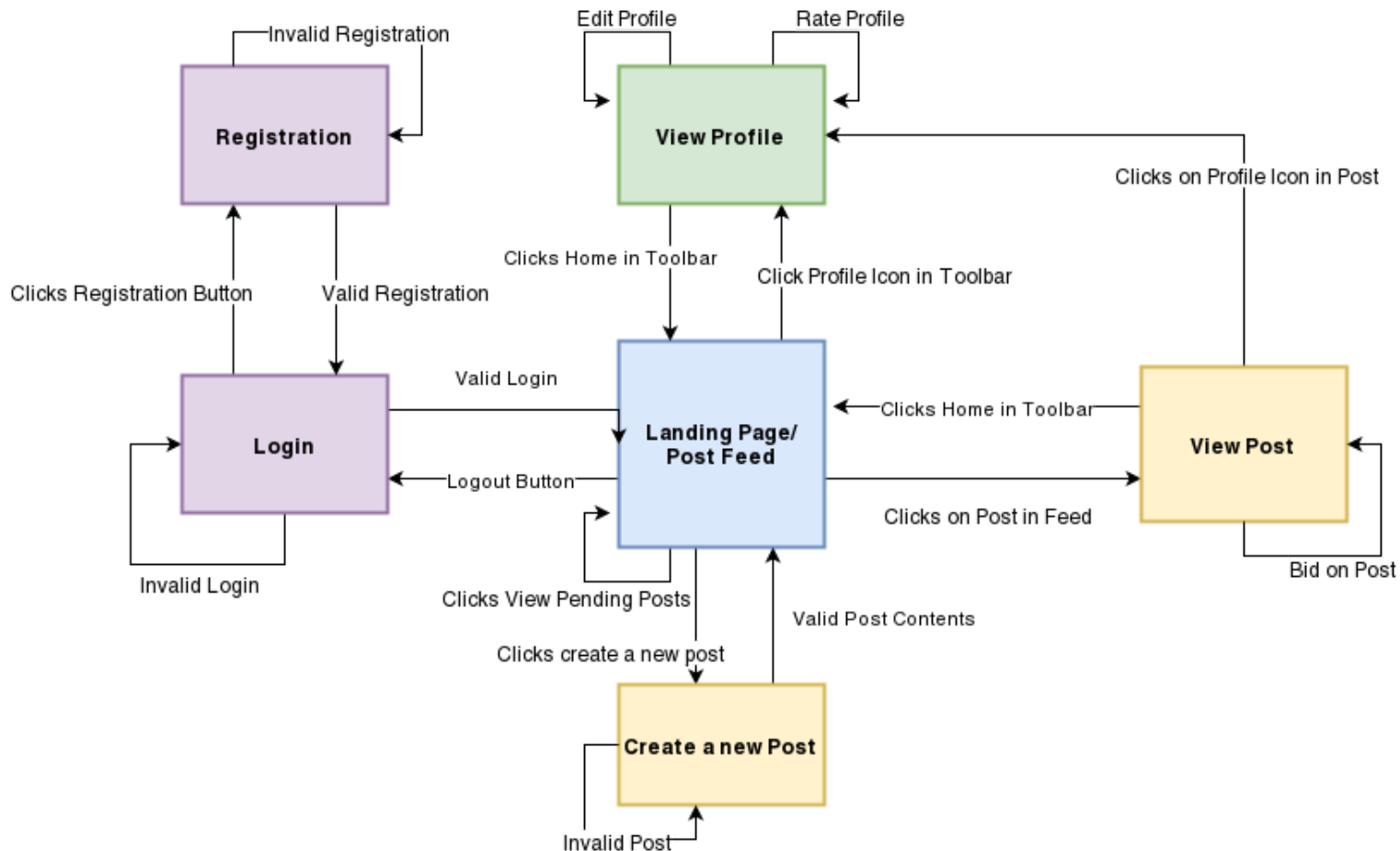
<Integer> UserID

Returns

<Integer> State

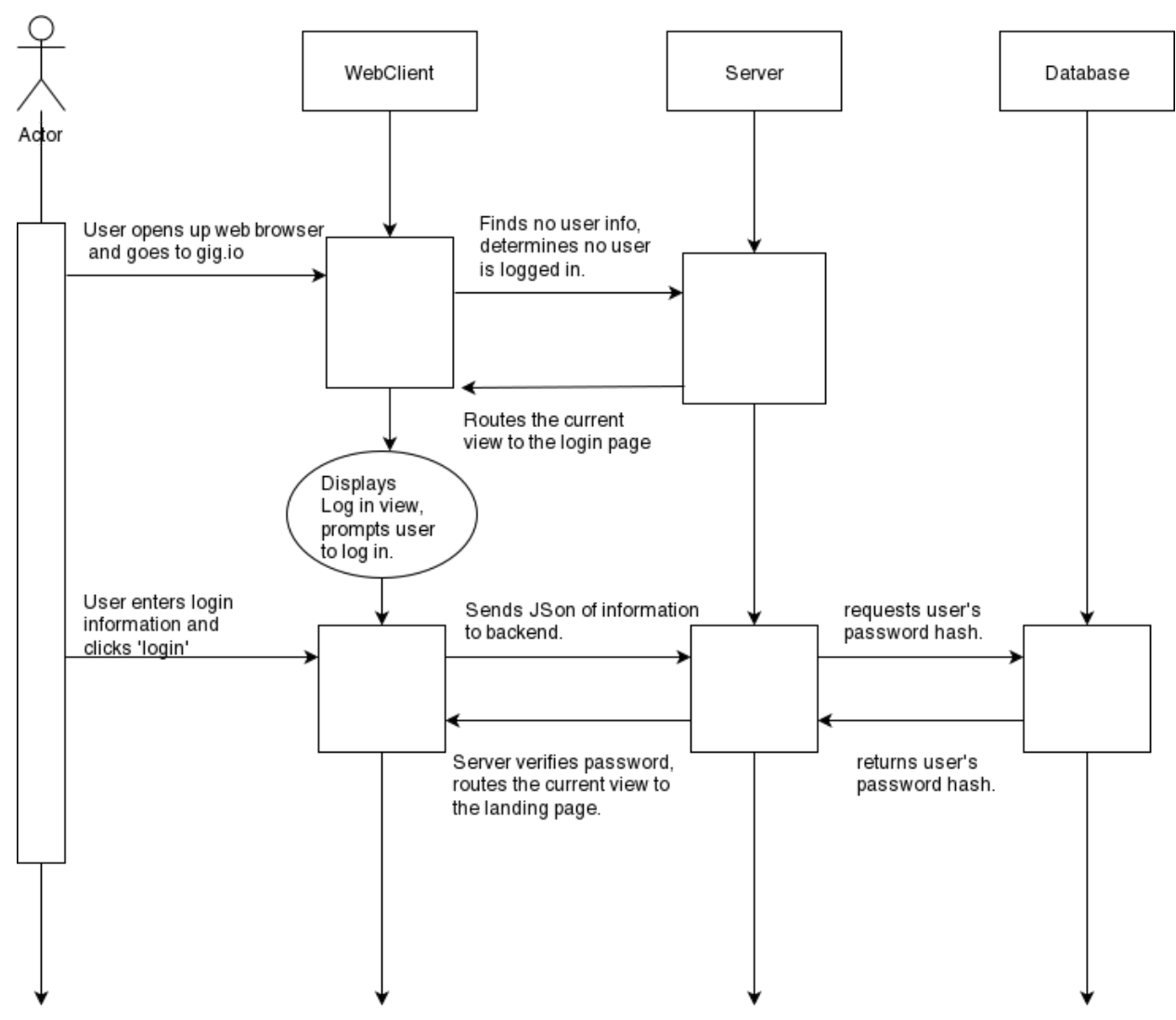
State Diagrams

Below is a map of our frontend schema. A few views are subject to change depending on how we structure our toolbar, however this describes most navigation between pages. The blue square represents home, the default landing page, yellow deal specifically with single-post displays, purple is pre-login, and green deals with displaying and editing profiles.

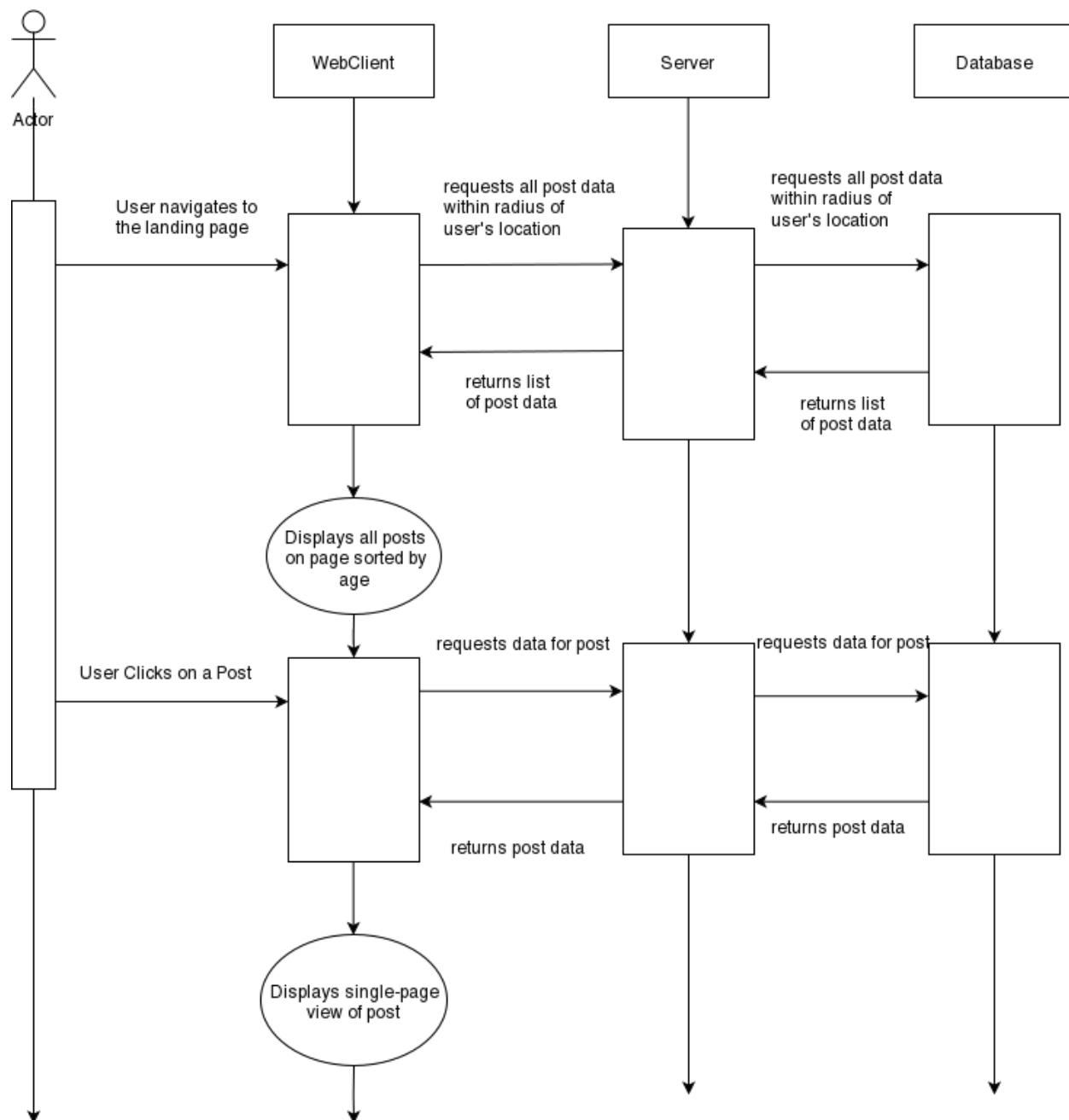


Sequence Diagrams

User Login



Landing Page/Post-View

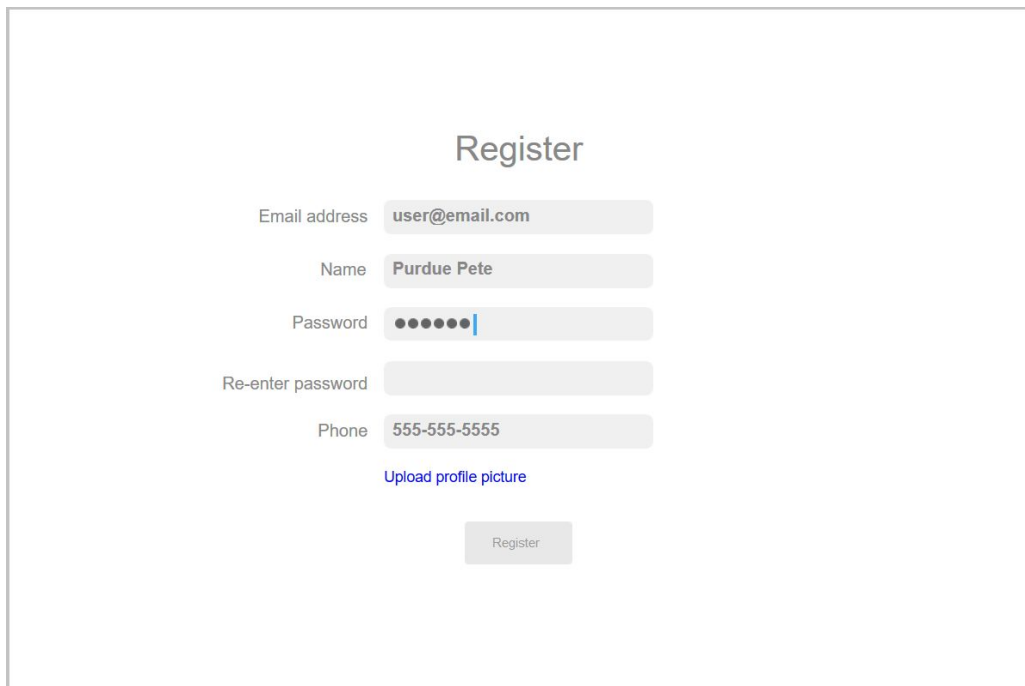


UI Mockups





A login page mockup for gig.io. The page features the 'gig.io' logo in a large, grey, sans-serif font at the top center. Below the logo, there are two input fields: 'Email' with the placeholder text 'user@email.com' and 'Password' with a masked password of seven dots. Below these fields are two buttons: 'Sign in' and 'Register'.

Home page if user is not logged in.




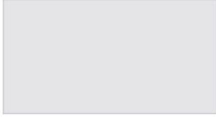
A registration page mockup for gig.io. The page features the title 'Register' in a large, grey, sans-serif font at the top center. Below the title, there are five input fields: 'Email address' with the placeholder text 'user@email.com', 'Name' with the placeholder text 'Purdue Pete', 'Password' with a masked password of seven dots, 'Re-enter password' (empty), and 'Phone' with the placeholder text '555-555-5555'. Below these fields is a link 'Upload profile picture' in blue text. At the bottom center is a 'Register' button.

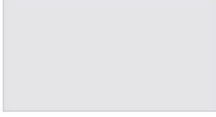
Registration page for new users.

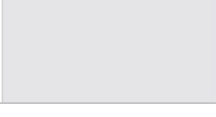
Pending


Pete

Location: Chicago ▼


Need lawn mowed
Current bid: \$8.30




Task name
Current bid: \$x.xx


Task name
Current bid: \$x.xx


Task name
Current bid: \$x.xx

My account
Sign out

Home page if user is logged in. The “pending” button on the toolbar will take the user to a similarly-formatted page containing pending tasks the user is involved in.

Pending


Pete

Profile


Email address: user@email.com

Name: Purdue Pete

Password: ●●●●●●

Re-enter password:

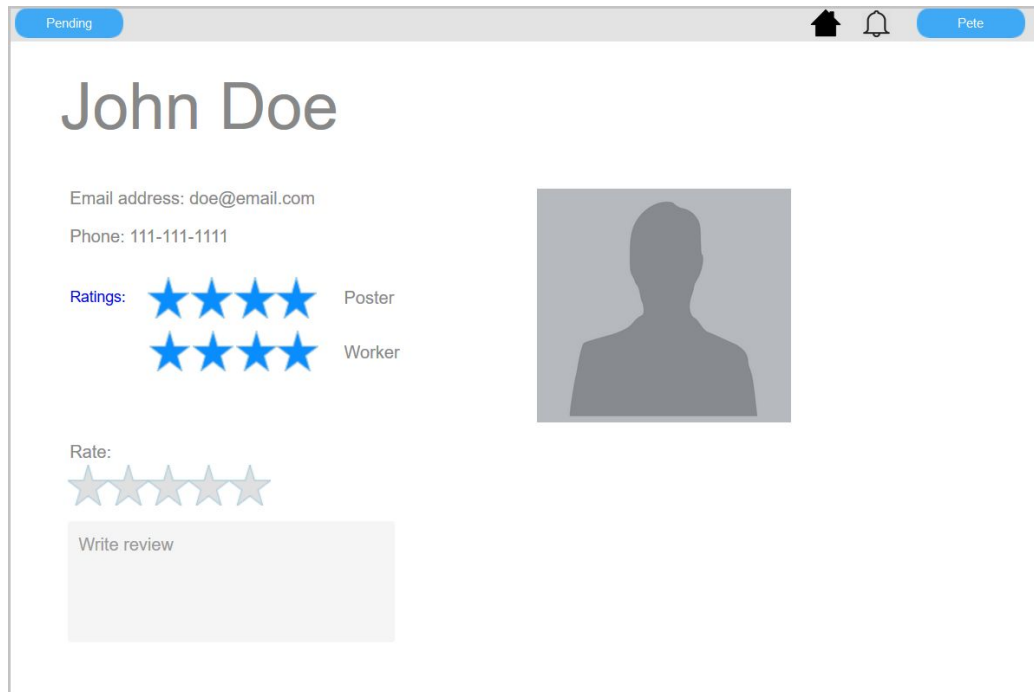
Phone: 555-555-5555


Update profile picture

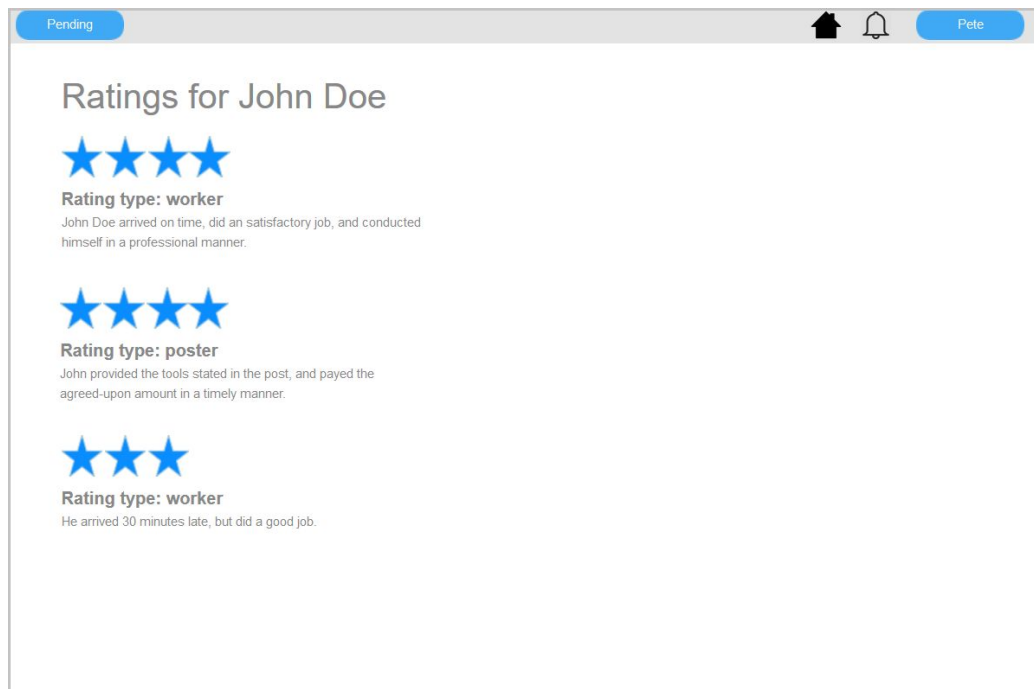
Your ratings: ★★★★★ Poster

★★★★★ Worker

Page where user can edit their profile and view their ratings.





Page where user can view another user's profile and rate them. The option to rate the other user will only appear if one of the users has accepted a bid from the other user.




Page with a user's ratings.

Pending



Pete

Need lawn mowed



I'm currently unable to mow my lawn. I am able to provide a mower, unless you want to bring your own.

Current bid: \$11.90

Place bid:

\$11.50

Bid

Poster: John Doe

Poster rating: ★★★★★

Page where a user can view and bid on a post.

Pending



Pete

Create Post

[Upload Images](#)

Enter description:

Starting bid:

Post-creation page.