MiniBid

Setup, Installation and Project Structure

Project Structure

```
MiniBid
|
└─── models                  #database schemas
└─── routes                  #api routes
└─── validations             #validating user input
└─── verifications           #authenticating users
└─── app.js                  #main app file
└─── .env                    #environment variables
└─── node_modules            #dependencies
└─── package-lock.json       #npm automatically generated document
└─── package.json            #metadata and npm packagage list
```

Node.js Libraries Used

MiniBid uses:

- express, for developing the server.
- nodemon, to aid development.
- mongoose, for database communication.
- body-parser, for reading requests.
- dotenv, for reading environment variables.
- joi, for enforcing validation.
- bcryptjs, for password hashing.
- jsonwebtoken, for enforcing authentication.

Setup and Installation

First a new MongoDB collection need to be created and deployed. The collection's connection link string should be retrieved. This link will be used to connect the MiniBid server to the new mongoDB collection.

Next there should be a `.env` file in MiniBid's root directory. The `.env` should have a variable `DB_CONNECTOR` with the MongoDB link as its value. There should also be a variable `TOKEN_SECRET` set to a secret value, MiniBid will use this value when authenticating user tokens.

To install and start MiniBid:

1. Navigate to project folder and install dependencies with:

```
$ npm install
```

> This command will use the `package.json` file to install all of MiniBid's dependencies.

2. Start MiniBid with:

```
$ npm start
```

> This will start MiniBid's server. Now MiniBid should be running on localhost port 3000. Clients can now send requests to MiniBid's API endpoints.

Enforcing authentication/verification functionalities

MiniBid only allows authenticated users to access the auctioning API.

This is done using the ‘jsonwebtoken’ library and the oAuth v2 protocol.

When a user logs in a JSON web token is created with the user’s ID and digitally signed with a `TOKEN_SECRET`, the result is given to the user as an `auth_token`.

When the user makes an API call they include their `auth_token` in the request header.

This is how users prove they are specific, registered user’s. As unique user IDs should produce unique `auth_tokens`.

The `auth_token` taken from the request header is decrypted using the `TOKEN_SECRET` , returning user’s ID if the decryption is valid. If the decryption is invalid, then access is denied.

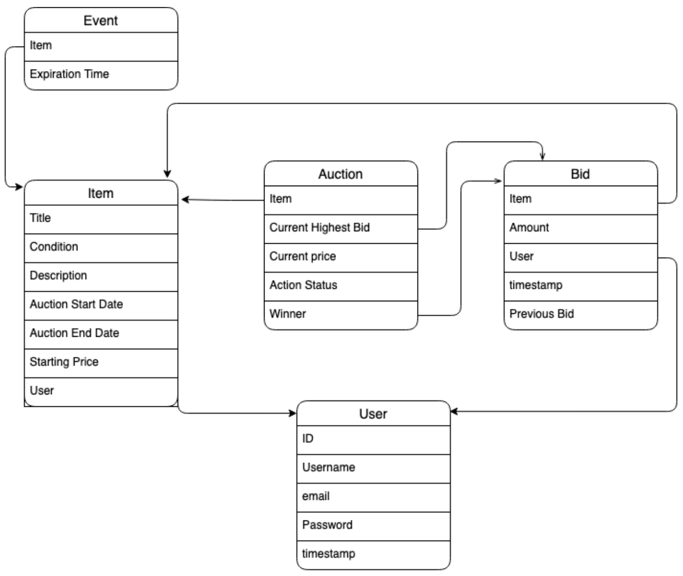
Creating and verifying these tokens allows MiniBid to control who is able to access the auctioning API.

Development of the MiniBid RESTful API

Brief Description of MiniBid’s Database Models

MiniBid uses five database models: `User`, `Item`, `Auction`, `Bid` and `Event`.

- `User` defines a MiniBid user. Users have a username, an email and a password (passwords are stored as hashes of actual passwords).
- `Item` defines an item, created and owned by a `User`. When a user creates an item they provide it with a title, the item’s condition (‘new’ or ‘used’), a description, an initial starting price and an expiration date. When an item expires the ‘item status’ field will indicate whether or not the item sold.
- `Auction` defines an auction on an `Item`. `Auctions` contain the item they are auctioning, the current price of the item, the current highest bid and, once they expire, the winner (if there is one). Auctions do not contain information regarding the time left to complete. This is because it would be difficult to achieve a high degree of accuracy when the time remaining is calculated server-side and then sent back to a client in a http response. A more responsive auction could be achieved with a stateful connection between the client and the server, but MiniBid should have a RESTful API and that would be in violation of the stateless property RESTful software should have. In this case, it makes more sense for a client to obtain an Item’s expiration date via MiniBid’s API and then calculate the Auction’s time remaining client-side.
- `Bid` defines a bid on an `Item` in an `Auction`. Bids hold information on the bidding amount, the bidding `User` and the `Bid` they out bid.
- `Events` are used by MiniBid to track the expiration of items. An `Event` contains a reference to an `Item` and an experation time.



MiniBid Application Logic Overview

MiniBid was developed to meet these goals:

> 1. Users should be able to register a unique identity.

All users are assigned a unique user id when they are added to the 'users' database collection. A user proves their identity by logging in with a password which was set when the user registered their account. The password is stored as a hash in order to not expose user passwords. When a user logs in the password is checked against the stored hash. The user then recieves an `auth_token` which is used to uniquely indentify them on all future API calls.

> 2. Authorised users should be able to post items for auction with a starting price and an end date. The item should not be sold after the end date and should not be sold for less than the starting price.

When a user posts an item, the user input is validated. The starting price is validated as a positive number with a precision of two decimal places; In order to ensure that the user submitted price can be converted into a valid currency.

The user supplied end date is validated as a date time in the future. The date should have ISO 8601 format.

Item expiration is implemented by, on `Item` creation, submitting an `Event` referencing both the item's id and it's expiration date into MiniBids 'events' database collection. The database is instructed to delete this `Event` when the expiration time is reached. MiniBid listens for deletions happening in the 'events' collection. When an `Event` is deleted MiniBid retrieves the `Item` the deleted event was attached to. MiniBid also retrieves the item's `Auction`. If the auction has any bids then the `Item` is marked as 'SOLD' and the `Auction` is updated with the winner's user id. Otherwise the `Item` marked as 'EXPIRED'.

> 3. Authorised users should be able to bid on an item, so long as it is not their own, has not expired and their bid is higher than the items current highest bid. When the item expires the user with the highest bid wins.

When an `Auction` is created its `current_price` field is set to the starting price of the item being sold. When a bid is made the `current_price` is updated to the bid amount. `Bids` are validated by ensuring they are higher than the auction's `current_price`.

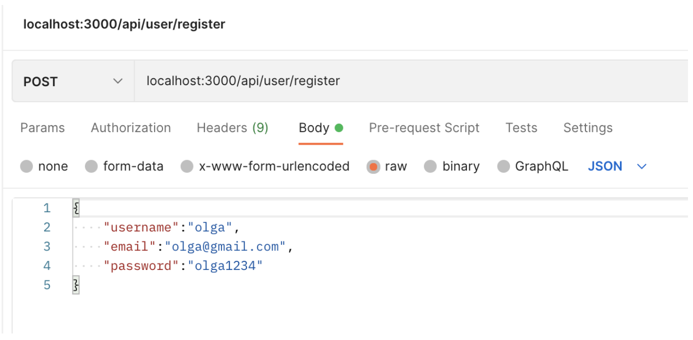
The accuracy of auctions is only to the minute; The winning bid should always be a bid placed within 60 seconds of the auction expiring, but this could mean, for example, a winner being a bid placed 30 seconds after the item had expired. This constraint is due to the speed MongoDB is able to delete an expired a document in a collection and notify applications listening for a changes. MiniBid could improve this by checking if the winning bid came seconds after the item's expiration, and if so declare the previous bid the winner. However, considering the bid timestamp isn't created until after the server recieves and validates the bid, there would still always be a possible delay between a user making a bid and the bid being acknowledged. In a future implementation, MiniBid could solve this by having the client send the bid's timestamp along with the bid and then the server validating the bid against the item's expiration.

MiniBid RESTful API Endpoints

Users should first register and login using the following endpoints:

- /api/user
  - /register
  - /login

> To register an account:

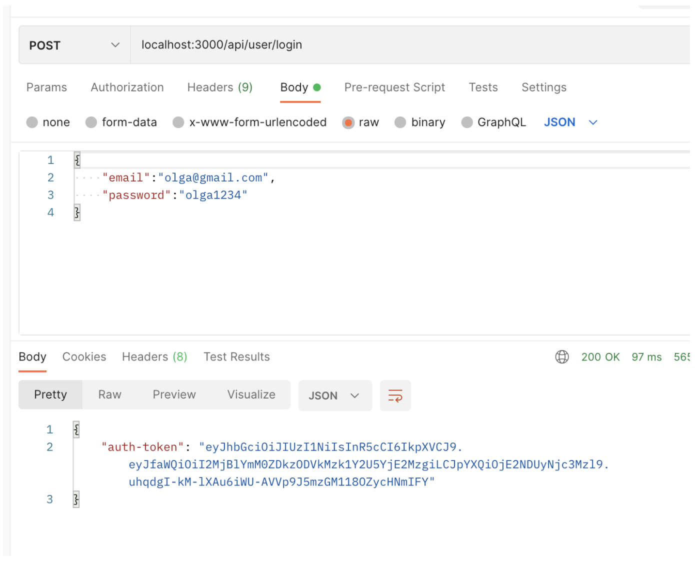


localhost:3000/api/user/register

POST localhost:3000/api/user/register

```
{
    "username":"olga",
    "email":"olga@gmail.com",
    "password":"olga1234"
}
```

> To login and recieve an `auth_token`:


POST localhost:3000/api/user/login

Params | Authorization | Headers (9) | Body | Pre-request Script | Tests | Settings

none | form-data | x-www-form-urlencoded | raw | binary | GraphQL | JSON

```
{
    "email":"olga@gmail.com",
    "password":"olga1234"
}
```

Body | Cookies | Headers (8) | Test Results — 200 OK 97 ms 565

Pretty | Raw | Preview | Visualize | JSON

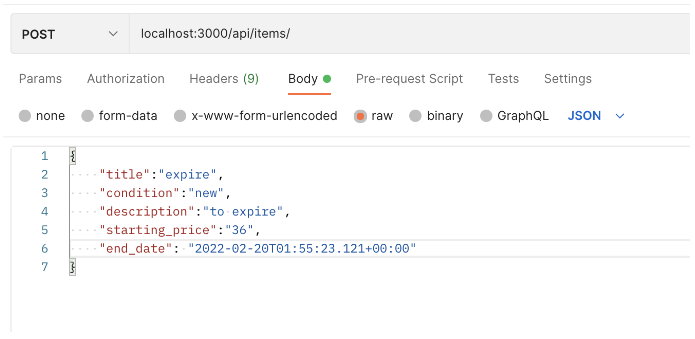
```
{
    "auth-token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.
        eyJfaWQiOiI2MjBlYmM0ZDkzODVkMzk1Y2U5YjE2MzgiLCJpYXQiOjE2NDUyNjc3Mzl9.
        uhqdgI-kM-lXAu6iWU-AVVp9J5mzGM118OZycHNmIFY"
}
```

For every other endpoint requests must come from registered users, they should have the user's `auth_token` in thier headers.

To read all items, post an item to sell or read an item's auction, users should send requests to the following API endpoints (where :itemId is the `Item:_id`):

- /api/items
    - /:itemId
    - /:itemId/auction

> To post and item for auction:


POST localhost:3000/api/items/

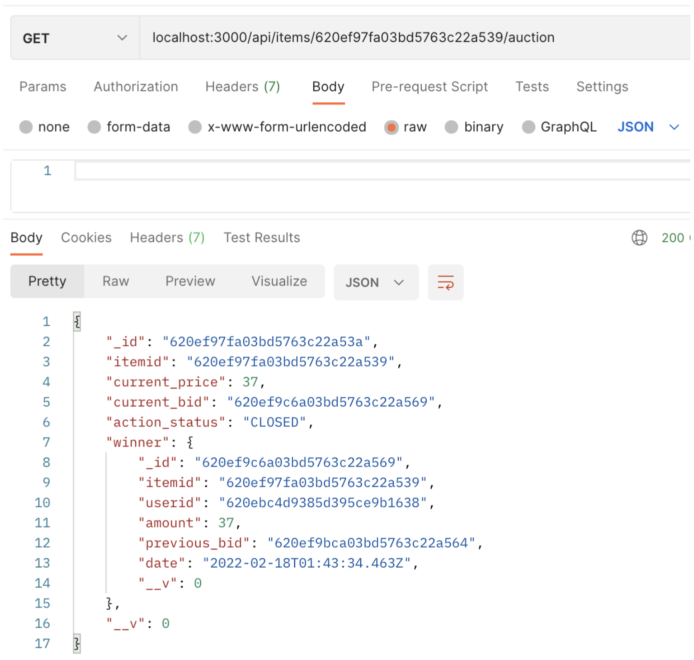
Params | Authorization | Headers (9) | Body | Pre-request Script | Tests | Settings

none | form-data | x-www-form-urlencoded | raw | binary | GraphQL | JSON

```
{
    "title":"expire",
    "condition":"new",
    "description":"to expire",
    "starting_price":"36",
    "end_date": "2022-02-20T01:55:23.121+00:00"
}
```

> Users must GET an item's auction in order to get the current price the item is selling for. Once an item expires it expires, users can findthe winner's `User:_id` by getting the auction:

localhost:3000/api/items/620ef97fa03bd5763c22a539/auction



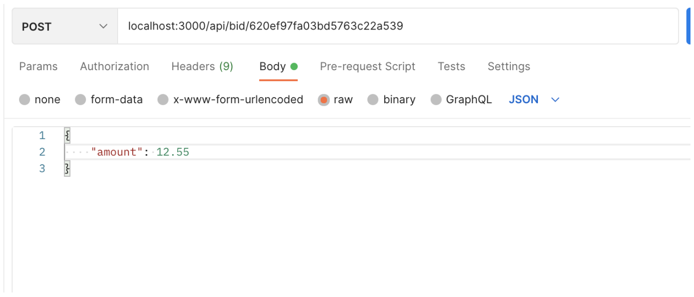
```
localhost:3000/api/items/620ef97fa03bd5763c22a539/auction
```

```
{
    "_id": "620ef97fa03bd5763c22a53a",
    "itemid": "620ef97fa03bd5763c22a539",
    "current_price": 37,
    "current_bid": "620ef9c6a03bd5763c22a569",
    "action_status": "CLOSED",
    "winner": {
        "_id": "620ef9c6a03bd5763c22a569",
        "itemid": "620ef97fa03bd5763c22a539",
        "userid": "620ebc4d9385d395ce9b1638",
        "amount": 37,
        "previous_bid": "620ef9bca03bd5763c22a564",
        "date": "2022-02-18T01:43:34.463Z",
        "__v": 0
    },
    "__v": 0
}
```

In order to bid on an item a user should send a POST requsest containing the `Bid` to the following endpoint (where :itemId is the `Item:_id`):

- /api/bid/:itemId

> Bid requests should be sent with an `auth_token` which authorizes any user **except** the user who created the `Item` (i.e. Users cannot bid on their own items).



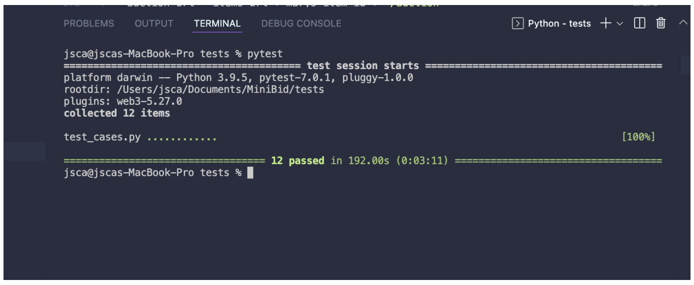
```
localhost:3000/api/bid/620ef97fa03bd5763c22a539
```

```
{
    "amount": 12.55
}
```

Testing

Testing was done using testcases written in Python. The testcases used can be found in `./test/test_cases.py`.

Running the tests with pytest:



```
jsca@jscas-MacBook-Pro tests % pytest
======================================== test session starts =========================================
platform darwin -- Python 3.9.5, pytest-7.0.1, pluggy-1.0.0
rootdir: /Users/jsca/Documents/MiniBid/tests
plugins: web3-5.27.0
collected 12 items

test_cases.py ............                                                                     [100%]

================================== 12 passed in 192.00s (0:03:11) ===================================
jsca@jscas-MacBook-Pro tests % 
```

Academic Declaration