Secured Game Rental Club

Problem Description

Modern gaming is expensive, and with the switch to digital ownership physical game rental services are far and few between. However an opportunity exists for a local game store that can act as the mediary of game rentals among its customers.

Solution Description

A web application platform to connect customers wishing to lend and borrow games using the physical storefront as a mediator.

Platform Features

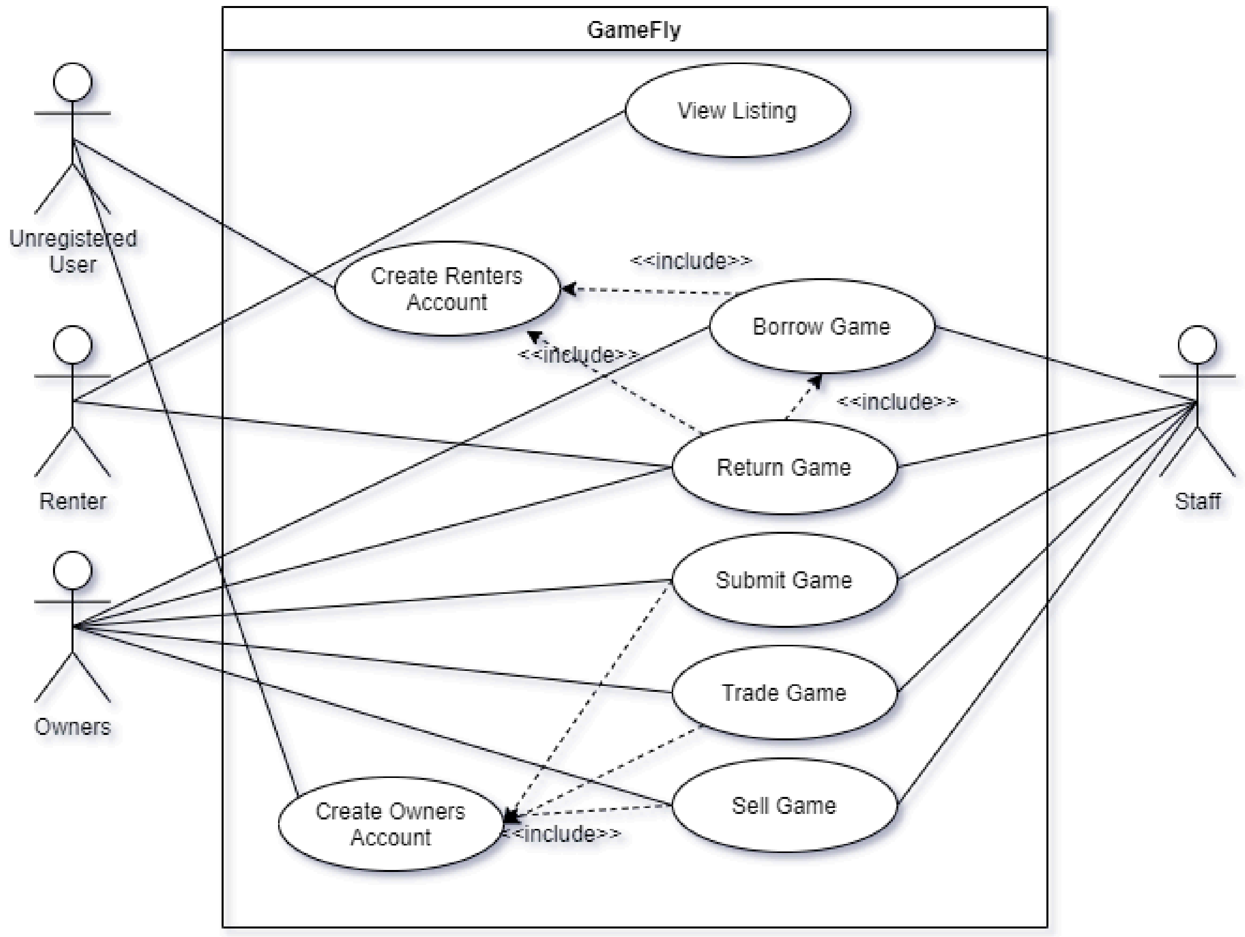
Minimum Viable Product features

- User Account
- View Game Catalog
- Borrow games, limited to collateral
- Submit games for rental

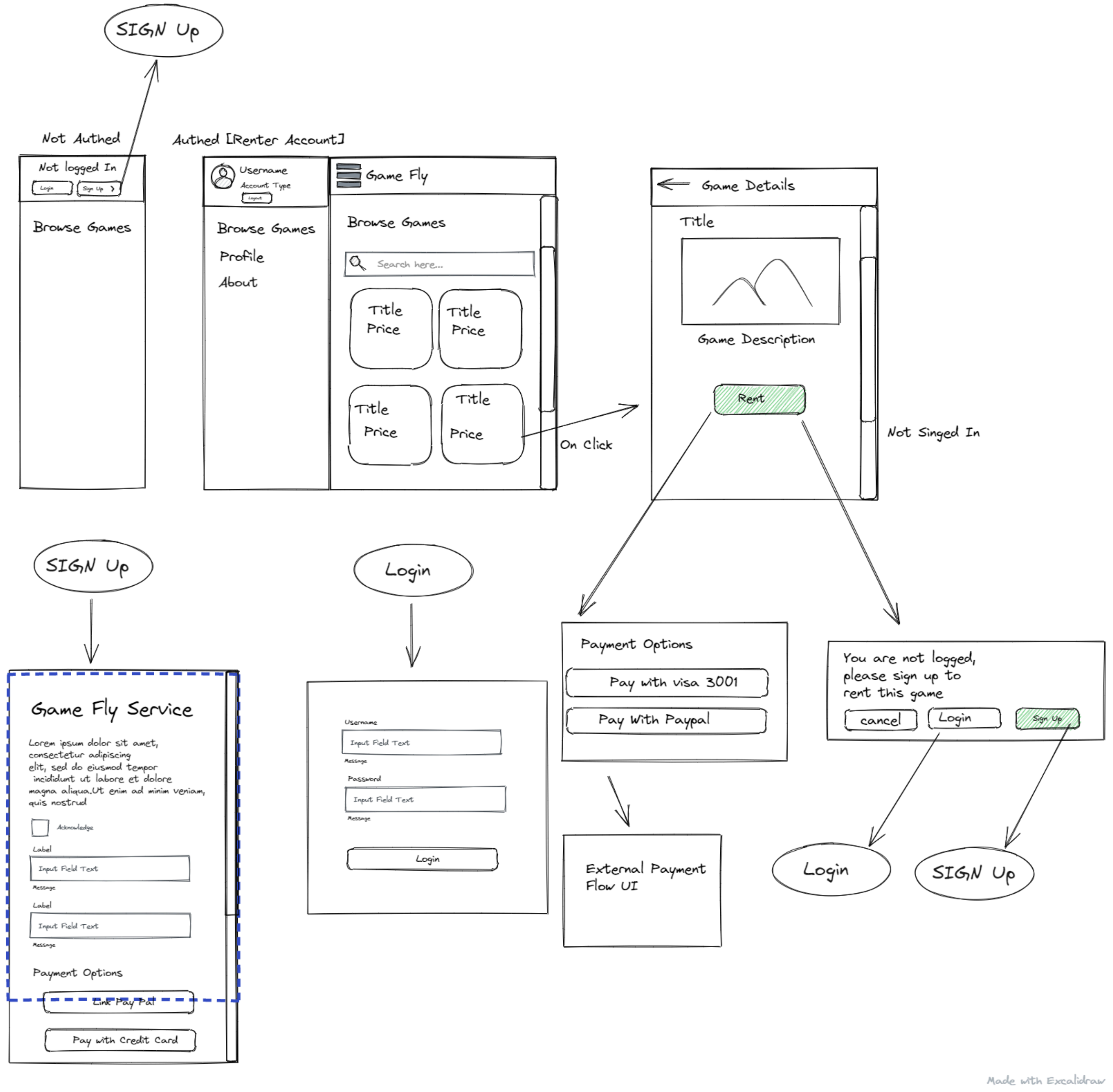
Postman Collection **Game Rental API** **Repository**

Note this repo uses FLASK JWT as that was the library used for token auth at the time. Auth would be implemented differently for recent projects as they use FLASk-JWT-Extended which is used in the more recent labs and the current version of FLASK MVC.

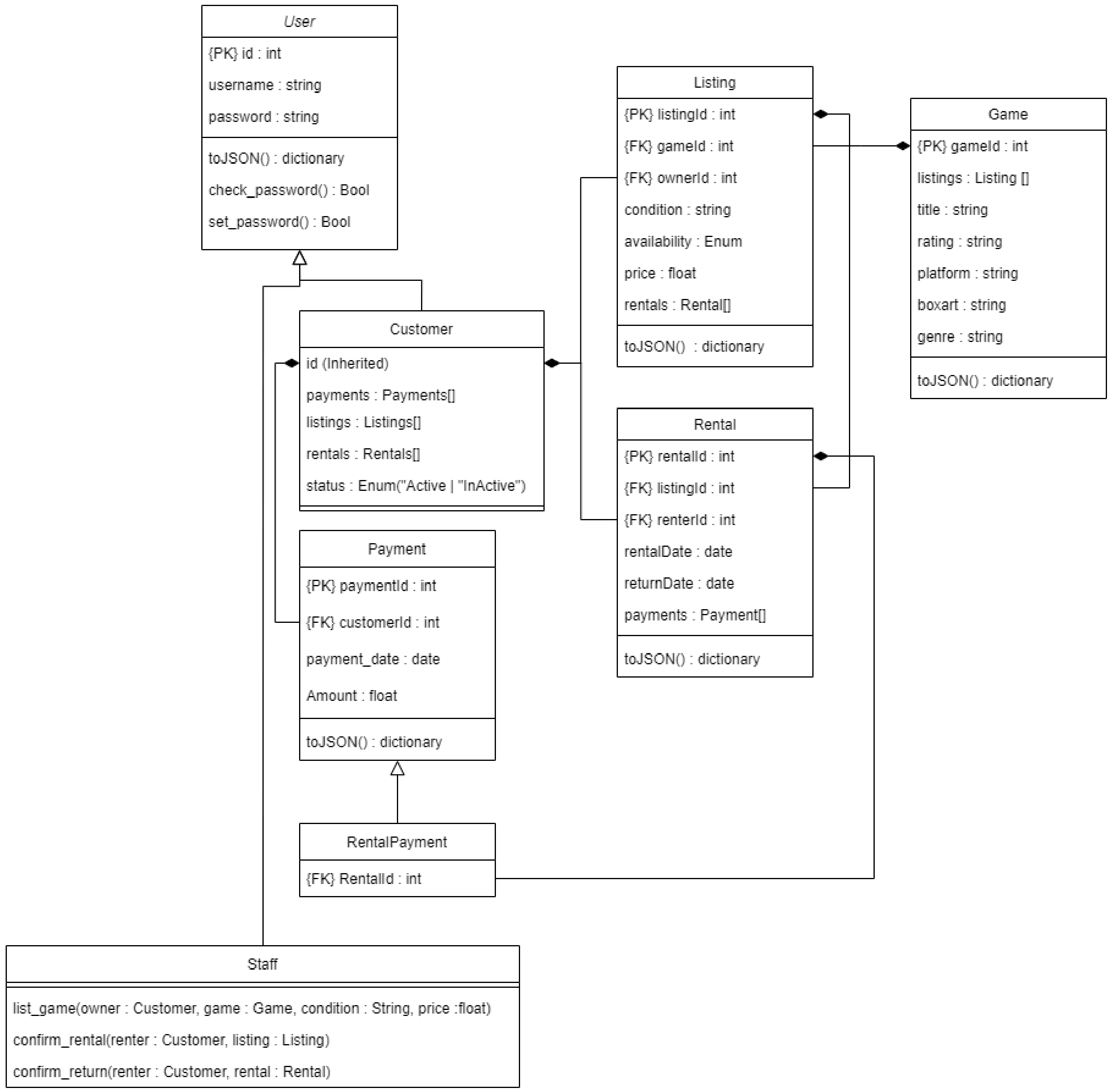
Use Case Diagram



Wireframes


SIGN Up
Not Authed
Authed [Renter Account]
Not logged In
Login
Sign Up
Browse Games
Username
Account Type
Logout
Browse Games
Profile
About
Game Fly
Browse Games
Search here...
Title
Price
Title
Price
Title
Price
Title
Price
On Click
Game Details
Title
Game Description
Rent
Not Singed In
SIGN Up
Game Fly Service
Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua.Ut enim ad minim veniam, quis nostrud
Acknowledge
Label
Input Field Text
Message
Label
Input Field Text
Message
Payment Options
Link Pay Pal
Pay with Credit Card
Login
Username
Input Field Text
Message
Password
Input Field Text
Message
Login
Payment Options
Pay with visa 3001
Pay With Paypal
External Payment Flow UI
You are not logged, please sign up to rent this game
cancel
Login
Sign Up
Login
SIGN Up
Made with Excalidraw

Model Diagram



Business Rules

- A rental that hasn't been returned past 2 weeks after the due date is considered lost, the borrower should have its account status set to banned. The store shall repurchase the game and relist it.
- Late fee is calculated on 10% of listing price per day after 2 weeks

- Each rental is for a period of 2 weeks

Out of Scope

- Dynamic rental limits,
- ie users can borrow the more they lend
- Full Online Payment integration

Planned Controllers

The code base is based on the flaskmvc template.

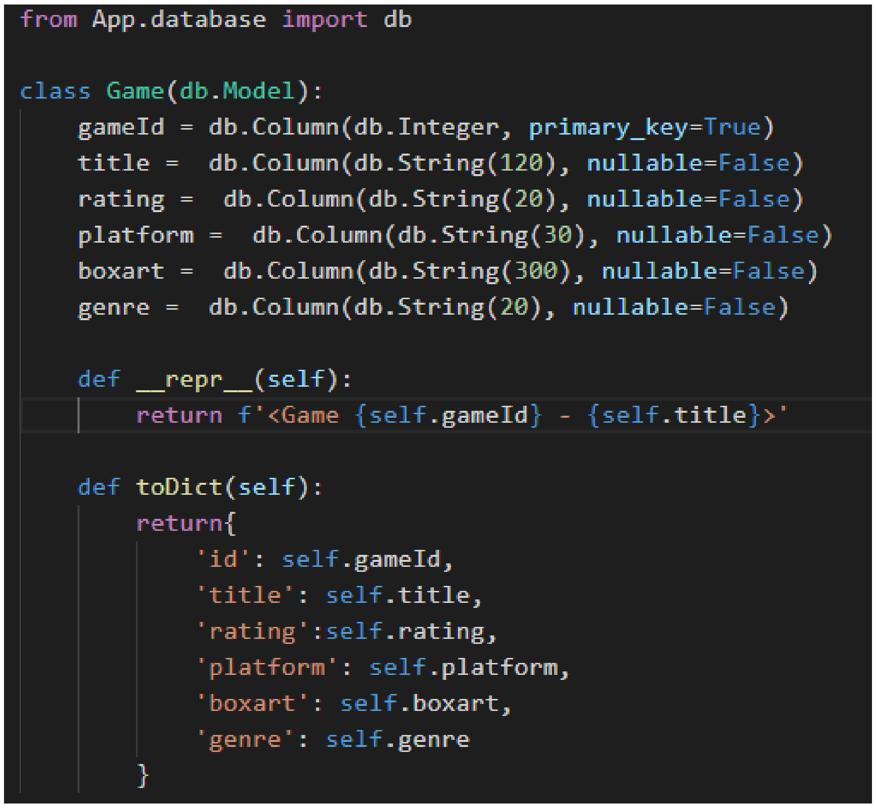
Req Num	Feature	Controller	Description
0	Authentication	@customer_required	
		@staff_required	
		auth.customer_login(username, password)	
		auth.staff_login(username, password)	
		auth.setup_jwt()	
		auth.logout()	
1	User Account	user.create_customer()	creates a customer object
		user.create_staff()	creates a staff object
2	Game Rental Catalogue	listing.get_avaiable_listings_json( )	Returns all available game listings
3	Borrow Game	rentals.rent_game(userId, listingid)	Updates a given listing to unavailable and creates a rental for the given user
4	Submit Games for rental	listing.list_game(userId, gameId, condition, price)	Creates a game listing by a given user
5	Log Payment	create_rental_payment(userId, rentalId, amount)	Creates a payment for a specified rental and user
6	Return Rental	rental.return_rental(rentalId)	Updates the return date of a given rental and updates the corresponding listing to available and create a rental payment including any late fees

Implementation Hints

Models

models/game.py

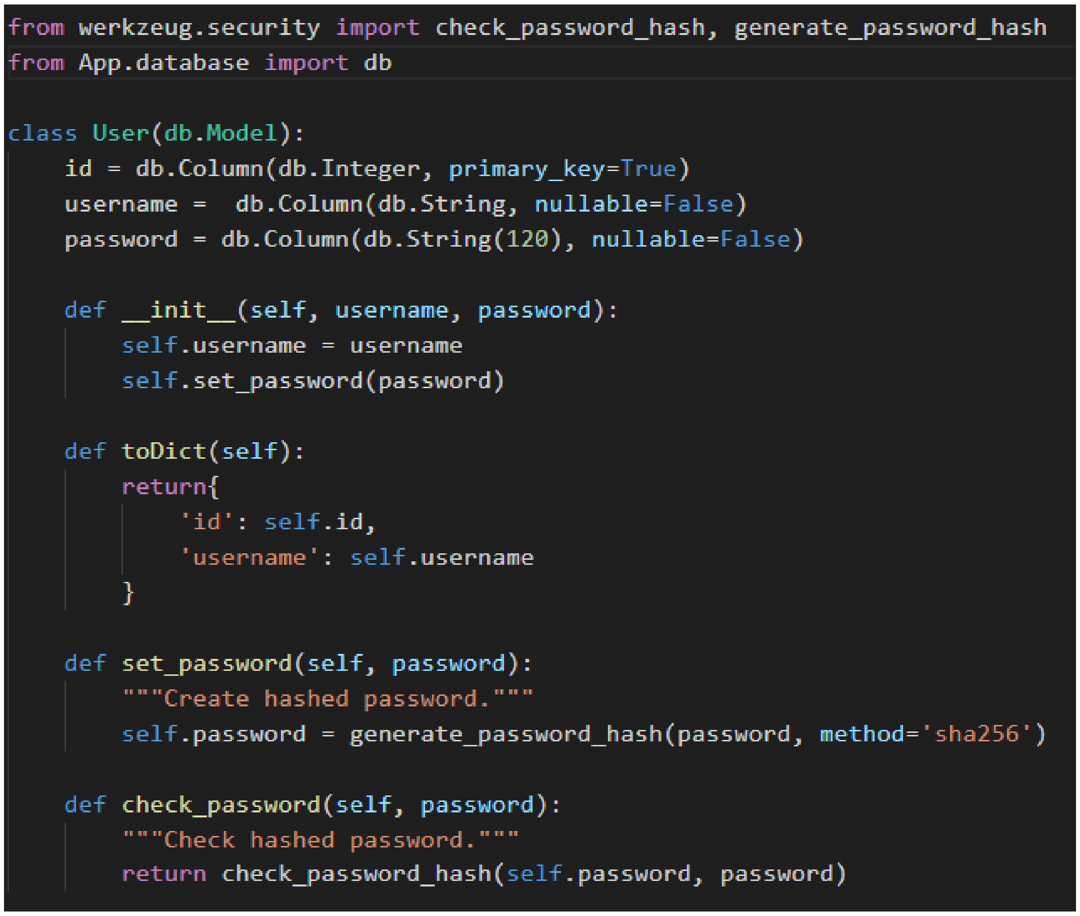
This is more or less a standard model with string properties. The **__repr__()** method defines how an object will be displayed on the console when passed to **print()**



```python
from App.database import db

class Game(db.Model):
    gameId = db.Column(db.Integer, primary_key=True)
    title =  db.Column(db.String(120), nullable=False)
    rating =  db.Column(db.String(20), nullable=False)
    platform =  db.Column(db.String(30), nullable=False)
    boxart =  db.Column(db.String(300), nullable=False)
    genre =  db.Column(db.String(20), nullable=False)

    def __repr__(self):
        return f'<Game {self.gameId} - {self.title}>'

    def toDict(self):
        return{
            'id': self.gameId,
            'title': self.title,
            'rating':self.rating,
            'platform': self.platform,
            'boxart': self.boxart,
            'genre': self.genre
        }
```

models/user.py
The user model is also the standard implementation from lab 4, except we define a constructor **__init__()** that encrypts the password when creating a user object.

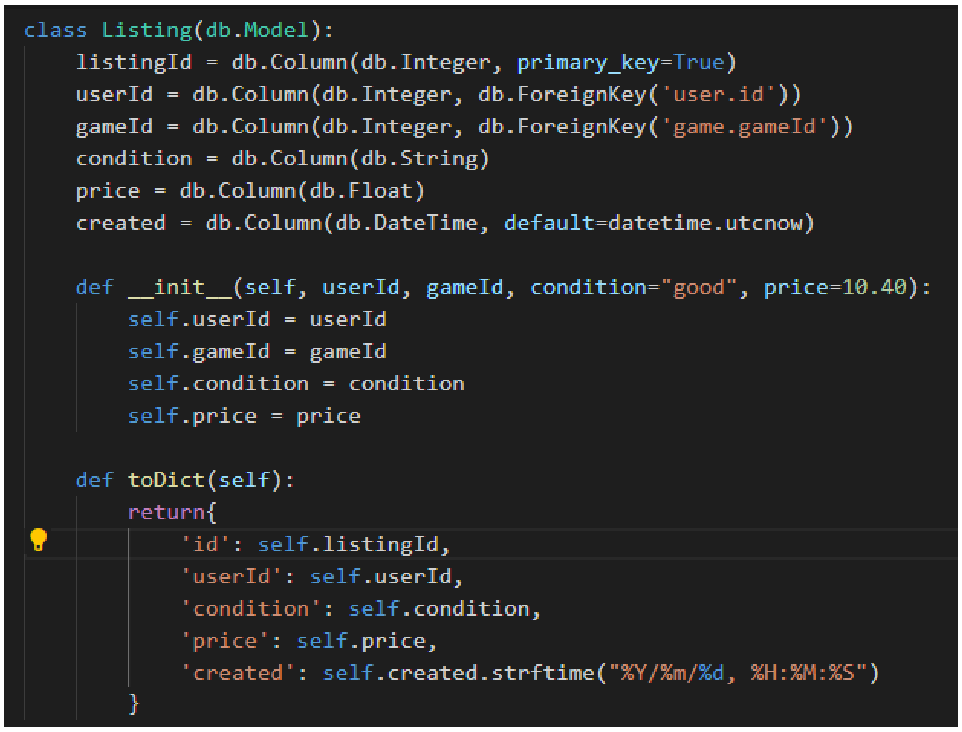
example: newuser = User('bob', 'bobpass')



```python
from werkzeug.security import check_password_hash, generate_password_hash
from App.database import db

class User(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    username =  db.Column(db.String, nullable=False)
    password = db.Column(db.String(120), nullable=False)

    def __init__(self, username, password):
        self.username = username
        self.set_password(password)

    def toDict(self):
        return{
            'id': self.id,
            'username': self.username
        }

    def set_password(self, password):
        """Create hashed password."""
        self.password = generate_password_hash(password, method='sha256')

    def check_password(self, password):
        """Check hashed password."""
        return check_password_hash(self.password, password)
```

models/listing.py

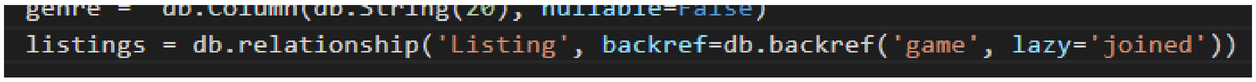
Because user to game is a many to many relationship, a bridge model listing must be created. A model Class is used instead of a Table object because it stores additional data (Association Object) on the relationship i.e. price, condition and created.



```python
class Listing(db.Model):
    listingId = db.Column(db.Integer, primary_key=True)
    userId = db.Column(db.Integer, db.ForeignKey('user.id'))
    gameId = db.Column(db.Integer, db.ForeignKey('game.gameId'))
    condition = db.Column(db.String)
    price = db.Column(db.Float)
    created = db.Column(db.DateTime, default=datetime.utcnow)

    def __init__(self, userId, gameId, condition="good", price=10.40):
        self.userId = userId
        self.gameId = gameId
        self.condition = condition
        self.price = price

    def toDict(self):
        return{
            'id': self.listingId,
            'userId': self.userId,
            'condition': self.condition,
            'price': self.price,
            'created': self.created.strftime("%Y/%m/%d, %H:%M:%S")
        }
```

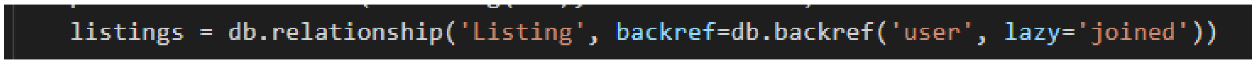
Relationship fields are then added to User and Game so that we can easily access all relevant listings from the User/Game object.

models/game.py



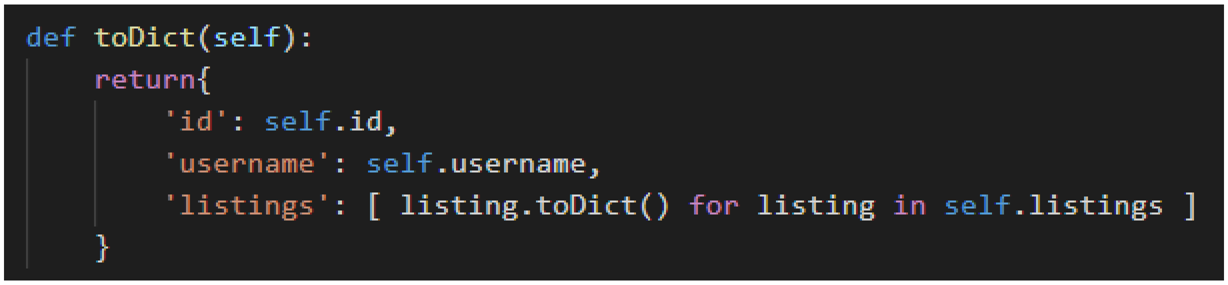
```python
genre = db.Column(db.String(20), nullable=False)
listings = db.relationship('Listing', backref=db.backref('game', lazy='joined'))
```

modesl/user.py



```python
    listings = db.relationship('Listing', backref=db.backref('user', lazy='joined'))
```

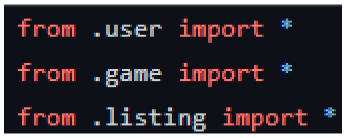
User's toDict() method is also updated to show an array of its relevant game listings.



```python
def toDict(self):
    return{
        'id': self.id,
        'username': self.username,
        'listings': [ listing.toDict() for listing in self.listings ]
    }
```

Note it is very important to import your models into the model folder so it can be accessed throughout the project.

models/__init__.py



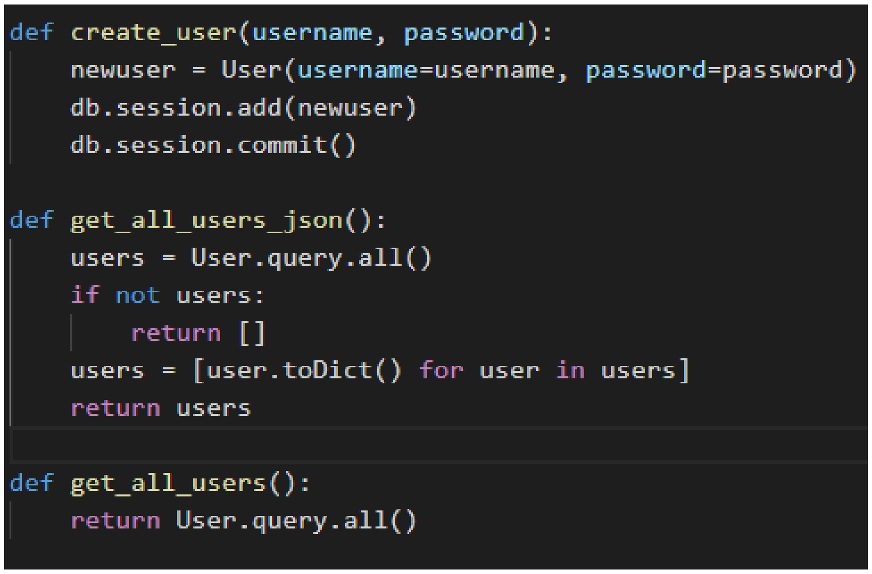
```python
from .user import *
from .game import *
from .listing import *
```

Controllers

Controllers are functions that manipulate our models according to any parameters passed to it from a view.

controllers/user.py

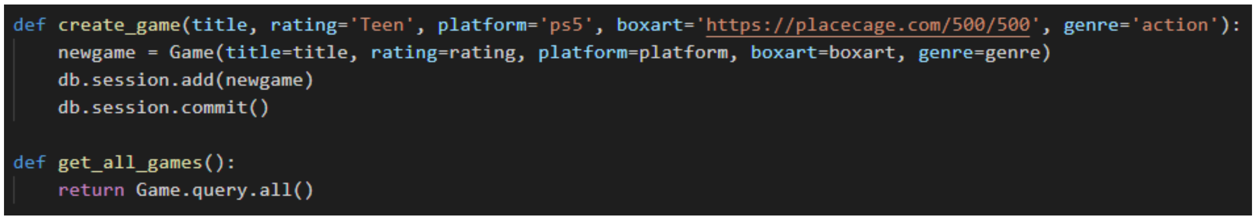
We have 3 user controllers for creating users and getting data as dictionaries or user objects



```python
def create_user(username, password):
    newuser = User(username=username, password=password)
    db.session.add(newuser)
    db.session.commit()

def get_all_users_json():
    users = User.query.all()
    if not users:
        return []
    users = [user.toDict() for user in users]
    return users

def get_all_users():
    return User.query.all()
```

controllers/game.py

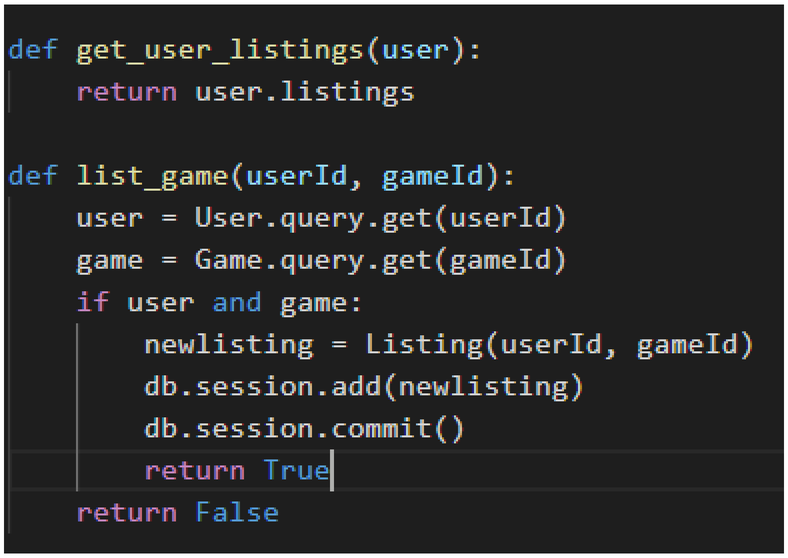
We have 2 game controllers for creating & retrieving games



```python
def create_game(title, rating='Teen', platform='ps5', boxart='https://placecage.com/500/500', genre='action'):
    newgame = Game(title=title, rating=rating, platform=platform, boxart=boxart, genre=genre)
    db.session.add(newgame)
    db.session.commit()

def get_all_games():
    return Game.query.all()
```

controllers/listing.py

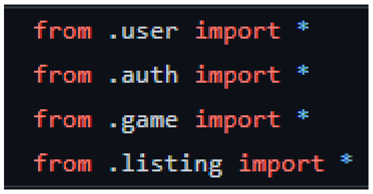
Finally we also have 2 controllers for listing. The first one retrieves all listings but the 2nd allows a user to list a game for renting.



```python
def get_user_listings(user):
    return user.listings

def list_game(userId, gameId):
    user = User.query.get(userId)
    game = Game.query.get(gameId)
    if user and game:
        newlisting = Listing(userId, gameId)
        db.session.add(newlisting)
        db.session.commit()
        return True
    return False
```

Just like with the models you need to import your controllers into the controllers module so they can be accessed.

controllers/__init__.py



```python
from .user import *
from .auth import *
from .game import *
from .listing import *
```

API Specification

When creating a REST api for a Client Side Rendered app, it is important to make a specification of the API so front end and back end developers can have a shared understanding on how to use/implement the API.

The table below is an example spec which guides the backend developers how to implement the views of the application. The spec also guides the front end developers on what to pass to and expect from the API routes.

NOTE: Before any view implementation is made an API spec should be first drafted.

Description	Method	URL	Request Headers, Body	Response Body, Status

Create account	POST	/signup	{
			“username”:<string>,
			“password”:<string>
			}
				Success
				{
				“message”:”account created”
				}, 201
				
				Bad Username
				{
				“error”: “username already taken”
				}, 400

Login	POST	/auth	{
			“username”:<string>,
			“password”:<string>
			}
				Success
				{
				“token”:<token>
				}, 200
				
				Bad Credentials
				{
				“error”: “invalid credentials”
				}, 400

(Customer) Enlist Game	POST	/listings	Authorization: JWT <token>
			
			{
			“gameid”:<id>,
			“ownder”:<id>,
			“condition”:< ‘good’ | ‘fair’ | ‘bad’ >,
			“price”: <float>
			}
				Success
				{
				“message”:”listing created”
				}, 201
				
				When token not provided
				{
				“error”:”not authenticated”
				}, 401
				
				Bad Game id
				{
				“error”:”game id <gameid> not found”

				}, 404 Bad Condition { “error”:”<condition> not a valid condition” }, 400

Show game listings	GET	/listings	?platform=< NSW |PS5|XBOX|PC>	[
				{
				“listingId”: <id>,
				“ownerId”: <id>,
				“gameId”: <id>,
				“condition”: < ‘good’ | ‘fair’ | ‘bad’ >,
				“availability”: true,
				“price”: <float>,
				“game”: {
				“gameId”:<id>,
				“title”:<string>,
				“rating”: <string>,
				“platform”: <platform>,
				“boxart”: <url>,
				“genre”: <string>
				}
				}
				]

(Staff) Make Payment	POST	/payment	Authorization: JWT <token>
			
			{
			“amount” :<float>
			}
				{
				“message”: “payment created”,
				“paymentId”: <id>
				}, 201

(Staff) Borrow Game	POST	/rentals	Authorization: JWT <token>
			
			{
			“listingId”: <id>
			“customerId” :<id>
			}
				Success
				{
				“message”: “rental created”
				}, 201
				
				Bad id
				
				{
				“error”: “Bad listing id given”
				}, 404

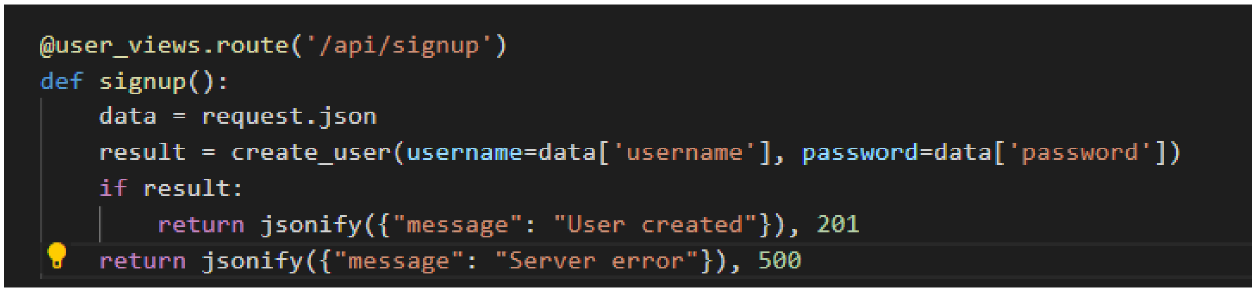
(Staff) Return Game	UPDATE	/rentals/<rentalI	Authorization: JWT <token>	Success

		d>	{
			“payment”:{
			“amount”: “float”
			}
			}
			
			or
			
			{
			“palymentId: <id>
			}
				{
				“message”: “rental updated”
				}, 201

Views

Now that some routes have been specified we can then implement the following view that conforms to the spec. Note how it makes use of the create_user() controller.

views/user.py



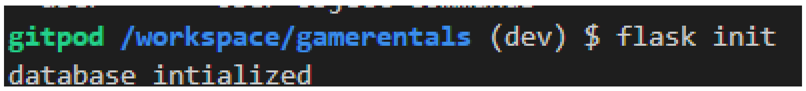
```python
@user_views.route('/api/signup')
def signup():
    data = request.json
    result = create_user(username=data['username'], password=data['password'])
    if result:
        return jsonify({"message": "User created"}), 201
    return jsonify({"message": "Server error"}), 500
```

Testing Controllers

wsgi.py can be used to create CLI commands which we can call to test our controllers. There are several custom commands created for creating users, games and listings.

Execute the following command (exclude the ‘$’):

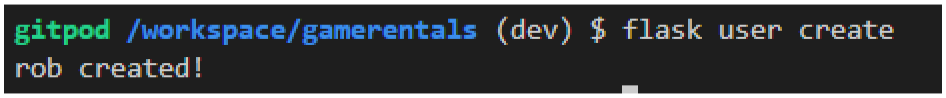
```
$ flask init
```



```
gitpod /workspace/gamerentals (dev) $ flask init
database intialized
```

This initializes the database, now try

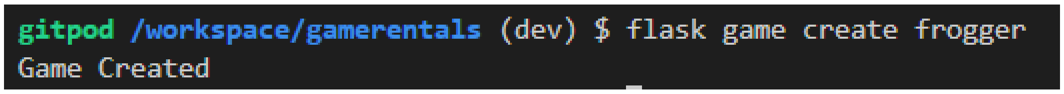
```
$ flask user create
```



```
gitpod /workspace/gamerentals (dev) $ flask user create
rob created!
```

This creates a default user, now we can create a game called ‘frogger’ with the following

```
$ flask game create frogger
```

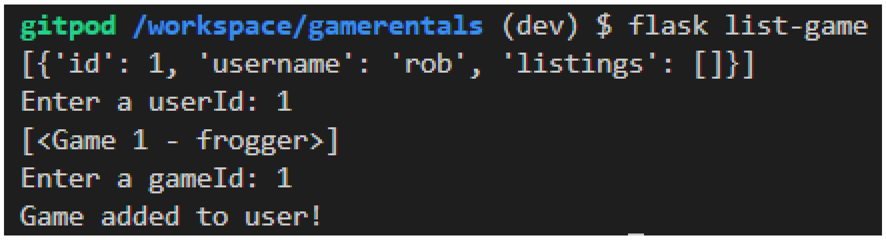


```
gitpod /workspace/gamerentals (dev) $ flask game create frogger
Game Created
```

Finally we can create a listing by running

```
$ flask create-listing
```

It will show the available users and games and prompt you for valid id’s.

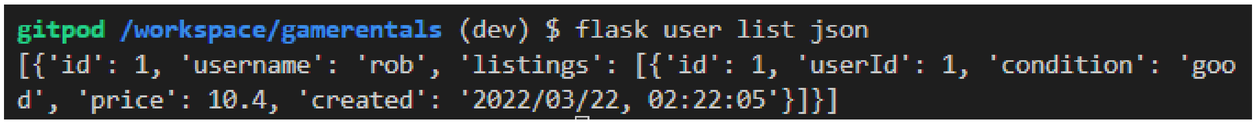


```
gitpod /workspace/gamerentals (dev) $ flask list-game
[{'id': 1, 'username': 'rob', 'listings': []}]
Enter a userId: 1
[<Game 1 - frogger>]
Enter a gameId: 1
Game added to user!
```

Now you when you list the user data as json with the following:

```
$ flask user list json
```

You can see the rob has a listing for frogger.



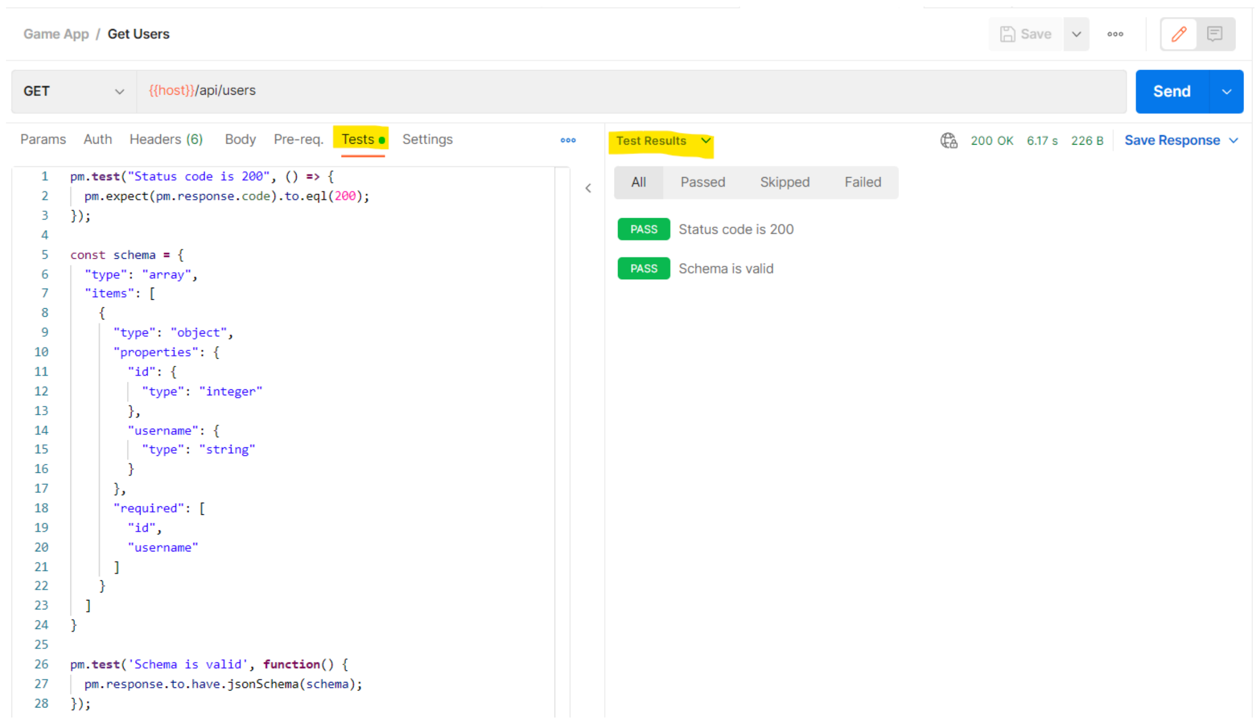
```
gitpod /workspace/gamerentals (dev) $ flask user list json
[{'id': 1, 'username': 'rob', 'listings': [{'id': 1, 'userId': 1, 'condition': 'goo
d', 'price': 10.4, 'created': '2022/03/22, 02:22:05'}]}]
```

You can use custom CLI commands for inserting data into your app.

Testing Views

Views are then tested using postman. A postman collection is created that tests all of the requests facilitated by the server.

[Postman Collection]



```
pm.test("Status code is 200", () => {
  pm.expect(pm.response.code).to.eql(200);
});

const schema = {
  "type": "array",
  "items": [
    {
      "type": "object",
      "properties": {
        "id": {
          "type": "integer"
        },
        "username": {
          "type": "string"
        }
      },
      "required": [
        "id",
        "username"
      ]
    }
  ]
}

pm.test('Schema is valid', function() {
  pm.response.to.have.jsonSchema(schema);
});
```

PASS Status code is 200
PASS Schema is valid