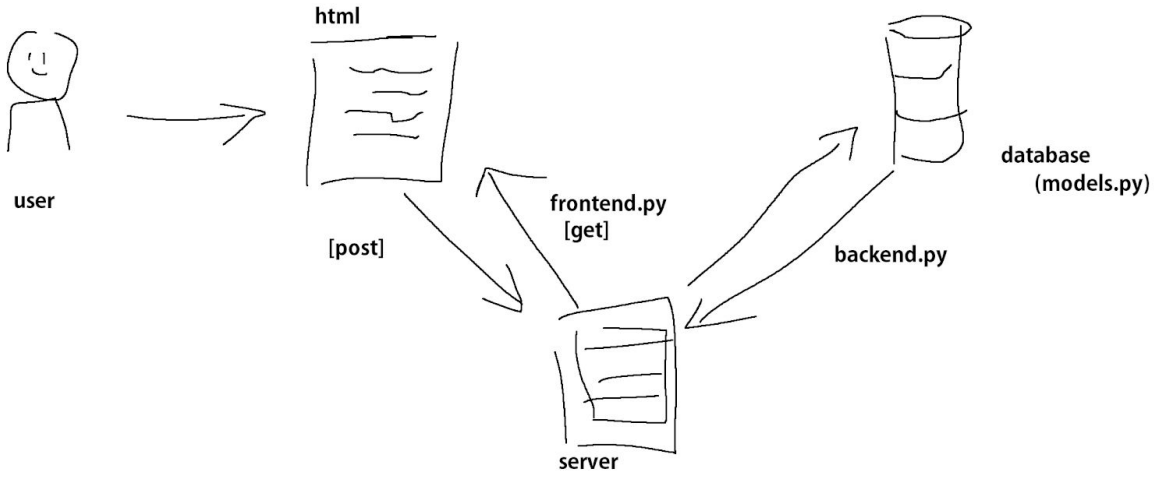


User interacts with HTML served by frontend.py, which contains routes and different methods that call backend.py which can interact with the database. Frontend.py contains methods all frontend http methods,such as post_login(), which may call the backend methods that contain business logic and interact with the database, such as login_user(). The database will contain classes for each entity, such as user, and ticket.

Code review was done in live meetings with the team.

R1	/login	[GET]
x		If the user hasn't logged in, show the login page
x		the login page has a message that by default says 'please login’
1.5		If the user has logged in, redirect to the user profile page
x		The login page provides a login form which requests two fields: email and passwords
		[POST]
x		The login form can be submitted as a POST request to the current URL (/login)
x		Email and password both cannot be empty
1.1		Email has to follow addr-spec defined in RFC 5322 (see https://en.wikipedia.org/wiki/Email_address for a human-friendly explanation)
1.2		Password has to meet the required complexity: minimum length 6, at

		least one upper case, at least one lower case, and at least one special character
1.3		For any formatting errors, render the login page and show the message 'email/password format is incorrect.'
x		If email/password are correct, redirect to /
1.4		Otherwise, redirect to /login and show message 'email/password combination incorrect'
R2	/register	[GET]
2.1		If the user has logged in, redirect back to the user profile page /
x		otherwise, show the user registration page
x		the registration page shows a registration form requesting: email, user name, password, password2
		[POST]
x		The registration form can be submitted as a POST request to the current URL (/register)
2.2		Email, password, password2 all have to satisfy the same required as defined in R1
x		Password and password2 have to be exactly the same
2.3		User name has to be non-empty, alphanumeric-only, and space allowed only if it is not the first or the last character.
2.4		User name has to be longer than 2 characters and less than 20 characters.
2.5		For any formatting errors, redirect back to /login and show message '{} format is incorrect.'.format(the_corresponding_attribute)
2.6		If the email already exists, show message 'this email has been ALREADY used'
2.7		If no error regarding the inputs following the rules above, create a new user, set the balance to 5000, and go back to the /login page

R3	/	[GET]
x		If the user is not logged in, redirect to login page

3.1		This page shows a header 'Hi {}'.format(user.name)
3.2		This page shows user balance.
x		This page shows a logout link, pointing to /logout
3.3		This page lists all available tickets. Information including the quantity of each ticket, the owner's email, and the price, for tickets that are not expired.
3.4		This page contains a form that a user can submit new tickets for sell. Fields: name, quantity, price, expiration date
3.5		This page contains a form that a user can buy new tickets. Fields: name, quantity
3.6		This page contains a form that a user can update existing tickets. Fields: name, quantity, price, expiration date
3.7		The ticket-selling form can be posted to /sell
3.8		The ticket-buying form can be posted to /buy
3.9		The ticket-update form can be posted to /update
R7	/logout	[GET, POST]
x		Logout will invalid the current session and redirect to the login page. After logout, the user shouldn't be able to access restricted pages.
R8	/*	[any]
8.1		For any other requests except the ones above, the system should return a 404 error

Create email format checking method in backend: validate_email(email)
Create password format checking method in backend: validate_password(password)

In login_user(): (joel)

- Use email format checker (1.1)
- Use password format checker (1.2)
- Returns a list of errors string if there are any errors

In /login route (joel)

- If login_user() returns a non-empty list, display those errors (1.3)

- Change message for no user from ‘login failed’ to ‘email/password combination incorrect’ (1.4)
    - Says to redirect rather than render_template, but they want a message and you can’t redirect with a message… so for now I have just used render_template (Ding: “*you can redirect with params and update the login route to get args from the request object and set message there*”)
- If user is logged in redirect to ‘/’ (1.5)

In /register route (Johnson)

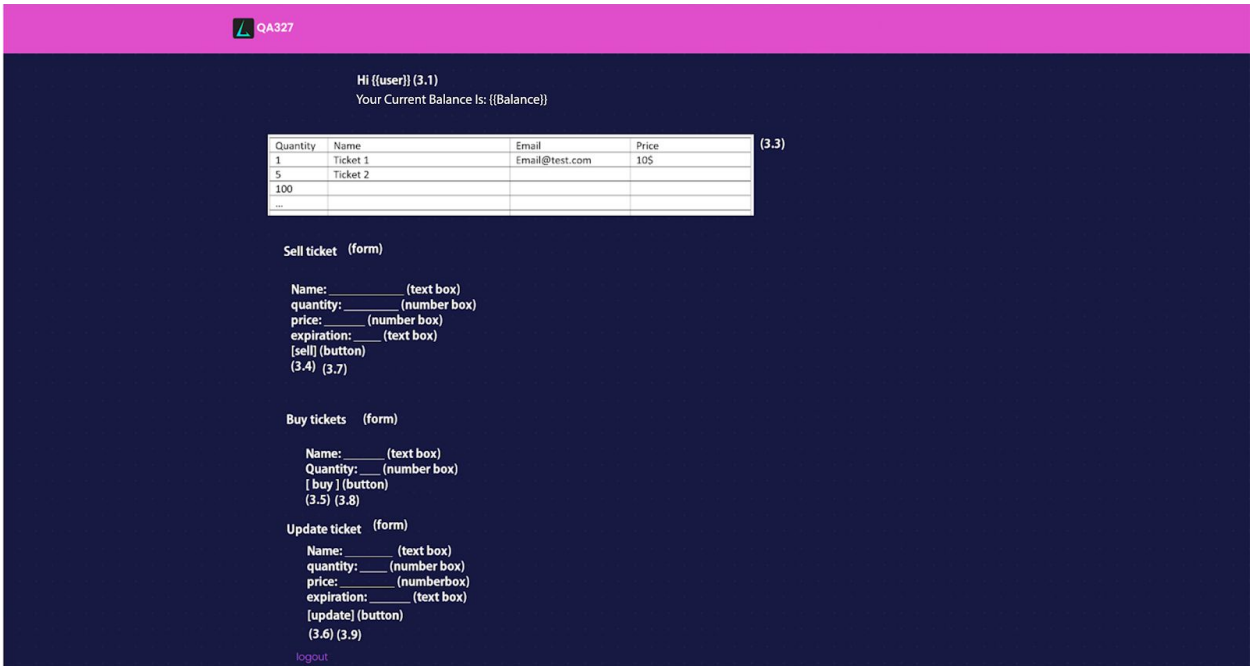
- **If user is logged in redirect to ‘/’ (2.1)**
- If register_user() returns any strings, format and display them (2.5)
- **If no errors redirect to ‘/login’ (2.7)**

In register_user() (isaac)

- **Use email format checker (2.2)**
- **Use password format checker (2.2)**
- **Returns a list of error string if there are any error**
- **Check if the username is non-empty, alphanumeric-only, and space allowed only if it is not the first or the last character. (2.3)**
- **Check if user name length is between 2 and 20 (2.4)**
- **Check if email is not in the database. Should give error because model has the field as unique (2.6)**
- **Set user balance to 5000 if no errors (2.7)**

In index.html (xinyang)

- Change message from ‘welcome’ to ‘Hi’ (3.1)
- Add a {{balance}} parameter with a message (3.2)
- Follow new design



Create /* route (Johnson)

- **Render 404.html (8.1)**

Create 404.html (Johnson)

- **Display message 404 not found (8.1)**

Database (Isaac)

- **Add field balance to user**
- **Add table tickets with fields: name, qty, price, expiration, owner**
-

Test plan

- How test cases of different levels (frontend, backend units, integration) are organized.
    - Folder for front end tests
    - Folder for back end tests
    - Folder for integration tests
    - Each folder contains a file for each requirement
    - Each file contains a bunch of tests for that requirement
- The order of the test cases (which level first which level second).
    - Backend tests first (as those are the most important and least brittle(shouldn't break))
    - Frontend tests (more brittle but we are testing an individual component)

    - Integration testing (requires both to work in the first place anyways)
- Techniques and tools used for testing.
    - Techniques
        - unit testing
        - integration testing
        - functionality coverage testing
        - Mocking
    - Tools
        - Selenium (UI/ frontend)
        - Pytest (unit testing front/backend/integration)
        - Github actions (continuous integration, so each commit is tested, and only passing branches can be merged)
- Environments (all the local environment and the cloud environment) for the testing.
    - Each member of the team will test locally on their computers (windows, mac, linux) first, then push their commits to their separate branches (on github which is in the cloud).
- Responsibility (who is responsible for which test case, and in case of failure, who should you contact)
    - The person who wrote the test case is responsible for it, in case of failure, contact the person who wrote it.
- Budget Management (you have limited CI action minutes, how to monitor, keep track and minimize unnecessary cost)
    - Each member of the group has 3000 minutes, the average build takes about 2 minutes. If we suppose the average build takes 5 minutes, we can have 600 builds per month (per user).
    - Each user will be limited to around 20 builds a day (which is a lot)
    - And we will also test locally before pushing commits.
    - We will regularly check our used minutes to ensure we stay within our individual budgets (you can see in the settings>billing tab)