Instruments Website

Link to github repository: https://github.com/LindeAkira/instruments

What is your project about, who is it for, what does it do?

My website is an instrument website. The user will be able to sign up, log in, browse instruments, and leave comments. Each instrument comes with an image and a description. The user will also be able to leave a comment for the instruments and see others' reviews. The main users for my website will be people who want to learn a new instrument so they can be informed by people who have played that instrument. The other kind of user will be veterans who want to inform people about instruments by leaving a review.

Final routes/functions:

Page	**Route**	**Function**
Signup	@app.route('/signup')	def signup()
Login	@app.route('/login')	def login()
Logout	@app.route('/logout')	def logout()
String	@app.route('/string')	def string()
Woodwind	@app.route('/woodwind')	def woodwind()
Brass	@app.route('/brass')	def brass()
Percussion	@app.route('/percussion')	def percussion()
Individual instrument page	@app.route('/instrument/<int:instrument_id>')	def instrument_details(instrument_id)
Add comment	@app.route('/comment/<int:instrument_id>', methods=['GET', 'POST'])	def add_comment(instrument_id)
Delete comment	@app.route('/delete_comment/<int:	def

	comment_id>/<int:instrument_id>', methods=['POST'])	delete_comment(comment_id, instrument_id)
404 page	@app.errorhandler(404)	def page_not_found(e):
414 page	@app.errorhandler(414)	def request_uri_too_long(e):
500 page	@app.errorhandler(500)	def internal_server_error(error):
Search page	@app.route('/search')	def search():
Admin comments page	@app.route('/admin/comments', methods=['GET', 'POST'])	def admin_comments():
URL limit	@app.before_request	def limit_url_length():

Iteration 1: Planning

Initial database notes and design:

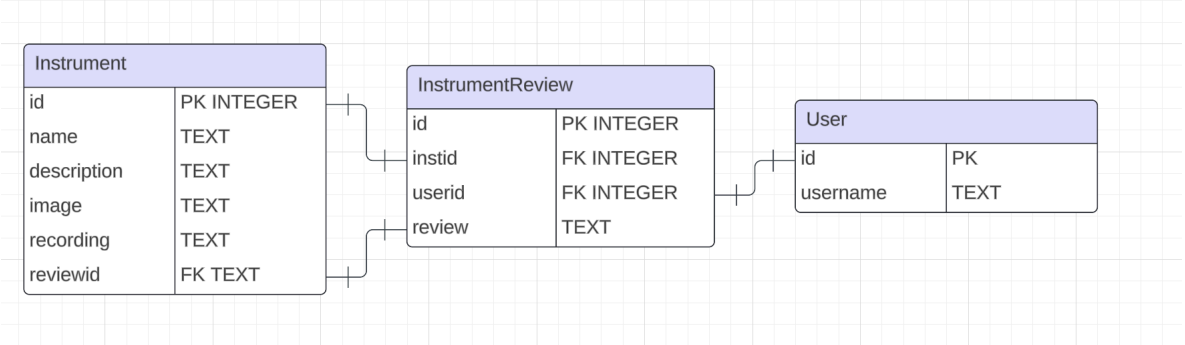
Tables:

- Instrument
- Instrument review
- User

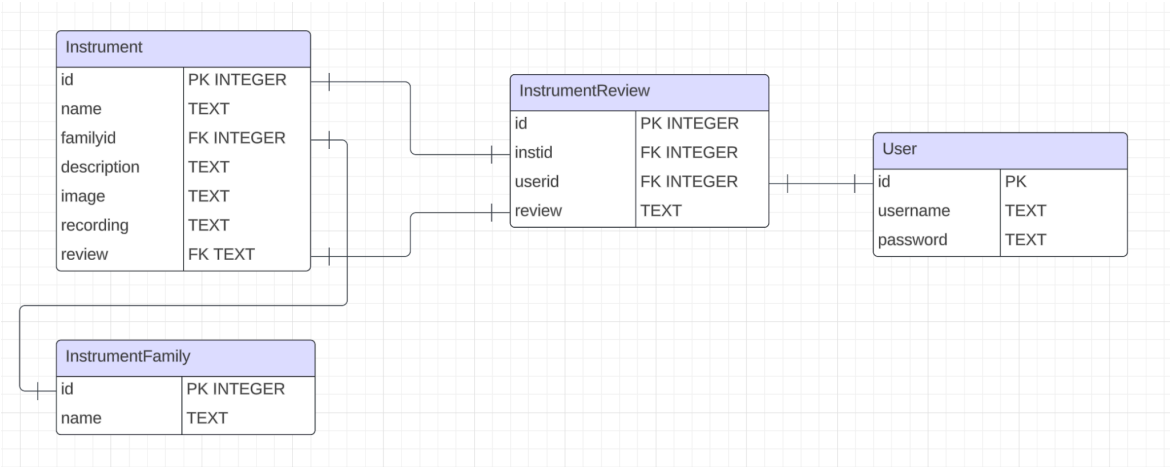
Functions:

- Login/sign up
- Review for instruments

Entity Relationship Diagram:



Update: The InstrumentFamily table has been added

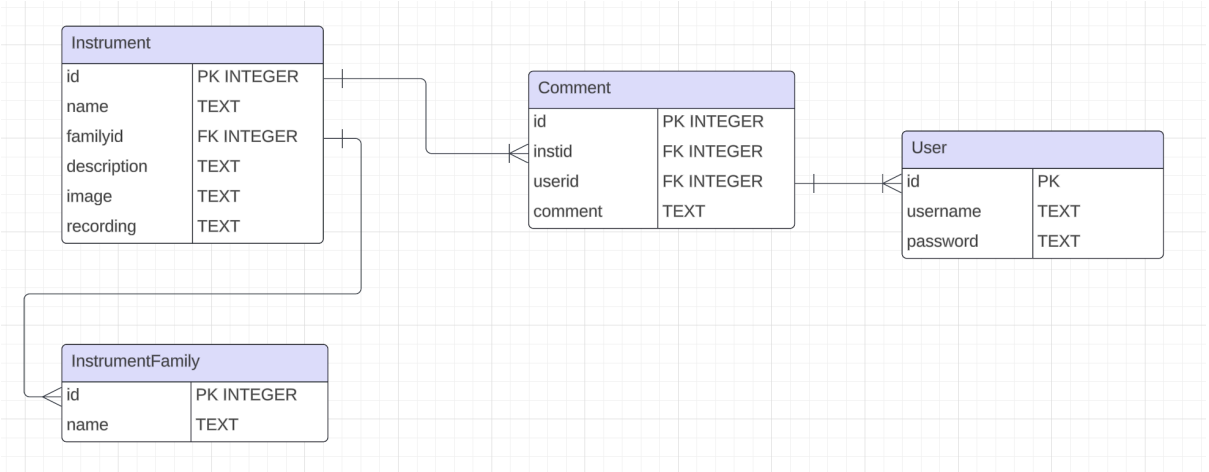


Changes:

1. **Addition of the InstrumentFamily Table:**
    - The second diagram introduces an **InstrumentFamily** table, which allows instruments to be categorised by families (e.g., strings, woodwinds, brass, etc.). This new table contains an **id** (primary key) and a **name** field to represent the family name. The **Instrument** table now has a **familyid** foreign key, linking each instrument to its family in the **InstrumentFamily** table.
2. **Addition of the password Field in the User Table:**
    - A **password** field has been added to the **User** table, indicating that users will now have credentials for authentication. This improves the security and user management in the system by enabling user authentication during login.

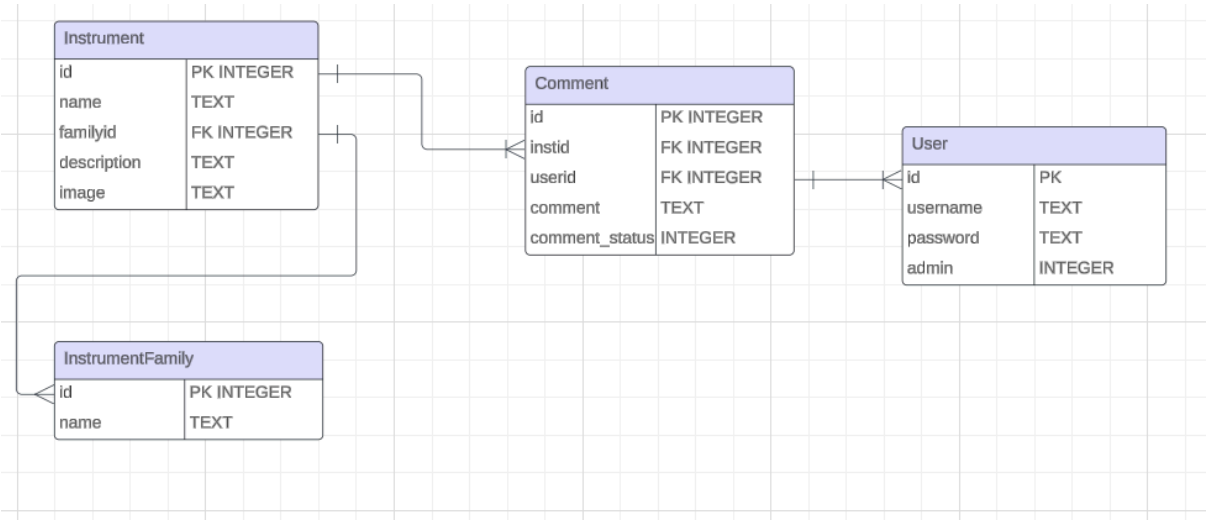
Why the Changes Were Made:

1. **Instrument Classification:**
    - Adding the **InstrumentFamily** table allows for better organisation and filtering of instruments. Users can now browse or search for instruments by family, which improves the scalability of the system as new instruments and families are added.
2. **User Authentication:**
    - The addition of a **password** field is necessary to ensure that users can log in securely. This change likely indicates that the system now includes more robust authentication features, such as login and password protection, which is essential for user accounts and the ability to leave reviews or comments.



Relationship Correctness:

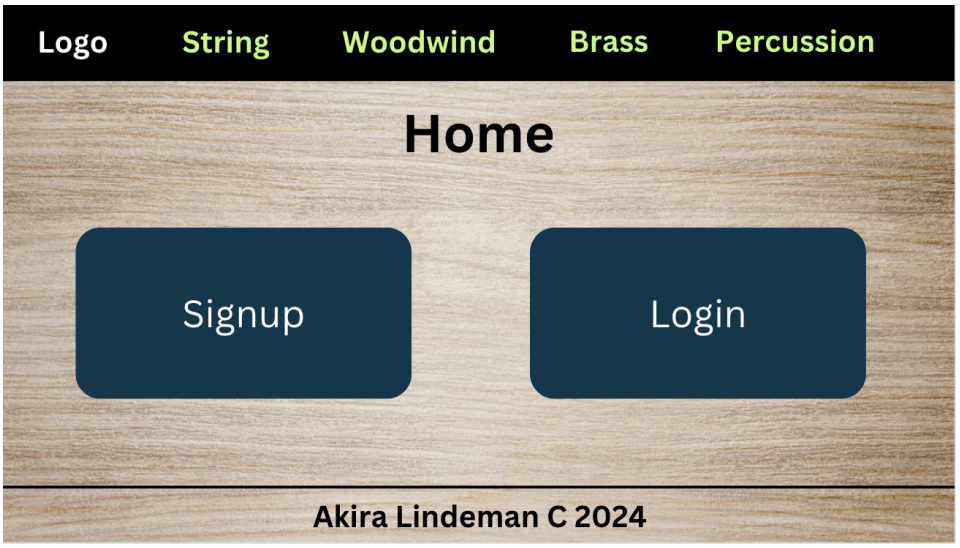
- **Instrument and InstrumentFamily**:
  - The **Instrument** table correctly has a foreign key **familyid** referencing the **InstrumentFamily** table. This many-to-one relationship makes sense because each instrument belongs to one family, while a family can have multiple instruments.
- **Instrument and InstrumentReview**:
  - The **InstrumentReview** table has a foreign key **instid** linking it to the **Instrument** table. This is a one-to-many relationship, where one instrument can have multiple reviews, but a review belongs to a single instrument, which is correct.
- **InstrumentReview and User**:
  - The **userid** in the **InstrumentReview** table references the **User** table. This one-to-many relationship is appropriate since a user can write multiple reviews, but each review belongs to only one user.



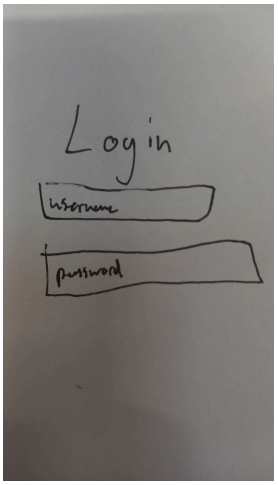
Changes:

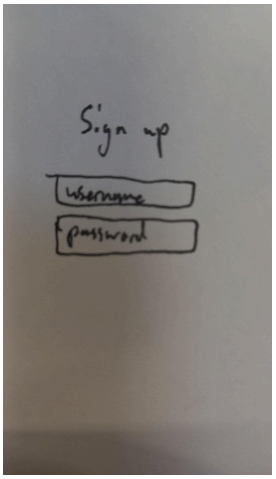
- Added admin column in the User table
    - This is so that comments can be checked by the admin and therefore, be displayed
- Deleted the recording column in the Instruments table
    - I canned the ide for adding a recording
- Added comment_status in the Comment table
    - This dictates whether or not the comment will be displayed

Website Design:

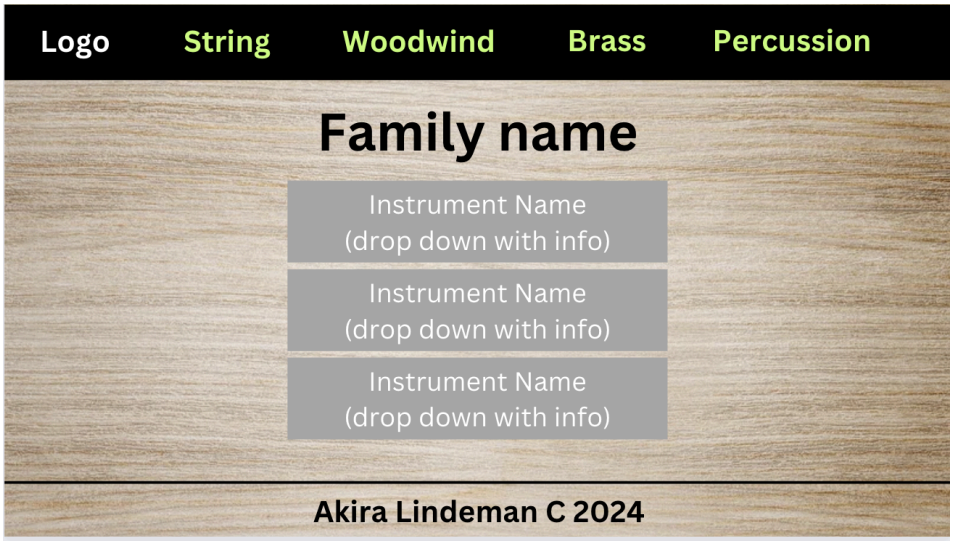


Wood was used for the background because wood is commonly used to make all instruments accept brass instruments. This may later be changed to a background image which has to do with instruments.





White background with blue font colour because that is the standard.



Routes:

Page	Route	Function
Signup	@app.route(‘/signup’)	def signup()
Login	@app.route(‘/login’)	def login()
Logout	@app.route('/logout')	def logout()
String	@app.route(‘/string’)	def string()
Woodwind	@app.route(‘/woodwind’)	def woodwind()
Brass	@app.route(‘/brass’)	def brass()
Percussion	@app.route(‘/percussion’)	def percussion()
Individual instrument page	@app.route('/instrument/<int:instrument_id>')	def instrument_details(instrument_id)
Add comment	@app.route('/comment/<int:instrument_id>', methods=['GET', 'POST'])	def add_comment(instrument_id)
Delete comment	@app.route('/delete_comment/<int:comment_id>/<int:instrument_id>', methods=['POST'])	def delete_comment(comment_id, instrument_id)

SQL Queries

Display String instruments	SELECT * FROM Instrument WHERE familyid = 1;
Searched String instruments	SELECT * FROM Instrument WHERE familyid = 1 AND name LIKE ?
Display String comments	SELECT * FROM Comment WHERE familyid = 1;
Display Woodwind instruments	SELECT * FROM Instrument WHERE familyid = 2;
Searched Woodwind instruments	SELECT * FROM Instrument WHERE familyid = 2 AND name LIKE ?
Display Woodwind comments	SELECT * FROM Comment WHERE familyid =2;
Display Brass instruments	SELECT * FROM Instrument WHERE familyid = 3;
Searched Brass instruments	SELECT * FROM Instrument WHERE familyid = 3 AND name LIKE ?
Display Brass comments	SELECT * FROM Comment WHERE familyid = 3;
Display Percussion instruments	SELECT * FROM Instrument WHERE familyid = 4;
Searched Percussion instruments	SELECT * FROM Instrument WHERE familyid = 4 AND name LIKE ?
Display Percussion comments	SELECT * FROM Comment WHERE familyid = 4;
Signup	INSERT INTO Users (username, password) VALUES (?, ?)
Login	SELECT * FROM Users WHERE username = ?
Display instrument details	SELECT * FROM Instrument WHERE id = ?
Display instrument comments	SELECT Comments.id, Comments.unchecked_comment, Comments.checked_comment, Users.username, Comments.user_id FROM Comments JOIN Users ON Comments.user_id = Users.id WHERE Comments.instrument_id = ?

Add comment	INSERT INTO Comments (instrument_id, user_id, unchecked_comment, checked_comment) VALUES (?, ?, ?, ?)

Iteration 2: Setting up Pages

Goals for iteration 2:

- Setup all of the routes and html pages. They don't need to look good yet but still should link to the layout.
- Link up all the pages. This means having buttons or nav bars in every page that can take you to where you need to be.
- Set up forms for signu and login

I first need to set up my flask routes on the routes.py file.

Added routes/pages/functions:

Route
@app.route('/')
@app.route('/signup')
@app.route('/login')
@app.route('/logout')
@app.route('/string')
@app.route('/woodwind')
@app.route('/brass')
@app.route('/percussion')
@app.route('/instrument/<int:instrument_id>')

For the next step, I need to make a templates folder where my pages will go, and a layout.html which I will use as a html template for most of my pages.

For my pages I set up:

home.html
signup.html
login.html
string.html
woodwind.html
brass.html
percussion.html
instrument.html
nav.html
footer.html
layout.html

The ones that I want to follow the layout.html has contain this code:

At the start:
{% extends “layout.html” %}
{% block content %}

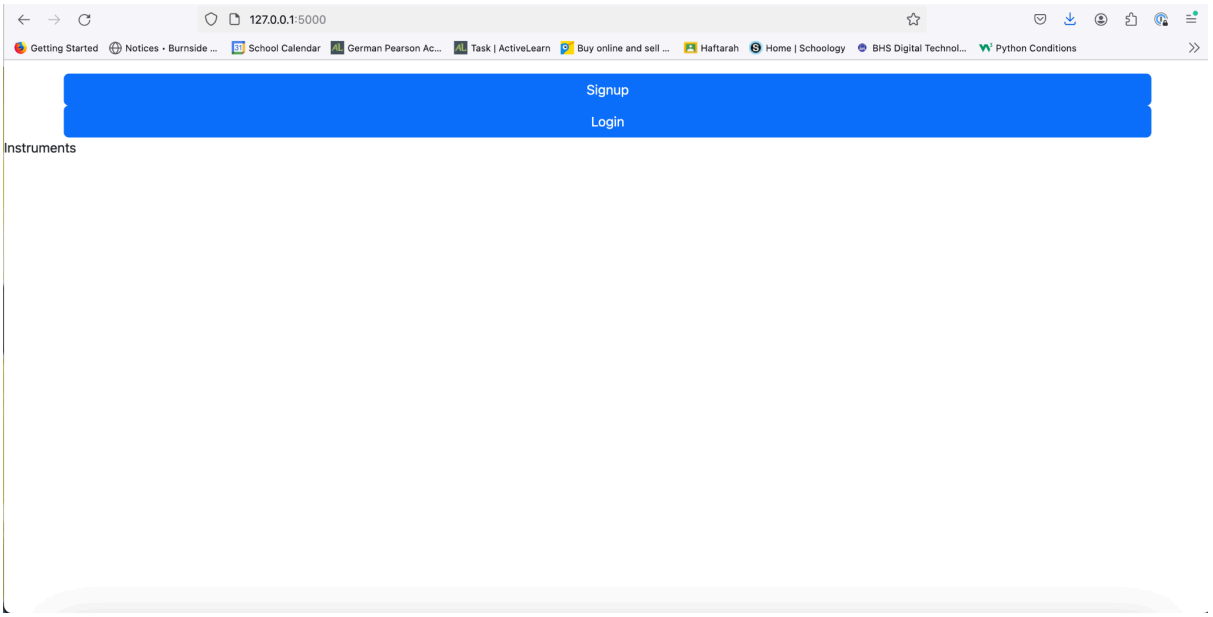
And at the end:
{% endblock %}

Home page:

Home page:
I want to set up the home page to have two buttons leading to the login and signup

```
  <div class="d-grid">
    <!-- "btn" creates the button, "btn-primary" makes the type of button primary, "btn-lg" makes the button large. -->
    <a href="/signup"<button type="button" class="btn btn-primary btn-block">Signup</button></a>
    <a href="/login"<button type="button" class="btn btn-primary btn-block">Login</button></a>
  </div>
</div>
```

I used Bootstrap for the buttons just so they look half decent and I will probably stick with something similar in the future. I added a home page because it is standard and the signup and login so that a user can get straight in with the interaction with the website so they can either create an account if they don't have one or log in if they do.



Signup and Login Pages:

Now I need to add the signup and login pages. I will need to create forms to retrieve data from the user.

I will use Flask's 'GET' and 'POST' requests from HTML files rendered in my routes. Here is an example of how I used it in my login route:

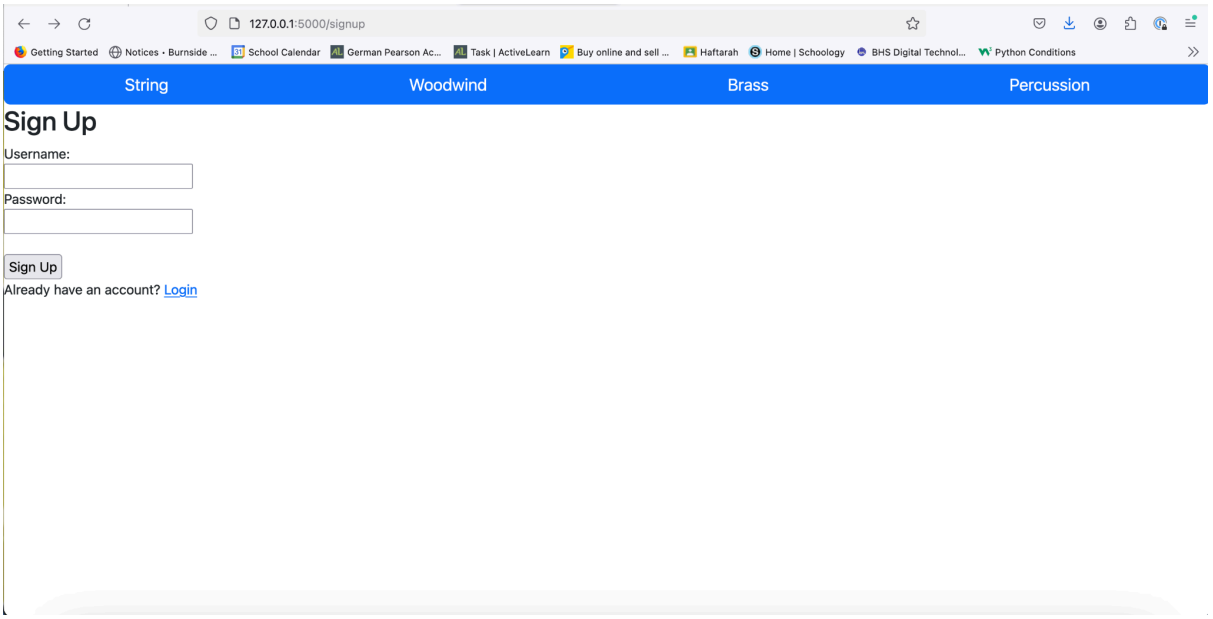
```python
@app.route('/login', methods=['GET', 'POST'])
def login():
    if request.method == 'POST':
        username = request.form['username']
        password = request.form['password']

        conn = sqlite3.connect("instruments.db")
        cur = conn.cursor()
        cur.execute("SELECT * FROM User WHERE username = ?", (username,))
        user = cur.fetchone()
        conn.close()

        if user and check_password(user[2], password):
            session['username'] = username
            return redirect(url_for('string'))
        else:
            flash('Invalid credentials', 'error')
    return render_template('login.html')
```

I need to put the methods GET and POST in my app route by doing: method=[‘GET’, ‘POST]. Request.method returns the type of form request used, in my case GET and POST. Request.form gives you the input of each box,in this case the username and password.

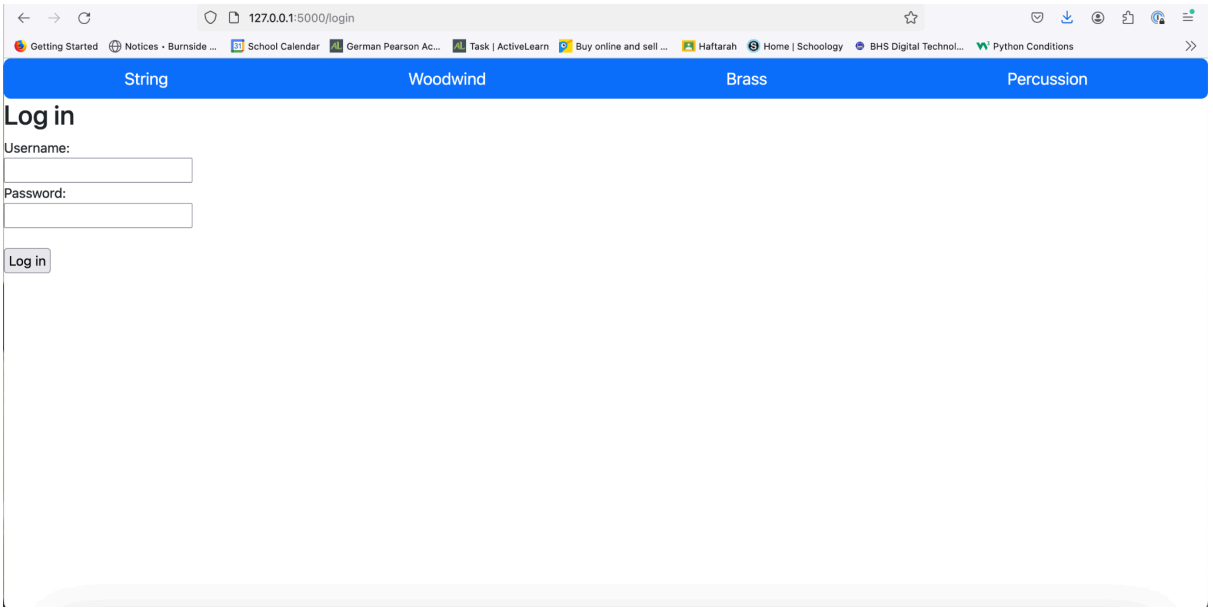
Here is an explanation of the code above:

When the user goes to the login page a it sends a GET request to the server. This retrieves, loads, and displays the login form. When a user submits the login form via a POST request, the username and password are retrieved from the form. The application then connects to a SQLite database, queries the User table for a matching username, and retrieves the user's data. If a user is found, the submitted password is compared with the stored hashed password using the check_password() function. If the credentials are valid, the username is stored in the session and the user is redirected to the string page. If the credentials are incorrect, an error message is flashed. If the request is a GET request or if login fails, the login page is rendered again for the user to try logging in.

Signup page:



Log in page:



Instrument family page (String):

The instrument family pages will display the buttons to the individual instrument pages of that family.

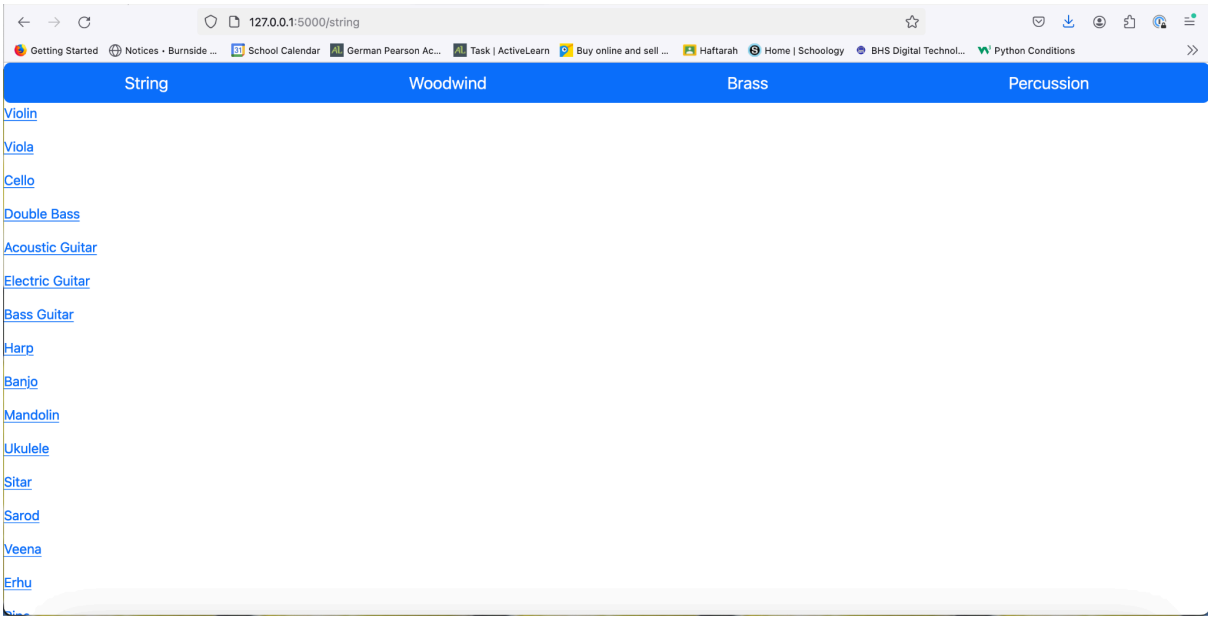
This is the code that I will use for this the family pages (String example):

```python
@app.route('/string')
def string():
    conn = sqlite3.connect('instruments.db')
    cursor = conn.cursor()
    cursor.execute("SELECT * FROM Instrument WHERE familyid = 1")
    string = cursor.fetchall()
    print(string)
    return render_template("string.html", results=string)
```

I use an sql queries to access my database to display all instruments of a certain family in this case string.

```
{% extends 'layout.html' %}

{% block content %}
  {% for string in results %}
    <p>
      <a href="/instrument/{{string[0]}}">{{ string[1] }}</a>
    </p>
  {% endfor %}

{% endblock %}
```

For my html I just have a for loop which loops through all of the string instruments and displays the name as a link to the instrument page.

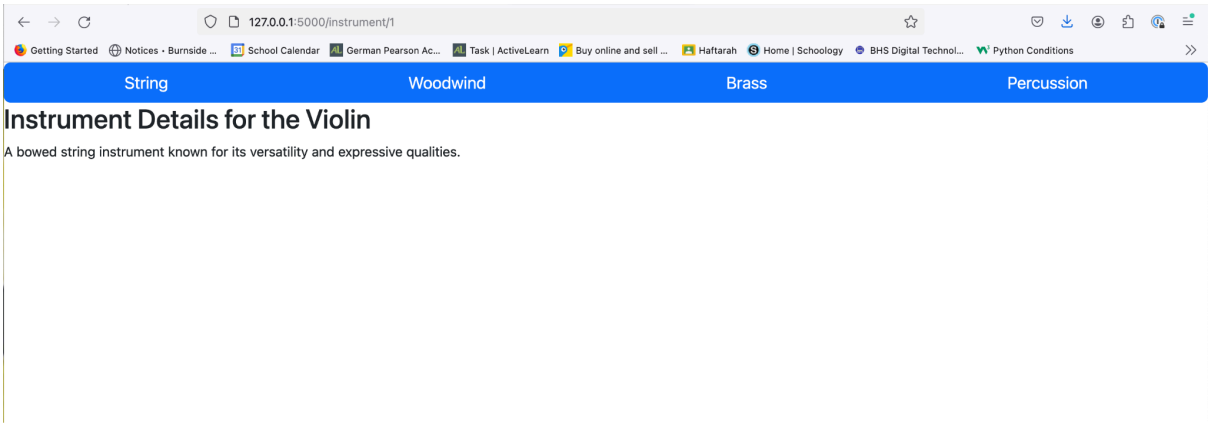


Individual instrument page (Violin)

For the individual instrument page it will just display the instrument name and the description of it when which is one of the columns in the database.

```python
# Individual instrument details page.
@app.route('/instrument/<int:id>')
def instrument_details(id):
    # print("The instrument id is {}".format(id))  # DEBUG
    conn = sqlite3.connect("instruments.db")
    cur = conn.cursor()
    cur.execute("SELECT * FROM Instrument WHERE id=?;", (id,))
    # fetchone returns a tuple containing the data for one entry
    instrument = cur.fetchone()
    conn.close()
    return render_template("instrument.html", instrument=instrument)
```

The first line defines a route in the Flask application, meaning it maps the URL pattern /instrument/<int:id> to the instrument_details function. The <int:id> part means that the URL expects an integer parameter called id. For example, if the URL is /instrument/1, the value 1 is passed as the id argument to the function. And will display information for the violin.



Iteration 3: Improving functions and adding comments

Goals for iteration 3:

- Improve login and signup by
    - encoding passwords
    - Validation Chekcs are added
- Add logout function
- Add comment function

This is not an aesthetically focused iteration. The CSS will come in a later iteration

Added routes/pages/functions:

Page	**Route**	**Function**
Logout	@app.route('/logout')	def logout()
Add comment	@app.route('/comment/<int:instrument_id>', methods=['GET', 'POST'])	def add_comment(instrument_id)

Improving signup and login functionality

For this iteration, one thing that I will develop is the security of the signup and login functions. One way that I will do this is by encoding a user's password hen they signup, so that when it is stored in my database I, and no one else who can access the database, can know what it is. I have done this by hashing the password in hexadecimal using the SHA-256 algorithm.

```
def hash_password(password):
    return hashlib.sha256(password.encode()).hexdigest()
```

This function first takes one parameter, "password", which is the password that needs to be hashed and "hashlib.sha256(password.encode())" creates a new SHA-256 hash object and hashes the encoded password ".hexdigest()" returns the hash as a hexadecimal string, which is a human-readable representation of the binary hash.

An example of this is :

30	Aaron	ae8a98872bfc075baddbd3fd9c9cb390219dbcdf9e551…

The user's password is encoded so me, the administrator, cannot see the password of my users.

This is used when a use is signing up.

```
# Add the user to the database
hashed_password = hash_password(password)
try:
    cur.execute("INSERT INTO User (username, password) VALUES (?, ?)
    conn.commit()
    flash("Account created successfully", "success")
    return redirect(url_for('login'))
```

But when they are logging in, the below function is used.

```
def check_password(stored_password, provided_password):
    return stored_password == hash_password(provided_password)
```

This function takes the parameters stored_password and provided password. Stored_password is the hashed/encoded password in the database that is retrieved, and provided_password is the password that the user enters when they are logging in. This function checks if the stored_password is the same as the provided password and compares them by hashing the provided password.

Validation checks

I have added certain validation checks to my signup and login functions so that the password and username follow certain guidelines.

```
# Validation check
if len(password) < 8 or not any(char.isdigit() for char in password) or not any(char.isupper() for char in password):
    flash("Password must be at least 8 characters long, must contain at least one digit, and must contain at least one uppercase letter", "error")
    return render_template('signup.html')
```

This line of code checks if the password length is less than 8 characters, not contain a digit, and not contain an uppercase letter. If the password doesn't meet all of these requirements, than It will flash an error message, and they won't be able to signup.

Add logout function

This function will just be a button that the user can press when they are logged in, to delete their user_id from session, so they are no longer logged in.

```
@app.route('/logout')
def logout():
    session.pop('username', None)
    flash("You have been logged out", "success")
    return redirect(url_for('lobby'))
```

The function removes the 'username' key from the session (which logs the user out), ensuring that no error occurs if the key doesn't exist. A flash message is displayed, informing the user that they have been logged out successfully. Finally, the user is redirected to the lobby page using redirect(url_for('lobby')), which generates the appropriate URL for the redirect.

Comments page

I decided to add a comments page where users can share their thoughts about instruments, and comments will be specific to each instrument. This will create another use for my website.

```
@app.route('/comment/<int:instrument_id>', methods=['GET', 'POST'])
def add_comment(instrument_id):
    if request.method == 'POST':
        # Get comment and user_id from the form
        comment_text = request.form.get('comment')
        user_id = session.get('user_id')
```

Again, I need to assign the methods GET and POST. I will have a form to get the comment from the user. The url will be /comment/ the instrument id as each instrument will have an individual comment page.

```
# Insert the comment into the database
query = "INSERT INTO Comments (instrument_id, user_id, unchecked_comment) VALUES (?, ?, ?);"
try:
    sql_queries(query, (instrument_id, user_id, comment_text), 'commit')
    flash('Comment added successfully!', 'success')
    return redirect(url_for('instrument_details', id=instrument_id))
except Exception as e:
    flash(f"An error occurred: {e}", 'error')
```

Here, I have a sql query to add the comment to my database, taking the instrument id, user id, and the comment. Once added it will display a flash message telling the user that their comment has been successfully added.

Iteration 4

- Add images of individual instruments so they will be displayed on the pages.
- Add sql_queries function to reduce repeated code
- Add a deleting function for comments
- Add search bar to individual instrument pages
- A blurred background and some more CSS such as
    - the nav bar is being improved
    - grid being added

Added routes/pages/functions:

Page	**Route**	**Function**
Delete comment	@app.route('/delete_comment/<int:comment_id>/<int:instrument_id>', methods=['POST'])	def delete_comment(comment_id, instrument_id)

Add images of individual instruments so they will be displayed on the pages.

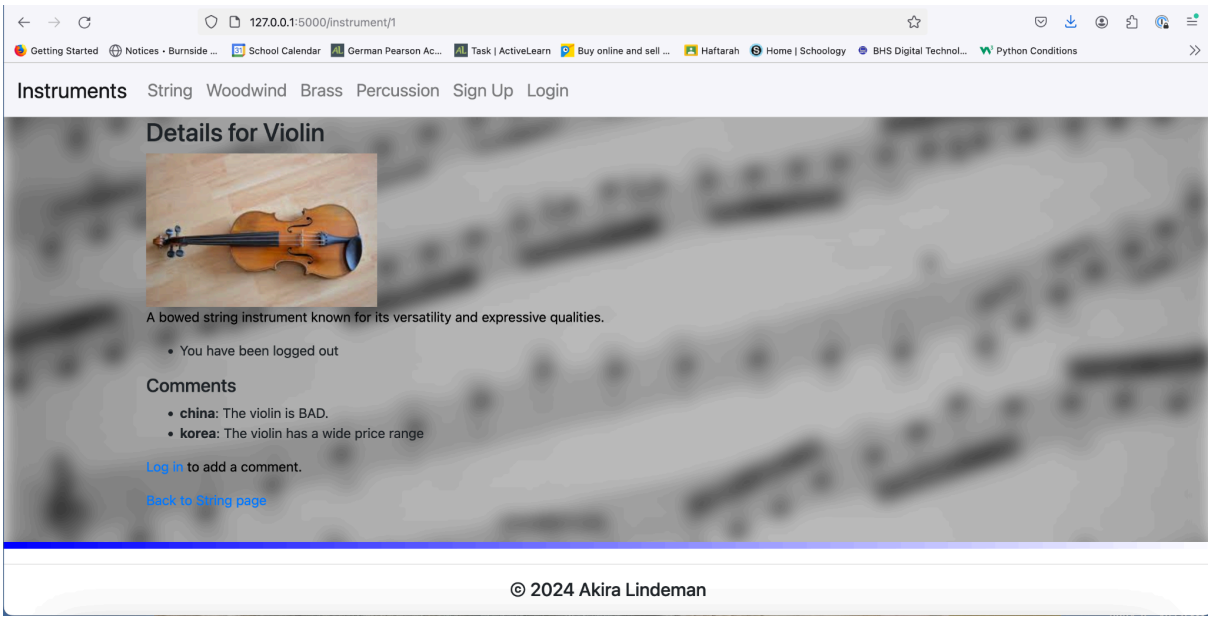
One thing that I had been struggling to add was images to my database. This was basically because I didn't know where to start with something that is not a string. After some research and speaking to my teacher, I had concluded that the best way to add images was to download the image and put the path to that image in my database. For example, if I wanted to add an image of a violin then I would add '/static/pictures/violin.jpeg' to the database. This is because the violin image is called violin.jpeg and is in the pictures folder which is in the static folder. This way I can write this in my html to display it:

```
<img src="{{ instrument[4] }}" alt="Image of {{ instrument[1] }}">
```

This works because in my SQL query I retrieve everything from an instrument:

```
# Fetch the instrument details
instrument_query = "SELECT * FROM Instrument WHERE id = ?"
```

And then I can take the thing at index 4 of the instrument table, which is the images column, which can then be used as a source to display an image.



Add sql_queries function to reduce repeated code

I decided to add this function to reduce repeated code as there are some lines of code that I use every time I use an SQL query such as:

```
result = cursor.fetchone()
connection.close()
return result
```

```python
def sql_queries(query, params, option):
    connection = sqlite3.connect('instruments.db')
    cursor = connection.cursor()
    cursor.execute(query, params)
    if option == 'fetchone':
        result = cursor.fetchone()
        connection.close()
        return result
    elif option == 'fetchall':
        result = cursor.fetchall()
        connection.close()
        return result
    elif option == 'commit':
        connection.commit()
        connection.close()
```

It first establishes a connection to the database (instruments.db) and creates a cursor to execute the provided SQL query with parameters to prevent SQL injection. Depending on the option argument, it can retrieve a single row (fetchone), multiple rows (fetchall), or commit changes to the database (commit). After executing the query, the function ensures the connection is closed to free up resources. It returns the result of the query if it's a fetch operation, otherwise commits changes when applicable.

This can be used, for example, when displaying individual instrument details.

```python
# Individual instrument details page.
@app.route('/instrument/<int:instrument_id>')
def instrument_details(instrument_id):
    # Fetch the instrument details
    instrument_query = "SELECT * FROM Instrument WHERE id = ?"
    instrument = sql_queries(instrument_query, (instrument_id,), 'fetchone')
```

It takes the query and parameter which in this case is instrument_id and the option fetchone as we are only fetching one instrument (one row).

Add a deleting function for comments

Adding this function will give the user more flexibility and give them the option to delete a comment if for whatever reason they want to do so. It gives them control over personal data (management over their own content that they created), and error correction if they make a mistake in their comment.

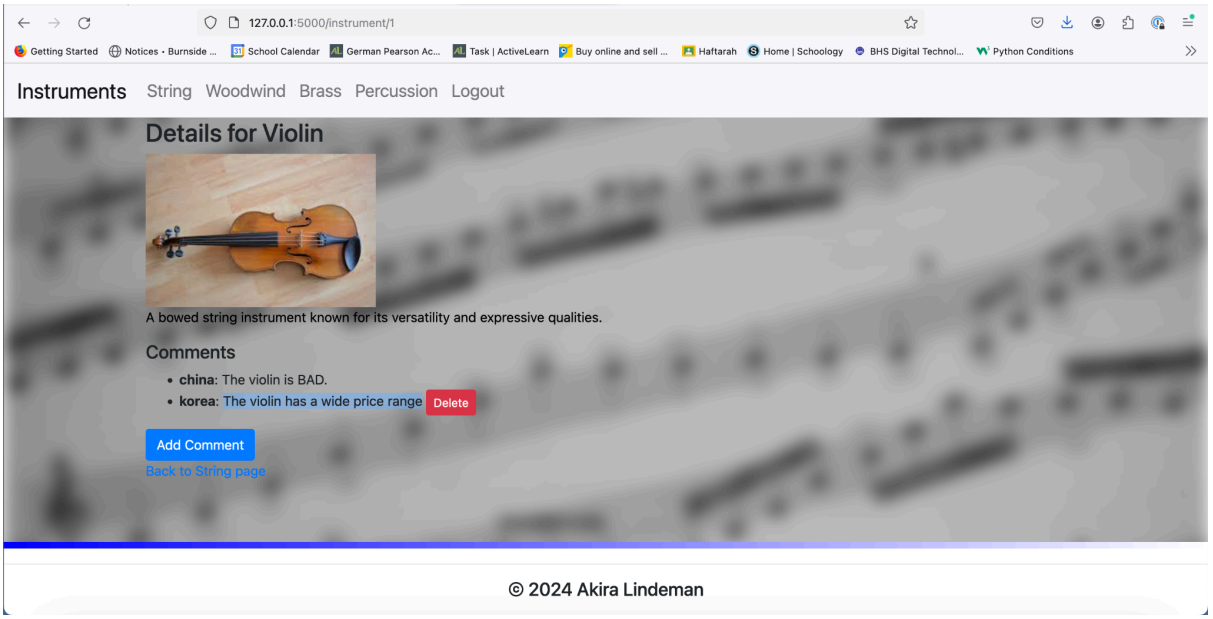
```python
@app.route('/delete_comment/<int:comment_id>/<int:instrument_id>', methods=['POST'])
def delete_comment(comment_id, instrument_id):
    user_id = session.get('user_id')

    if not user_id:
        flash('You need to be logged in to delete your comment', 'error')
        return redirect(url_for('login'))

    # Check if the comment belongs to the logged-in user
    query = "SELECT user_id FROM Comments WHERE id = ?"
    comment = sql_queries(query, (comment_id,), 'fetchone')

    if comment and comment[0] == user_id:
        try:
            delete_query = "DELETE FROM Comments WHERE id = ?"
            sql_queries(delete_query, (comment_id,), 'commit')
            flash("Comment deleted successfully.", "success")
        except Exception as e:
            flash(f"An error occurred: {e}", "error")

    else:
        flash('You can only delete you own comments', 'error')

    return redirect(url_for('instrument_details', instrument_id=instrument_id))
```

First, it retrieves the current user_id from the session. If the user isn't logged in, a flash message informs them they need to log in, and they are redirected to the login page. Next, the function checks whether the comment with the specified comment_id belongs to the logged-in user by querying the database. If the comment exists and the user IDs match, the function attempts to delete the comment using a DELETE SQL query inside a try-except block to handle potential errors. If the deletion is successful, a success message is flashed. If an error occurs, the exception is caught and displayed. If the comment doesn't belong to the user, an error message is shown. Finally, the user is redirected back to the instrument details page.

Key points:

- Session validation ensures the user is logged in.
- Ownership check confirms that only the comment's owner can delete it.
- The try-except block handles potential SQL errors during deletion.

For example, I am currently logged in as ‘korea’, and I added a comment saying, “The violin has a wide price range”. I now have the option to delete it.



There is another comment by ‘china’ saying that “The violin is BAD” but I (logged in as ‘korea’) do not have the option to delete ‘china’s’ comment.

```
<h2>Comments</h2>

{% if comments %}
<ul>
  {% for comment in comments %}
    <li>
      <strong>{{ comment[3] }}</strong>: {{ comment[2] }}
      {% if session.get('user_id') == comment[4] %}
      <form action="{{ url_for('delete_comment', comment_id=comment[0], instrument_id=instrument[0]) }}" method="post" style="display:inline;">
          <input type="submit" value="Delete" class="btn btn-danger btn-sm">
      </form>
      {% endif %}
    </li>
  {% endfor %}
</ul>
{% else %}
```

First, it checks if there are any comments to show using the {% if comments %} condition. If comments exist, it loops through them using a for loop, displaying each comment's content. The comment's username (comment[3]) is shown in bold, followed by the comment text (comment[2]). If the current logged-in user (retrieved from the session) matches the user who posted the comment (comment[4]), a delete button is displayed. This button submits a form that triggers the delete_comment function for the respective comment, allowing the user to delete their own comment. If no comments are available, the template does not render the list.

Add search bar to individual instrument pages

This is an addition to my plan as originally I did not intend to have a search bar, but a peer pointed out to me that it is hard to navigate through the instrument especially if there are 40 to a page. So I decided on a search bar so that the user can just look up a specific instrument themselves.

This form will be on every family page:

```
<!-- Search Form -->
<form method="GET" action="{{ url_for('string') }}">
  <input type="text" name="search" placeholder="Search for a string instrument">
  <button type="submit">Search</button>
</form>
```

Search Form (HTML):

- The <form> element has a method="GET" and an action attribute pointing to the string route ({{ url_for('string') }}). This means when the form is submitted, the input data is sent via the URL query parameters (e.g., ?search=term).
- The <input> field allows users to type in a search term. The name="search" attribute associates the input with the key search in the URL query string.
- The submit button triggers the form submission, which sends the search term to the backend route.

This is the python

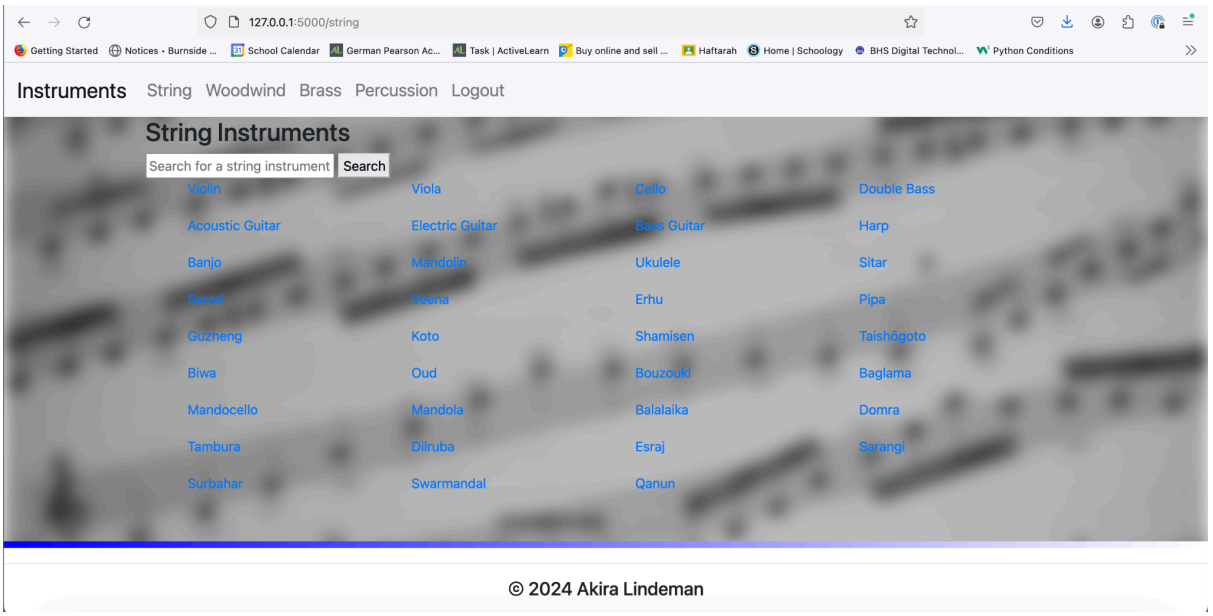
```
@app.route('/string')
def string():
    search_term = request.args.get('search')
    if search_term:
        query = "SELECT * FROM Instrument WHERE familyid = 1 AND name LIKE ?"
        params = ('%' + search_term + '%',)
    else:
        query = "SELECT * FROM Instrument WHERE familyid = 1"
        params = ()
    results = sql_queries(query, params, 'fetchall')
    return render_template("string.html", page='string', results=results)
```

The function begins by extracting the search term from the URL's query parameters using request.args.get('search'). This allows users to search for specific instruments by name. If there is a search term, the function constructs a SQL query to find instruments in the database that match the search term and belong to the string family (familyid = 1). The query uses a LIKE clause

with wildcard characters (%) to allow partial matches, and the search term is embedded within these wildcards to facilitate flexible searching.

If no search term is provided, the function defaults to a broader query that retrieves all string instruments without filtering. The results of the query, whether filtered or not, are obtained by calling the sql_queries function with the constructed SQL query and parameters. This function executes the query and returns the fetched data. Finally, the string() function renders the string.html template, passing the search results and a page identifier (page='string') to it. This template will then display the list of string instruments, potentially filtered by the user's search input.

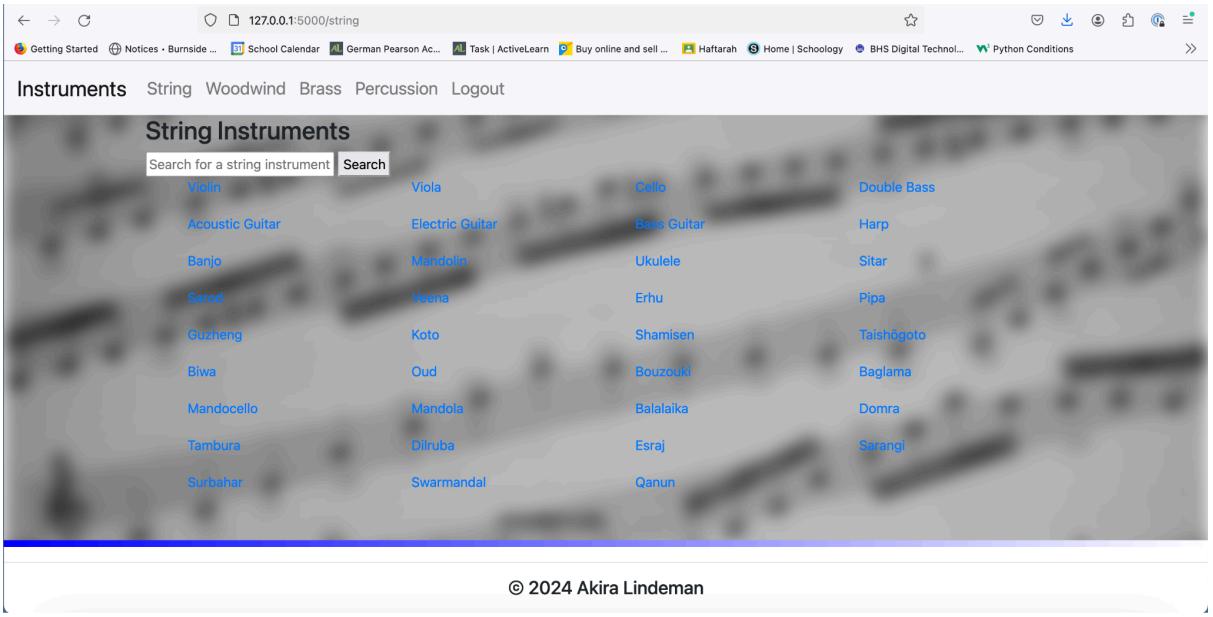
This is what this looks like:



Add more CSS

Blurred background and some

I added a blurred background of some sheet music as this is a musical website so I thought that it would fit. Also, it is black and white so it is easy to match, and sheet music is not specific to any instrument or family of instruments.



This is what is currently looks like on the string page and I will change the styling of the individual instrument buttons later to make them more visible.

Improved CSS for the nav bar

I wanted to make the nav bar look more professional and I did this by using CSS for specifically the nav bar.

```
<nav class="navbar navbar-expand-lg navbar-light bg-light fixed-top">
  <a class="navbar-brand" href="/string">Instruments</a>

  <div class="collapse navbar-collapse" id="navbarNav">
```

The Bootstrap classes in my **<nav>** element create a navigation bar with a light colour scheme that expands on larger screens and collapses into a hamburger menu on smaller ones. The **navbar-expand-lg** ensures this behaviour, while **navbar-light** and **bg-light** apply a light background with dark text. The **fixed-top** class makes the navbar stick to the top of the page as you scroll. Inside, the **navbar-brand** class styles the "Instruments" link as a brand element, and the **collapse navbar-collapse** with **id="navbarNav"** manages the collapsible portion of the navbar for smaller screens.

```css
.navbar {
    margin-bottom: 20px;
}

.navbar-brand {
    font-size: 1.6em;
}

.nav-item .nav-link {
    font-size: 1.3em;
    color: #555;
}

.nav-item .nav-link:hover {
    color: #000;
}
```

- .navbar:
    - margin-bottom: 20px;
    - This rule adds a bottom margin of 20 pixels to the navigation bar. This creates space between the navbar and the content below it, ensuring that the navbar doesn't sit too closely against the rest of the page content.
- .navbar-brand:
    - font-size: 1.6em;
    - This rule sets the font size of elements with the navbar-brand class to 1.6 times the size of the surrounding text.
- .nav-item .nav-link:
    - font-size: 1.3em;
    - color: #555;
    - These rules style the links inside navigation items. The font size is set to 1.3 times the surrounding text size, making the links slightly larger than the default text size for better readability. The colour #555 is a dark grey, providing a softer alternative to black for text colour.
- .nav-item .nav-link:hover:
    - color: #000;
    - This rule changes the text color of navigation links to black (#000) when a user hovers over them. This provides a visual cue to indicate that the link is interactive and can be clicked.

Grid added

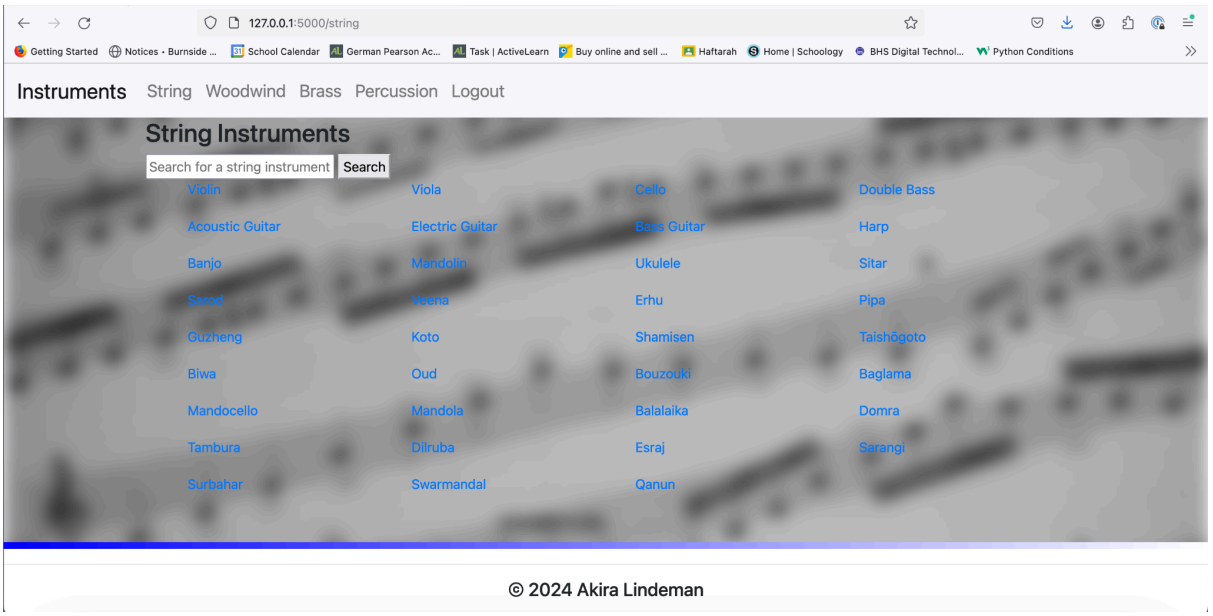
```css
body {
    display: grid;
    grid-template-rows: auto 1fr auto; /* Header, content, footer */
```

I implemented CSS Grid in the **'body'** and '.**container'** elements in my stylesheet. In the '**body'** element, the property '**display: grid**' along with '**grid-template-rows: auto 1fr auto;**' sets up a grid layout with three rows: one for the header, one for the main content, and one for the footer. This ensures that these sections are laid out in a vertical stack, with the header and footer taking up only as much space as needed, and the main content area expanding to fill the remaining space.

The '.**container'** class uses '**grid-row: 2**' to place itself in the middle row of this grid, which is designated for the main content.

The grid-based layout helps maintain an organized structure, ensuring that the header, content, and footer are properly aligned and spaced on the page.

```css
.container {
    grid-row: 2; /* This places the container in the middle row */
    padding: 20px;
}
```



- **Instrument List Layout**:

The instruments are neatly arranged in multiple columns. Each instrument name is in a specific "cell" of the grid, creating a balanced and uniform appearance.

- **Search Bar Alignment**:
  The search bar and text above it ("String Instruments") are centrally aligned.

- **Navigation Menu**:
  The horizontal navigation menu at the top (Instruments, String, Woodwind, Brass, Percussion, Logout) likely uses a grid to space the elements evenly across the top, giving a structured and clean look.

- **Footer Alignment**:
  The footer text ("© 2024 Akira Lindeman") is centred and spaced appropriately within the grid layout, ensuring that it does not overlap or disrupt other elements on the page.

Iteration 5

- Home page deleted
- Search bar added to nav bar
- Improved CSS for:
  - Signup and login
  - Individual instrument buttons
- Nav bar changes depending on if you are logged in or not
- Error pages are added
  - 404
  - 500
  - 414
- Changing the comment validation system so I don’t have to hack into the database and change the status

Added routes/pages/functions:

Page	**Route**	**Function**
404 page	@app.errorhandler(404)	def page_not_found(e):
414 page	@app.errorhandler(414)	def request_uri_too_long(e):
500 page	@app.errorhandler(500)	def internal_server_error(error):

Search page	@app.route('/search')	def search():

At this point most of my function were written and mostly working, so besides some minor changes, I didn't edit them much. In this iteration I focused on CSS and some error detection by adding the error pages.

A home page is very standard for websites but I didn't really see the need in having one as the user can just be directed to the string page as they can access everything through the nav bar anyways. All of the redirects, for example after logging in, will go to the string page.

Improved CSS

Bootstrap was used in my website to structure and style the navigation bar, ensuring responsiveness and a consistent layout across devices. The **navbar** class from Bootstrap provides essential styling, making the navigation bar mobile-friendly, while the **navbar-expand-lg** class ensures that the navbar expands on larger screens but collapses into a toggleable menu on smaller devices. The **navbar-light** and **bg-light** classes apply a light color scheme, providing a clean, modern look with light background colors.

Here's an example from my code: **<nav class="navbar navbar-expand-lg navbar-light bg-light fixed-top">**, which defines the structure and position of the navbar.

Within the navigation bar, the links to different instrument families and actions like login and sign-up use the **nav-item** and **nav-link** classes, which help style the links with appropriate spacing, font sizing, and hover effects.
For instance, the code **<li class="nav-item">String
** shows how Bootstrap is applied to individual links, making them neatly aligned and visually consistent. Additionally, the form inside the navbar uses the **form-inline** class, which ensures that the search bar and button are displayed inline for better alignment and spacing, as seen here: **<form class="form-inline ml-auto" action="/search" method="GET">**.

Signup and Login pages

I will use the same styling for the signup and login pages as this is standard.

```css
.login-page .background, .signup-page .background {
    /* background: #fff; */
    border-radius: 50px;
    padding: 20px;
    width: 100%;
    max-width: 390px;
    text-align: center;
}

.login-page h2, .signup-page h2 {
    font-size: 28px;
    margin-bottom: 20px;
    color: #333;
}

.login-page label, .signup-page label {
    display: block;
    margin-bottom: 4px;
    text-align: left;
    font-weight: bold;
    color: #555;
}

.login-page input[type="text"],
.login-page input[type="password"],
.signup-page input[type="text"],
.signup-page input[type="password"] {
    width: 100%;
    padding: 10px;
    margin-bottom: 15px;
    border: 1px solid #ccc;
    border-radius: 4px;
    box-sizing: border-box;
}

.login-page input[type="submit"],
.signup-page input[type="submit"] {
    width: 100%;
    padding: 10px;
    background-color: #14e006;
    border: none;
    border-radius: 4px;
    color: white;
    font-size: 16px;
    cursor: pointer;
}

.login-page input[type="submit"]:hover,
.signup-page input[type="submit"]:hover {
    background-color: #00b31b;
}

.login-page p, .signup-page p {
    margin-top: 15px;
    font-size: 14px;
}

.login-page a, .signup-page a {
    color: #007bff;
    text-decoration: none;
}

.login-page a:hover, .signup-page a:hover {
    text-decoration: underline;
}
```

```css
.login-page .background, .signup-page .background {
    background: #fff;
    border-radius: 10px;
    padding: 20px;
    width: 100%;
    max-width: 390px;
    text-align: center;
}
```

.login-page .background, .signup-page .background

This targets elements with the class **.background** inside the **.login-page** and **.signup-page** classes.

- **background: #fff;**
  Adds a white background to make the text more readable

```css
.login-page label, .signup-page label {
    display: block;
    margin-bottom: 4px;
    text-align: left;
    font-weight: bold;
}
```

.login-page label, .signup-page label

Targets **label** elements inside **.login-page** and **.signup-page.**

- **display: block;**
  Ensures the label behaves like a block element, taking up the full width available.
- **margin-bottom: 4px;**
  Adds 4px of space below the label.
- **text-align: left;**
  Aligns the label text to the left.
- **font-weight: bold;**
  Makes the label text bold.

```css
.login-page input[type="text"],
.login-page input[type="password"],
.signup-page input[type="text"],
.signup-page input[type="password"] {
    width: 100%;
    padding: 10px;
    margin-bottom: 15px;
    border: 1px solid #ccc;
    border-radius: 4px;
    box-sizing: border-box;
}
```

.login-page input[type="text"],
.login-page input[type="password"],
.signup-page input[type="text"],
.signup-page input[type="password"]

This targets **text** and **password** input fields inside **.login-page** and **.signup-page.**

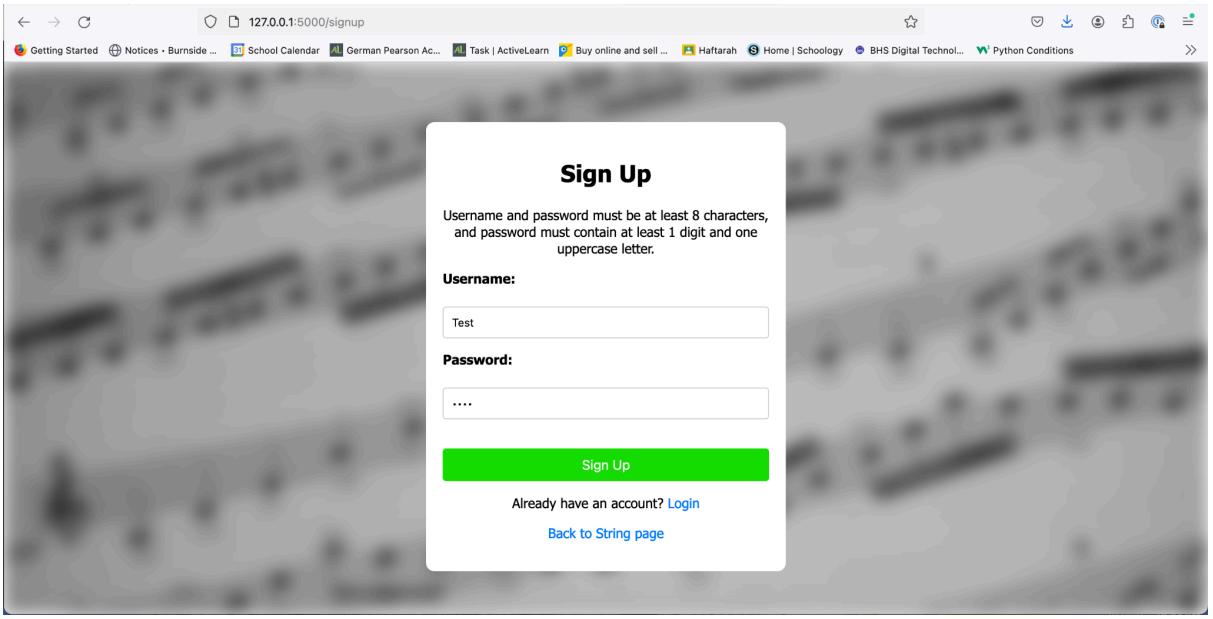
- **width: 100%;**
  Makes the input fields stretch to 100% of their container's width.
- **padding: 10px;**
  Adds 10px of padding inside the input fields.
- **margin-bottom: 15px;**
  Adds 15px of space below the input fields.
- **border: 1px solid #ccc;**
  Gives the input fields a 1px solid border with a light gray color.
- **border-radius: 4px;**
  Rounds the corners of the input fields slightly for a softer look.
- **box-sizing: border-box;**
  Ensures that the padding and border are included in the element's total width and height.

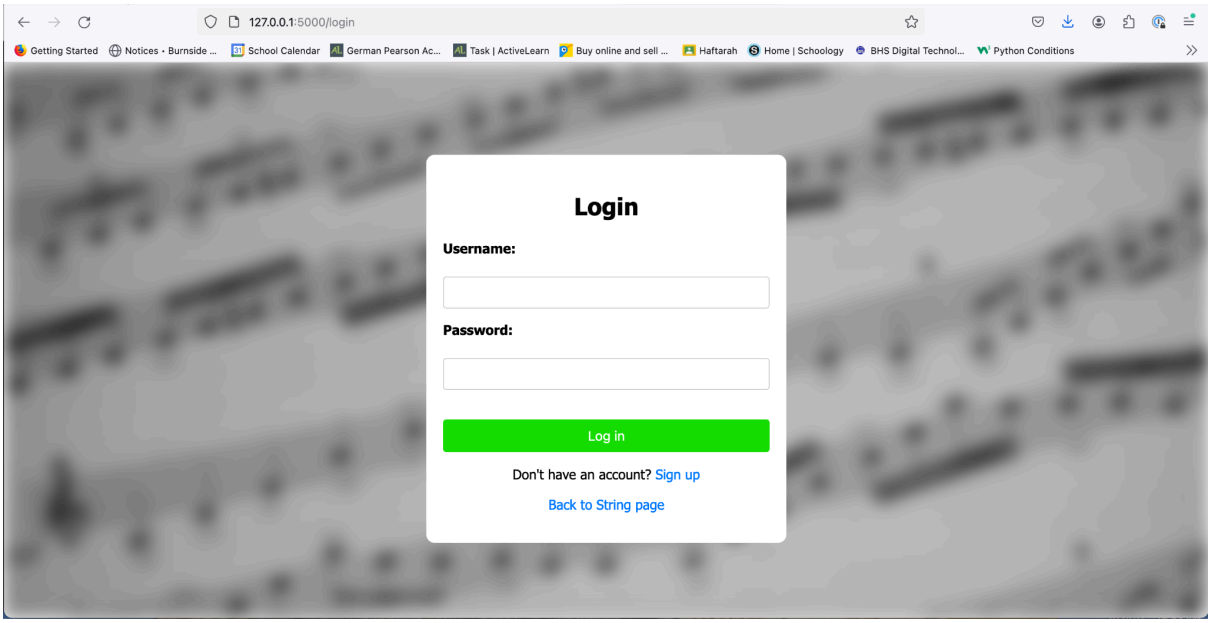
```css
.login-page input[type="submit"],
.signup-page input[type="submit"] {
    width: 100%;
    padding: 10px;
    background-color: #14e006;
    border: none;
    border-radius: 4px;
    color: white;
    font-size: 16px;
    cursor: pointer;
}
```

.login-page input[type="submit"],
.signup-page input[type="submit"]
This applies to **submit** buttons in both **.login-page** and **.signup-page.**

- **width: 100%;**
  Makes the submit buttons fill 100% of the container's width.
- **padding: 10px;**
  Adds 10px of padding inside the buttons.
- **background-color: #14e006;**
  Sets the background color of the submit buttons to a bright green.
- **border: none;**
  Removes any default borders from the submit buttons.
- **border-radius: 4px;**
  Slightly rounds the corners of the submit buttons.
- **color: white;**
  Makes the text color white.
- **font-size: 16px;**
  Sets the font size of the button text to 16px.
- **cursor: pointer;**
  Changes the cursor to a pointer when hovering over the buttons, indicating it's clickable.

This all looks like this:





Individual instrument buttons

I wanted to change the instrument buttons so they don't just look like links but actual buttons. I will do this by making the buttons have a radius that is a different colour so that they stand out from the background, and have a hover styling.

```css
/* Instrument Name Styles */
.instrument-list a {
    display: block;
    font-size: 18px;
    font-weight: bold;
    color: #ffffff;
    background-color: rgba(0, 0, 0, 0.5); /* Adds a semi-transparent dark background */
    padding: 10px;
    border-radius: 5px;
    text-align: center;
}

.instrument-list a:hover {
    background-color: rgba(0, 0, 0, 0.9); /* Darkens the background on hover */
    transform: scale(1.03); /* Slightly enlarges the text on hover */
}

.instrument-list {
    list-style-type: none;
    padding-left: 50px;
    margin: 0;
    display: grid;
    grid-template-columns: repeat(auto-fill, minmax(200px, 1fr));
    gap: 20px;
  }

.instrument-item {
    border-radius: 5px;
    padding: 15px;
    text-align: center;
}
```

```css
.instrument-list a {
    display: block;
    font-size: 18px;
    font-weight: bold;
    color: #ffffff;
    background-color: rgba(0, 0, 0, 0.5); /* Adds a semi-transparent dark background */
    padding: 10px;
    border-radius: 5px;
    text-align: center;
}
```

.instrument-list a

This section styles the individual instrument buttons (which are links).

- **display: block**: Ensures each button (link) takes up the full width of its container.
- **font-size: 18px**: Sets the text size to 18px, making it larger and easier to read.
- **font-weight: bold**: Makes the text bold for better visibility.
- **color: #ffffff**: The text color is white to contrast against the dark background.
- **background-color: rgba(0, 0, 0, 0.5)**: The background is a semi-transparent dark color to make the text more readable against various backgrounds.
- **padding: 10px**: Adds space around the text inside the button, giving it a padded, clickable area.

- **border-radius: 5px**: Rounds the corners of the button for a softer appearance.
- **text-align: center**: Centers the text within the button for consistent alignment.

```
.instrument-list a:hover {
    background-color: rgba(0, 0, 0, 0.9); /* Darkens the background on hover */
    transform: scale(1.03); /* Slightly enlarges the text on hover */
}
```

.instrument-list a:hover

Styles the buttons when hovered over by the user.

- **background-color: rgba(0, 0, 0, 0.9)**: Darkens the button's background when hovered, providing visual feedback.
- **transform: scale(1.03)**: Slightly increases the button size (by 3%) on hover, making it feel interactive.

```
.instrument-list {
    list-style-type: none;
    padding-left: 50px;
    margin: 0;
    display: grid;
    grid-template-columns: repeat(auto-fill, minmax(200px, 1fr));
    gap: 20px;
}
```

.instrument-list

- **list-style-type: none**: Removes the default bullet points for the list.
- **padding-left: 50px**: Adds padding on the left side, giving some space from the edge.
- **display: grid**: Uses a grid layout to organize the instrument buttons.
- **grid-template-columns: repeat(auto-fill, minmax(200px, 1fr))**: Creates flexible columns for the buttons. Each button will take at least 200px, and the grid will automatically adjust the number of columns based on available space.
- **gap: 20px**: Adds spacing between the buttons to prevent them from being too close to each other.

Nav bar changes depending on if you are loggin in or not

Initially I had a homepage where the user would first get directed to when they visiteed by website and they could login or signup, but I didn't want to make a different page if you were logged in as it wouldnt make sense for someone who is already logged in to have access to a signup button. I also wanted a user to be able to access the instruments and data even if they weren't logged in, so for me it made sense to just get rid of the home page (as it served no function) and just change the navbar according to they're login status.

I will do this by having an if statement in the html for the navbar which will have the extra buttons.

```
<!-- Defines a container for the navigation links,
applies basic Bootstrap navbar styling, Makes the navbar responsive,
expanding it on large screens,
Uses a light color scheme with light background colors, fixes to top -->
<nav class="navbar navbar-expand-lg navbar-light bg-light fixed-top">
  <a class="navbar-brand" href="/string">Instruments</a>

  <div class="collapse navbar-collapse" id="navbarNav">
    <ul class="navbar-nav">
      <li class="nav-item"><a class="nav-link" href="/string">String</a></li>
      <li class="nav-item"><a class="nav-link" href="/woodwind">Woodwind</a></li>
      <li class="nav-item"><a class="nav-link" href="/brass">Brass</a></li>
      <li class="nav-item"><a class="nav-link" href="/percussion">Percussion</a></li>

      {% if session.get('user_id') %}
        <li class="nav-item"><a class="nav-link" href="/logout">Logout</a></li>
      {% else %}
        <li class="nav-item"><a class="nav-link" href="/signup">Sign Up</a></li>
        <li class="nav-item"><a class="nav-link" href="/login">Login</a></li>
      {% endif %}
    </ul>

    <!-- Add search bar here -->
    <form method="GET" action="{{ url_for('search') }}" class="instrument-search-form"></form>
    <form class="form-inline ml-auto" action="/search" method="GET">
      <input class="form-control mr-sm-2" type="search" name="search" placeholder="Search for an instrument" aria-label="Search" required>
      <button class="btn btn-outline-success my-2 my-sm-0" type="submit">Search</button>
    </form>

  </div>
</nav>
```

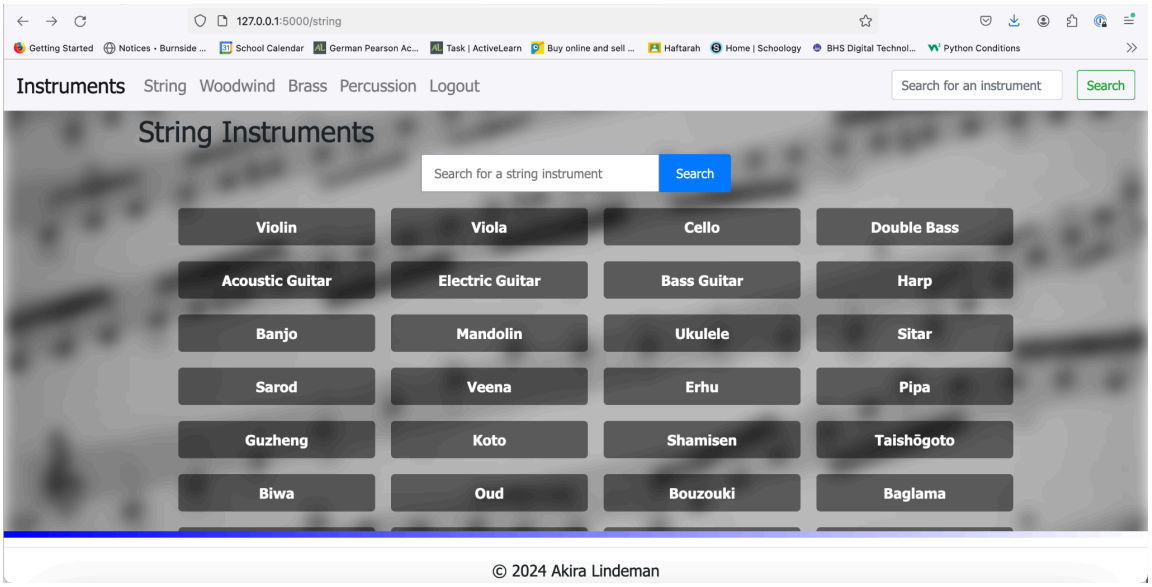
```
<li class="nav-item"><a class="nav-link" href="/string">String</a></li>
<li class="nav-item"><a class="nav-link" href="/woodwind">Woodwind</a></li>
<li class="nav-item"><a class="nav-link" href="/brass">Brass</a></li>
<li class="nav-item"><a class="nav-link" href="/percussion">Percussion</a></li>
```

The buttons for the string, woodwind, brass, and percussion pages are always there as I always want users to be able to access these pages and information.

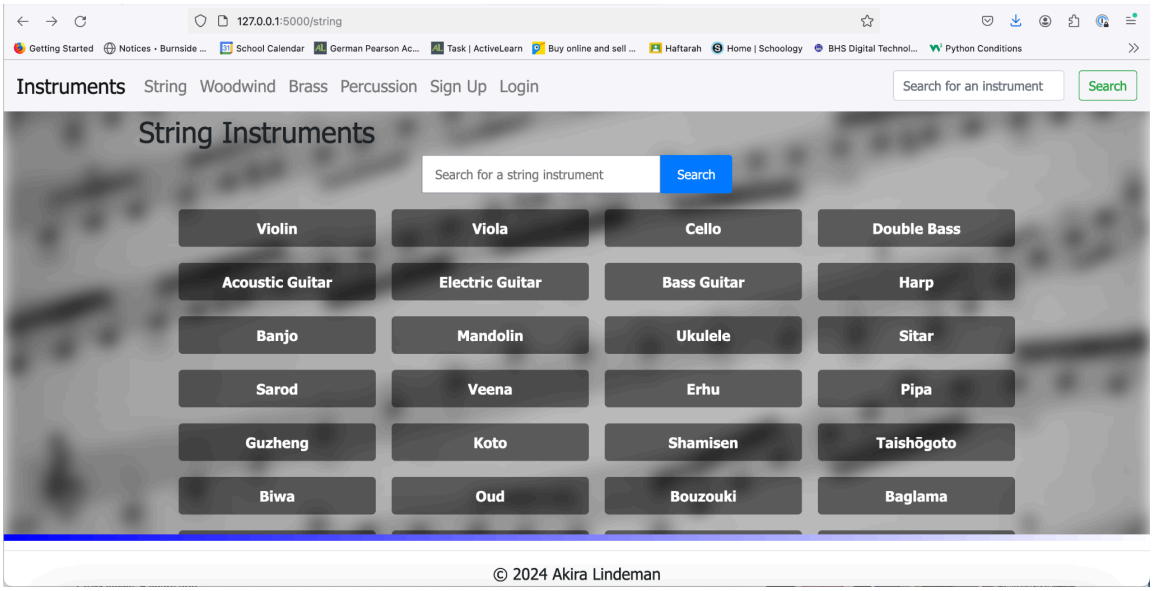
```
{% if session.get('user_id') %}
  <li class="nav-item"><a class="nav-link" href="/logout">Logout</a></li>
{% else %}
  <li class="nav-item"><a class="nav-link" href="/signup">Sign Up</a></li>
  <li class="nav-item"><a class="nav-link" href="/login">Login</a></li>
{% endif %}
```

This if statement checks whether a user is logged in or not by checking if user_id is in session as this is what happens when the user *is* logged in.
If they are logged in then I want them to have the option to logout.
In all other scenarios (session does not contain user_id thus the user is not in session) they have access to the signup or login buttons.

Logged in user (user_id in session)



Logged out user (user_id not in session)



Adding error page

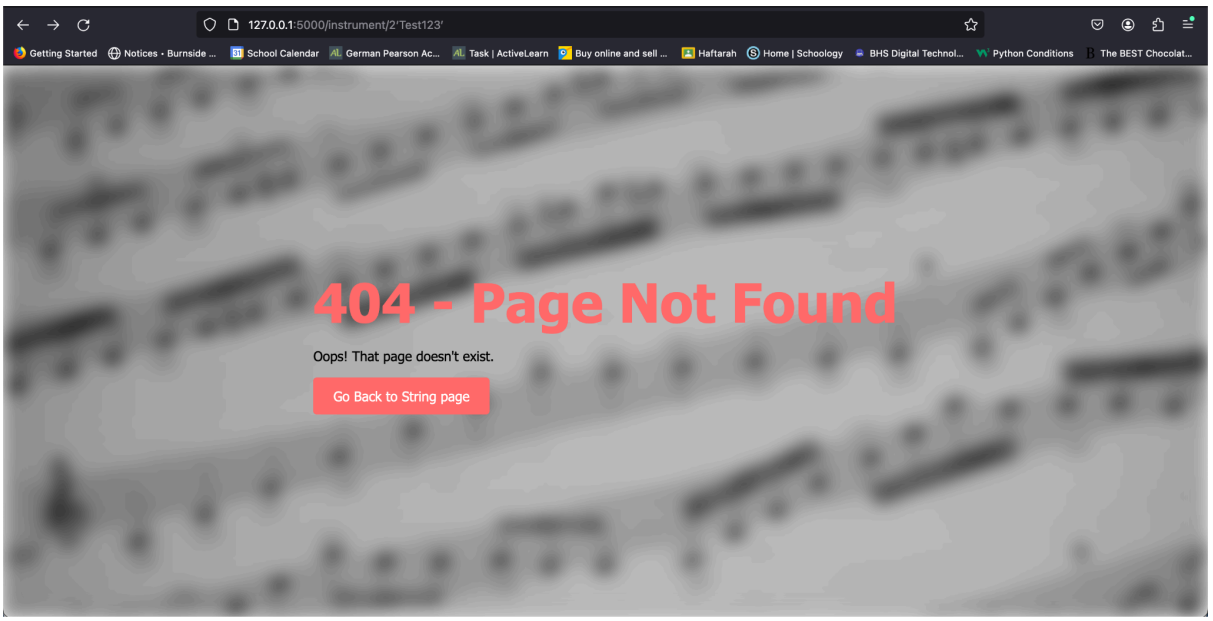
I need to add errors 404, 414, and 500 pages so for error protection.

The purpose of an error 404 page is that if the user searches for something in the URL that doesn’t exist, then they will be redirected to the error 404 page which is a much more clean thing to deal with compared to the website breaking.

A 414 error page indicates that the URL requested by the user is too long for the server to handle. This error typically occurs when a URL exceeds the server's capacity, often due to excessively long query strings or improperly formatted requests. Having a custom 414 error page on a website serves several purposes. It enhances user experience by providing a clear explanation of the issue rather than a generic error message, helping users understand why their request failed. Additionally, a customized 414 page can guide users on how to shorten their URL or retry their request, offering useful instructions or a link back to a relevant page, rather than leaving them confused. This type of page also adds a professional touch to the website, ensuring consistency and polish in error handling across various scenarios.

A 500 error page indicates an internal server error, meaning something has gone wrong on the server, but the exact issue is unclear. Having a custom 500 error page is important for several reasons. First, it improves user experience by providing a more friendly and informative message, reassuring users that the error is being addressed rather than displaying a confusing or alarming technical error. It can also offer users a way to navigate back to the homepage or other important sections of the site, preventing frustration. Additionally, a custom 500 page maintains the website’s branding and style consistency, making the site appear more professional even when errors occur. Lastly, it allows the developer to log and diagnose server issues while ensuring that users are not left in the dark.

Example of error page (404)



The code:

```
@app.errorhandler(404)
def page_not_found(e):
    return render_template("404.html"), 404
```

@app.errorhandler(404): This is a Flask decorator that tells the app to trigger the following function whenever a 404 error occurs. A 404 error happens when a user tries to access a page or resource that doesn't exist on the server.

def page_not_found(e): This is the function that will run when the 404 error is triggered. The argument e refers to the error object (in this case, a 404 error).

return render_template("404.html"), 404:

- **render_template("404.html")**: This renders the **404.html** template, which is the custom HTML page the user will see when they land on a non-existent page.
- , **404**: This explicitly sends the **404 HTTP status code** back to the browser, ensuring that the browser and any search engines understand that this is a "Not Found" page.

The other error pages and functions work in very similar ways and as you can see the code is very similar:

```
@app.errorhandler(414)
def request_uri_too_long(e):
    return render_template('414.html'), 414


@app.errorhandler(500)
def internal_server_error(error):
    return render_template("500.html"), 500
```

Changing the comment validation system

At the moment when a user submits a comment, I (the administrator) have to go into the database and change the status of the comment from unchecked_comment to checked_comment. This is a bad way to do it so I will make a page on my website that only the administrator can access, and then change the columns in the comments table to so that there is one called comment_status that is either a 0 or 1. The comment will display if the comment_status is 1.

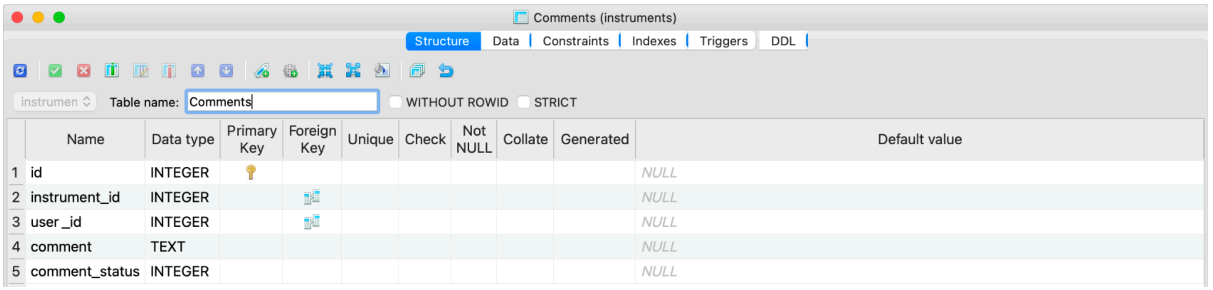
Iteration 6: Final Iteration - address to testing

This iteration focuses on the changes that I made from my testing and feedback.

- Adding an admin page to the website to approve comments
- Styling the individual instrument button with images in a ‘card’ format
- Styling error messages to be more prominent
- Implemented abort instead of render_template for error pages
- Maximum URL length
- All other changes made from testing

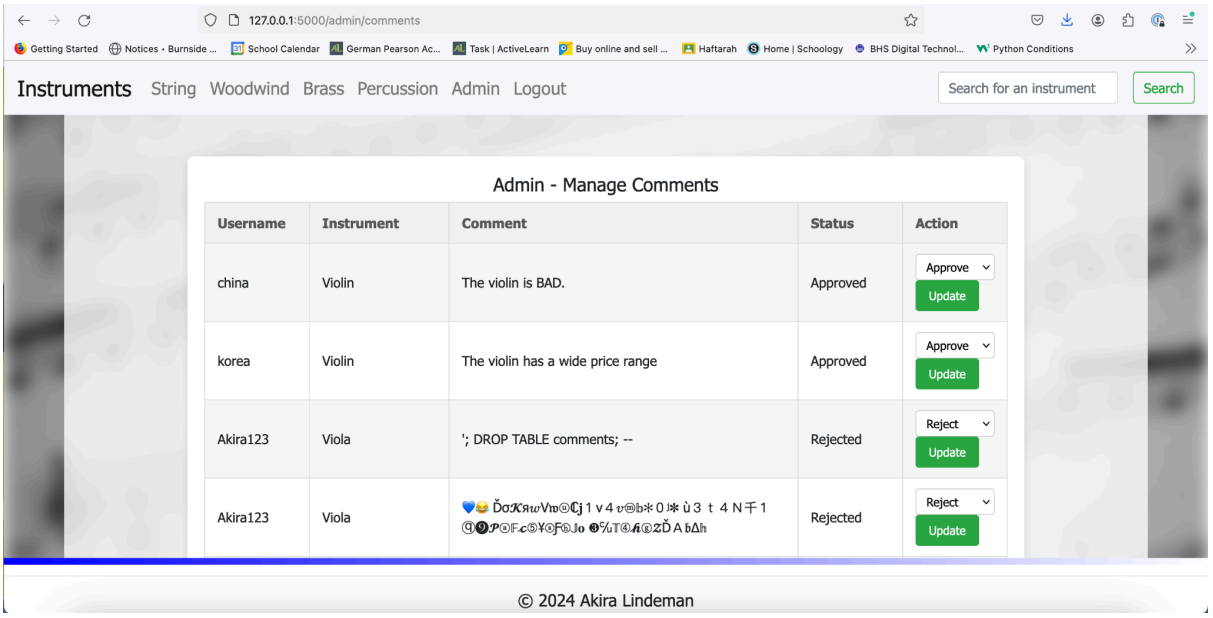
Adding an admin page to the website to approve comments

Previously, I had to hack into the database and manually copy the unchecked comment into the checked comment column and then it would display. This is not a good way to do it, so I changed the database so that it would have a column called comment_status which would either be 0 or 1, and the comment would display if the comment_status was 1.



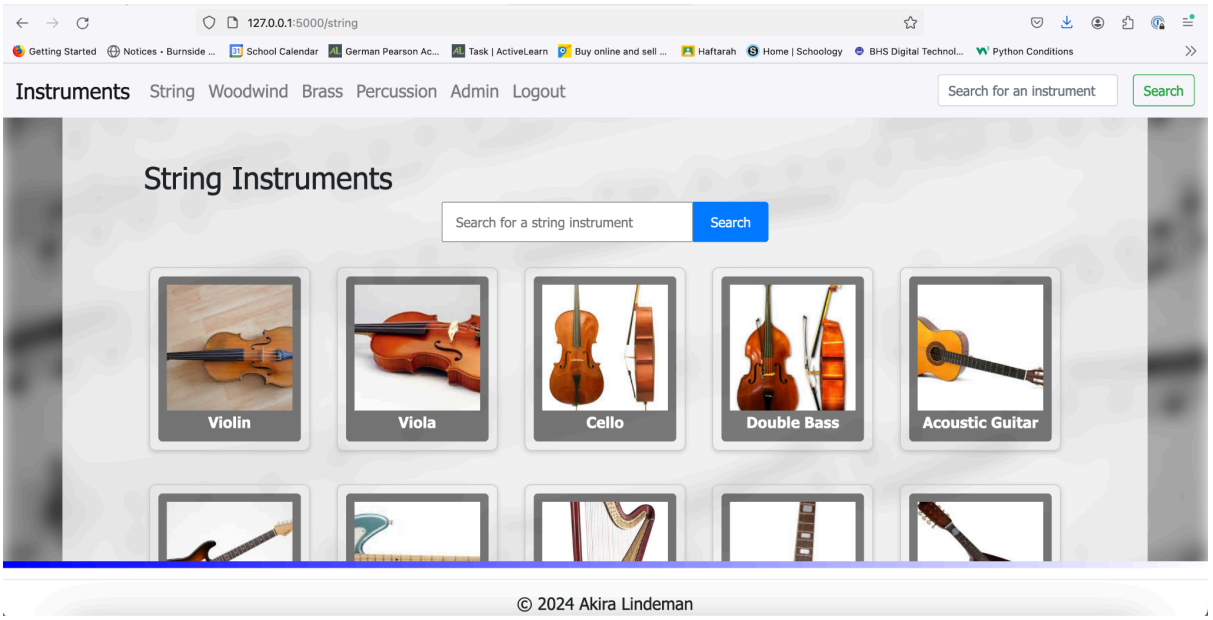
	Name	Data type	Primary Key	Foreign Key	Unique	Check	Not NULL	Collate	Generated	Default value
1	id	INTEGER								NULL
2	instrument_id	INTEGER								NULL
3	user_id	INTEGER								NULL
4	comment	TEXT								NULL
5	comment_status	INTEGER								NULL

I also added a new page for the admin to be able to change the statuses from the website instead of hacking into the database.



Styling the individual instrument button with images in a ‘card’ format

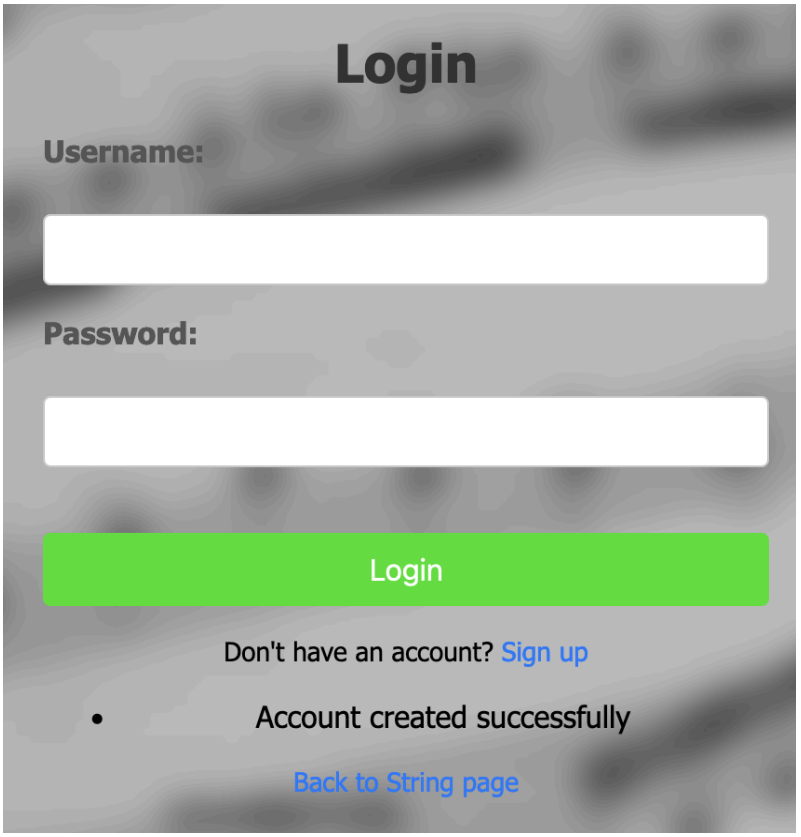
Previously, the individual instrument buttons were displayed in a grid with just the name of the instrument. After receiving some feedback, I thought that for someone who doesn’t know much about instruments, having just the names would be difficult for them to identify what it is. So, I decided to display the instruments in a grid using the ‘card’ format which many other websites follow.



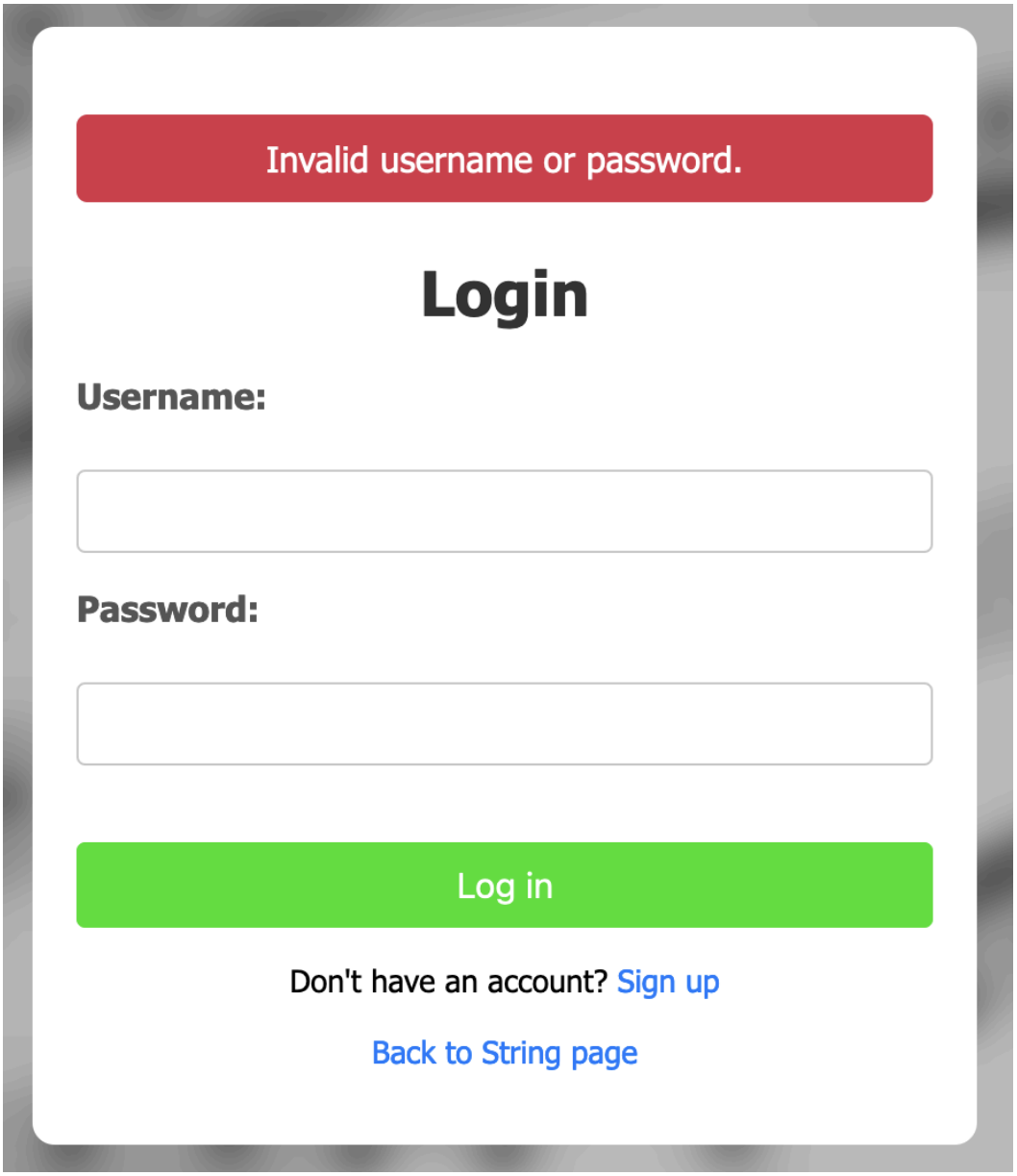
This was people will be able to identify instruments by sight as well as the name.

Styling error messages to be more prominent

I have had error messages through many of my iterations, but they had always just been black text towards the bottom of the page.
For example:



This is an issue because the user may not notice it, and therefore not know what happened or what was going on. To solve this, I styled the messages to appear at the top of the ‘body’ and have a coloured background depending on what type of message it was.
For example, and error message would have a red background and white text. This is impossible to miss.



Implemented abort instead of render_template for error pages

To make my website more robust, I changed the way that I handled errors from render_template('404.html') to abort(404).It standardises error handling, ensuring that the application automatically triggering custom error handlers. It indicates an actual error rather than a successful page rendering. Additionally,

abort can improve performance by halting further execution in the route, making it a more efficient way to manage 404 errors while leveraging Flask's built-in error management capabilities.
This was utilised for all of my error pages.

404 Error Handling:

```
@app.errorhandler(404)
def page_not_found(e):
    return render_template("404.html"), 404
```

- Instrument Details Route:
    - When checking for the existence of an instrument:

```
if not instrument_exists or not instrument_exists[0]:
    raise KeyError
```

The KeyError is caught and results in:

```
except KeyError:
    abort(404)
```

- Handling Missing Instruments:
    - After querying the instrument, you correctly check if the result is None, again using abort(404) to handle the case where the instrument ID does not correspond to any existing record.

- General Exception Handling:
    - I caught any unexpected exceptions and call abort(500):

```
except Exception as e:
    print(f"General Exception occurred: {e}")
    abort(500, description=str(e))
```

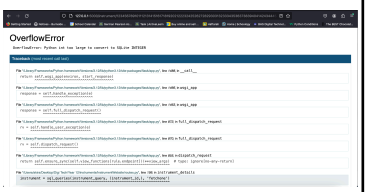
This sends users to my custom 500 error page, improving user experience during unexpected failures.

Custom Error Pages:
I've set up custom templates for 404, 414, and 500 errors, which enhances user experience by providing informative feedback when something goes wrong.

Security and Maintainability:
By using abort(), I also enhancing the security of your application by not revealing unnecessary information about the internal workings when an error occurs.

Maximum URL Length

One of the things that I tested was adding an obscenely longstring in the url after a route. For example, putting hundreds of characters after http://127.0.0.1:5000/instrument/1

What am I testing	How am I testing it	Expected result	Actual result	Did it work? (if not, what went wrong)
Input a very long string of numbers in the url to see if it handles it without breaking	Input the numbers from 1 to 1000 into the url after the slash	It should redirect to ther error 404 page as the page doesnt exist		It did not work. I will add  These will handle the errors depending on what error is raised.  Now it works

This worked for this case, but if i put thousands of characters after, it break:

This was a weird issue because I had the error handler 414 but it wasn't running properly.

```python
@app.errorhandler(414)
def request_uri_too_long(e):
    return render_template('414.html'), 414
```

```python
@app.route('/instrument/<int:instrument_id>')
def instrument_details(instrument_id):
    try:
        MAX_URL_LENGTH = 200  # Set your URL length limit

        # Check the length of the request path
        if len(request.path) > MAX_URL_LENGTH:
            abort(414)  # Abort if the URL length exceeds the limit
```

This seemed to be happening because of the specific browser's acceptable length.

So, I made a separate function for the MAX_URL_LENGTH. This way all pages will be able to handle long url lengths.

```python
@app.before_request
def limit_url_length():
    MAX_URL_LENGTH = 200  # Set your URL length limit
    if len(request.path) > MAX_URL_LENGTH:
        abort(414)
```

This function runs before every HTTP request, so if the url is too long, the code will not run. This also reduces repeated code because I don't have to put this code in every function.
MAX_URL_LENGTH is a special built-in variable that is constant and doesn't change during execution.

@app.before_request: This decorator registers the **limit_url_length** function to be called before handling each incoming HTTP request.

It checks the length of the request path (the part of the URL that comes after the domain) using **len(request.path)**.

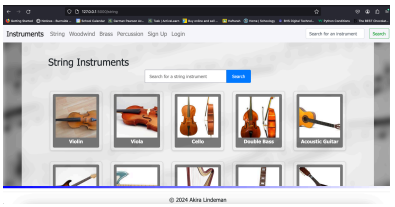
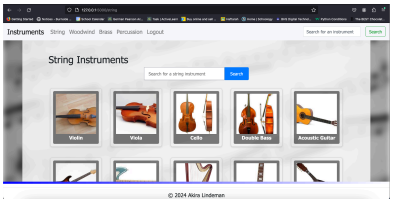
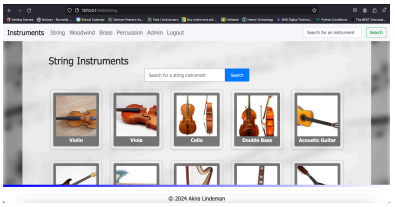
If the length of the request path is too long, the function calls **abort(414)**. This will return an HTTP status code of 414, which means "URI Too Long." and render the 414 function.

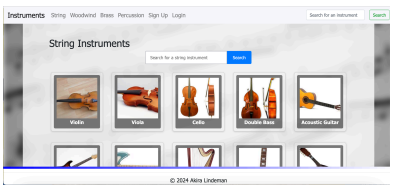
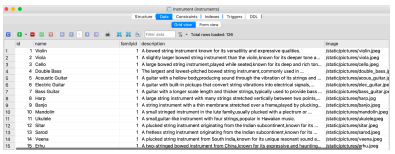
```
@app.errorhandler(414)
def request_uri_too_long(e):
    return render_template('414.html'), 414
```

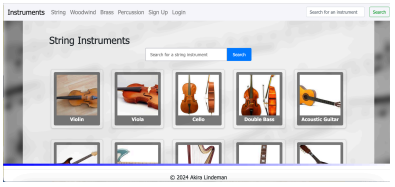
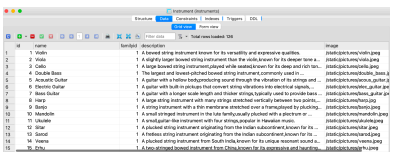
Testing

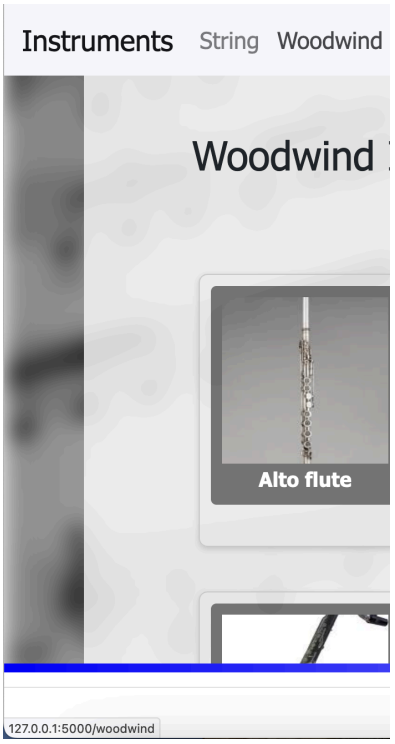
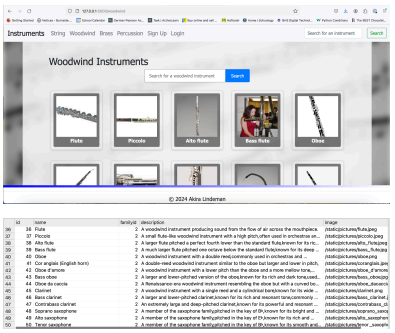
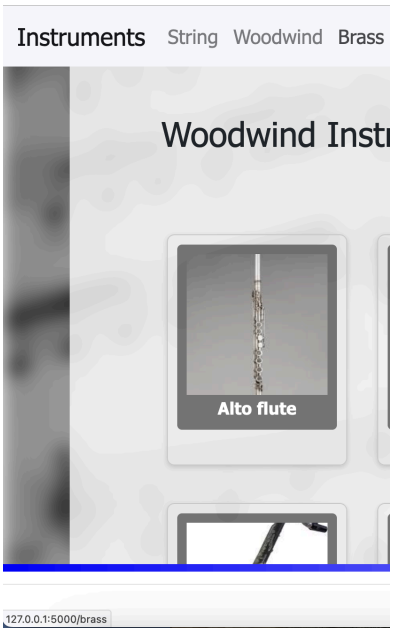
Nav Bar

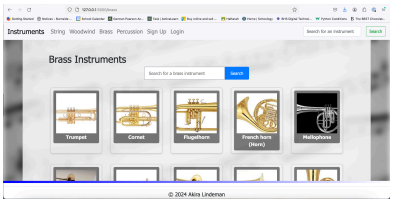
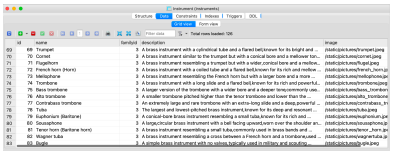
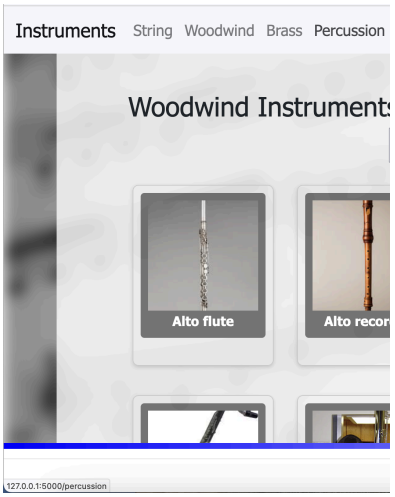
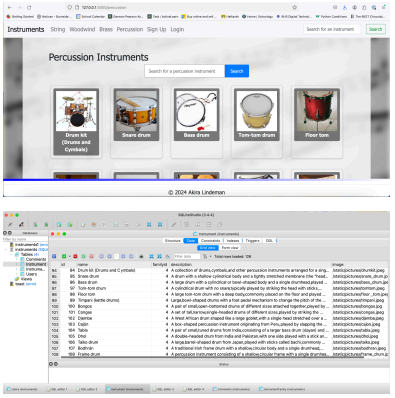
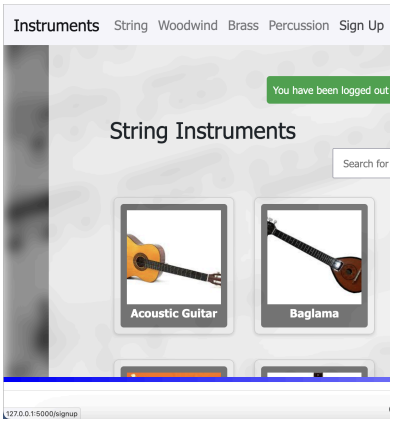
What am I testing	How am I testing it	Expected result	Actual result	Did it work? If not, what went wrong

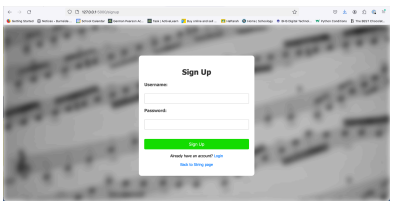
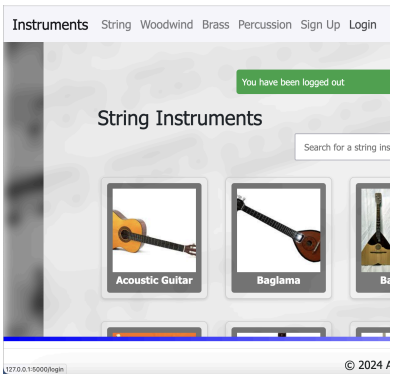
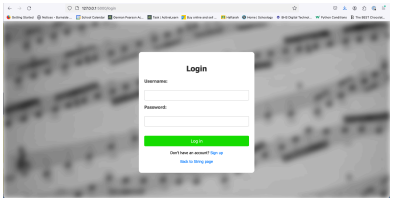
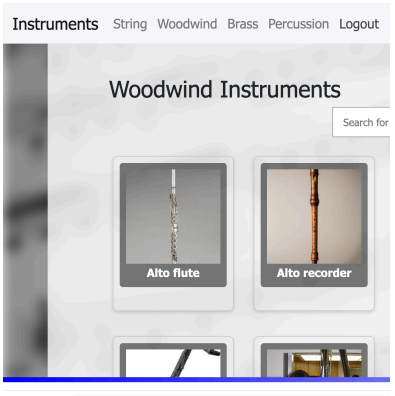
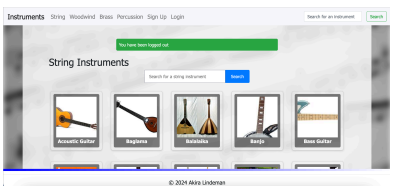
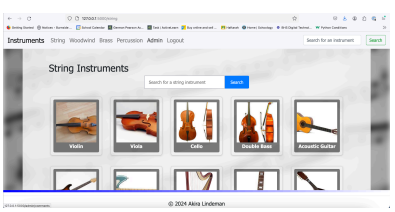
				and how will you fix it
Correct nav bar displays if you are logged out	Logged out and seeing the nav bar	The nav bar should have Instrument, String, Woodwind, Brass, Percussion, Search		Pass
Correct nav bar displays if you are logged in	Logged in and seeing the nav bar	The nav bar should have Instrument, String, Woodwind, Brass, Percussion, Logout, Search		Pass
Correct nav bar displays if you are logged in and admin	Logged in with an admin account and seeing the nav bar	The nav bar should have Instrument, String, Woodwind, Brass, Percussion, Admin, Logout, Search		Pass

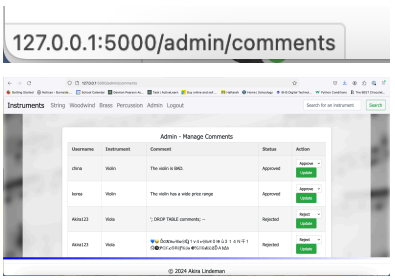
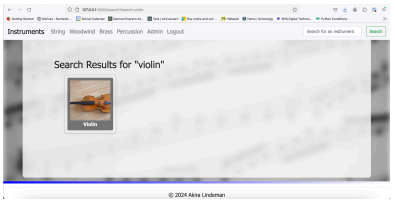
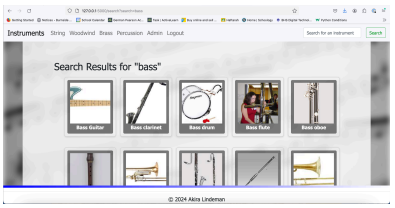
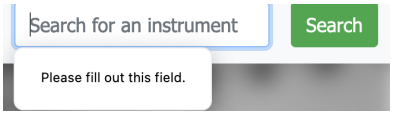
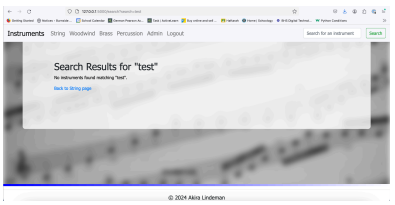
Instruments button on the top left of the nav bar works	Hovering to see the link, and clicking from a different page to see if it goes to the string page.	Shows the url: http://127.0.0.1:5000/string, and goes to the string page when clicked. Data integrity: This matches with the information in the database.	  	Pass

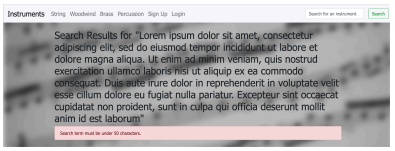
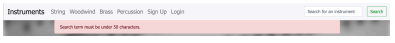
String button works	Hovering to see the link	Shows the url: http://127.0.0.1:5000/string and goes to the string page when clicked. Data integrity: This matches with the information in the database.	  	Pass

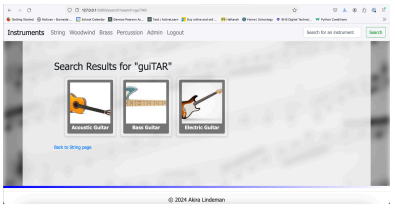
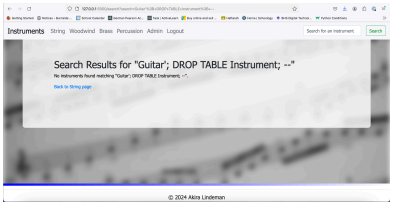
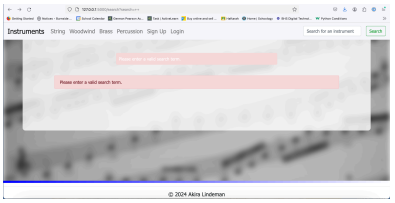
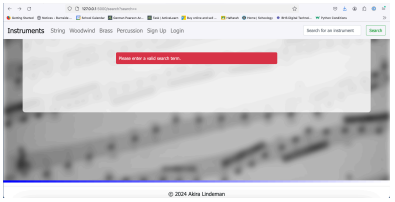
Woodwind button works	Hovering to see the link	Shows the url: http://127.0.0.1:5000/woodwind and goes to the woodwind page when clicked. Data integrity: This matches with the information in the database.	  	Pass
Brass button works	Hovering to see the link	Shows the url: http://127.0.0.1:5000/brass and goes to the brass page when clicked. Data integrity: This matches with the information in the		Pass

		database.	 	
Percussion button works	Hovering to see the link	Shows the url: http://127.0.0.1:5000/percussion and goes to the percussion page when clicked. Data integrity: This matches with the information in the database.	 	Pass
Signup button works	Hovering to see the link	Shows the url: http://127.0.0.1:5000/signup and goes to the signup page when clicked.		Pass

				
Login button works	Hovering to see the link	Shows the url: http://127.0.0.1:5000/login and goes to the login page when clicked.	 	Pass
Logout button works	Hovering to see the link, and after clicking, and seeing if the the flash message displays if userid is ‘popped’ from session	Shows the url: http://127.0.0.1:5000/logout And flash message	 	Pass
Admin button works	Hover over the button	Shows the url: http://127.0.0.1:5000/admin/comments and goes to the		Pass

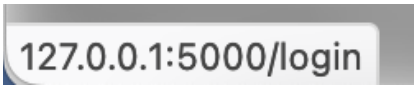
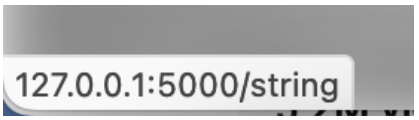
		admin page when clicked.		
Search bar works for valid inputs	Searching for an instrument and part of an instrument's name	The buttons for the instruments searched (or parts of the instruments searched) displays. Check if the buttons work.	 	Pass
Search bar doesn't do anything when you search for nothing	Click search button without entering anything	Page stays the same but a message displays as filling out the input field is mandatory		Pass
Search bar works for searches that arent instruments	Search "test" which is not an instrument and thus not in my database	Displays results for "test" which is none so displays message saying that no instrument was found for that input.		Pass

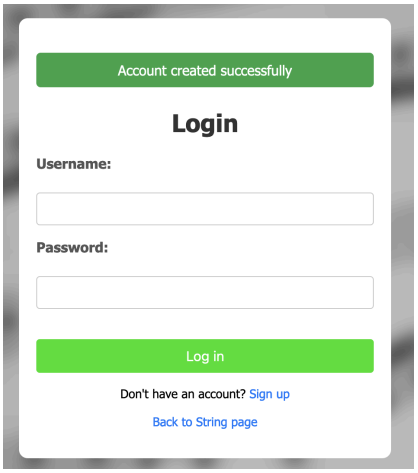
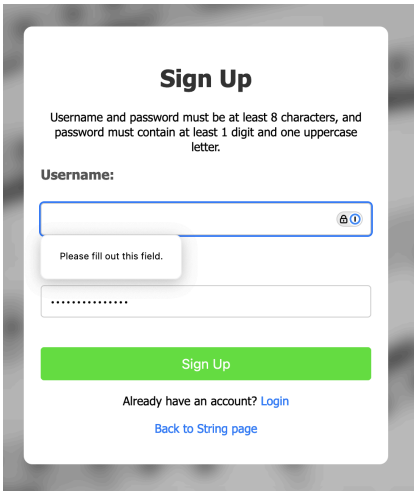
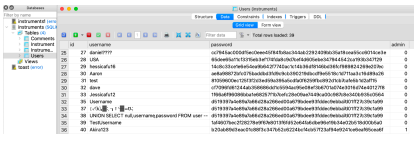
Search bar is able to deal with inputs that are too long	Input a paragraph of lorem ipsum	Displays an error message saying that the input was too long		In the case that the search term is too long I don't want it to display the search term again. I will add this line of code to empty the search_term variable if it is too long: search_term = None # Clear the search term to prevent it from being displayed I will also change the search_result.html so that it only displays the searched term if it is shorter than 50 characters: {% if search_term %} Search Results for "{{ search_term }}" {% endif %}
""	""	""		Pass I will change this so that I will have a physical limit on the maximum length by adding the maxlength in

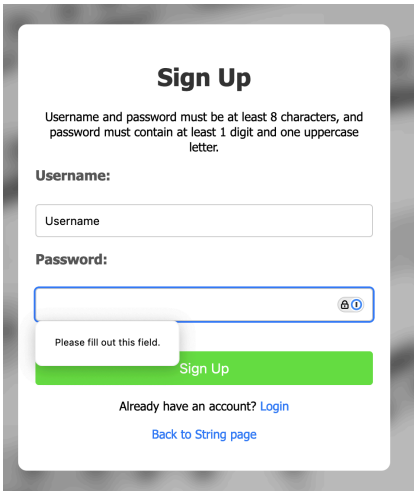
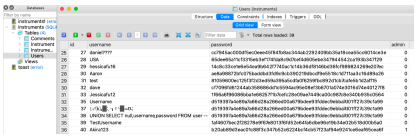
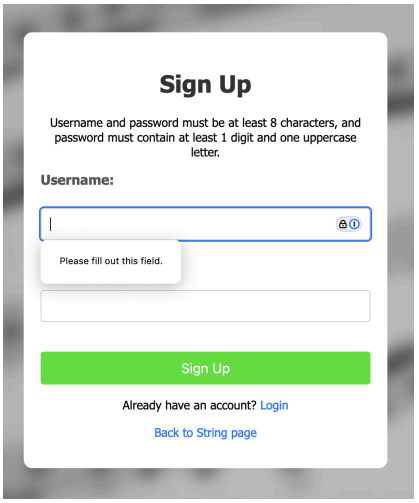
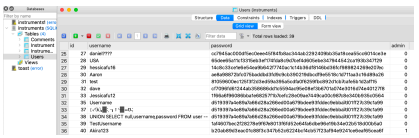
				the tag in the html  I could not input any more than 50 characters.
Can the search bar handle incorrect casing	Input ‘guiTAR’	All instruments with ‘guitar’ in their name should display		Pass
Can the search bar can handle sql injection	Enter a search term that could potentially be a SQL injection "Guitar'; DROP TABLE Instrument; --"	It should just display the message for the instrument not found		Pass, no effect on the database
Search a space	Enter a space in the search bar and then search	No instruments should display	 	The flash message is displayed twice. I will change the code slightly so that the the error message is only displayed in the **flash-messages div**. The main content

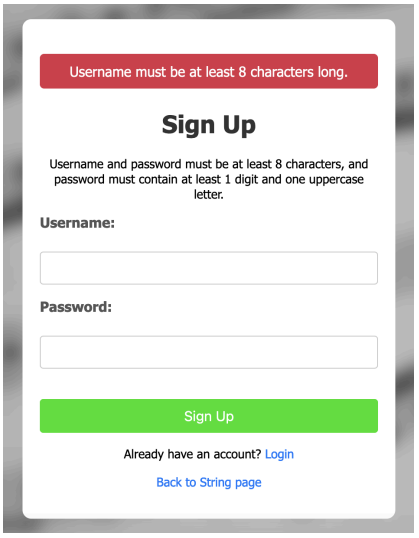
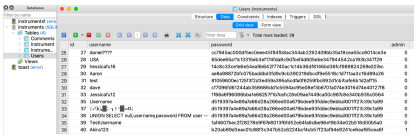
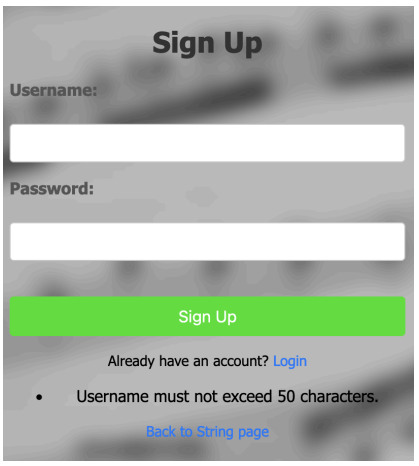
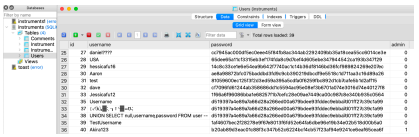
				will only render when there's no error message. This should fix the double display issue.

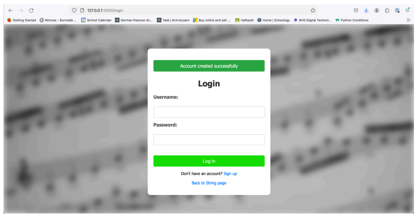
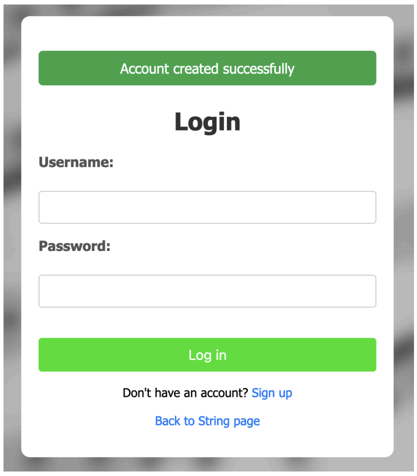
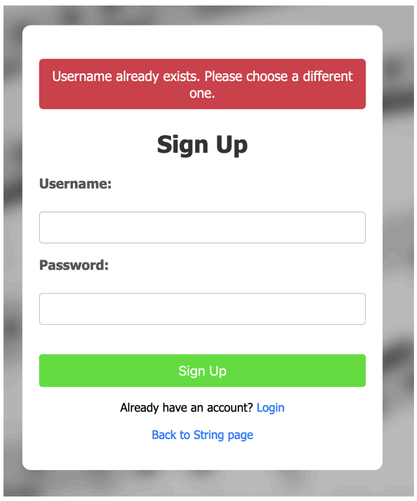
Signup page

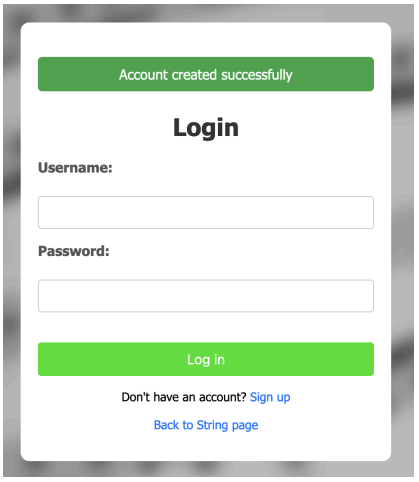
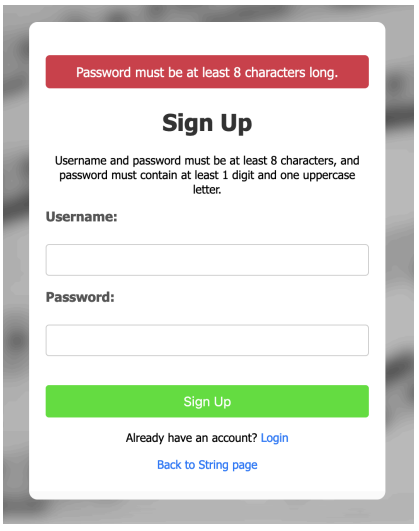
What am I testing	How am I testing it	Expected result	Actual result	Did it work? If not, what went wrong and how will you fix it
Does the Login button work	Hovering over the button to see the route	The url: http://127.0.0.1:5000/login should show		Pass
Does the ‘Back to string page’ button work	Hovering over the button to see the route	The url: http://127.0.0.1:5000/string should show		Pass

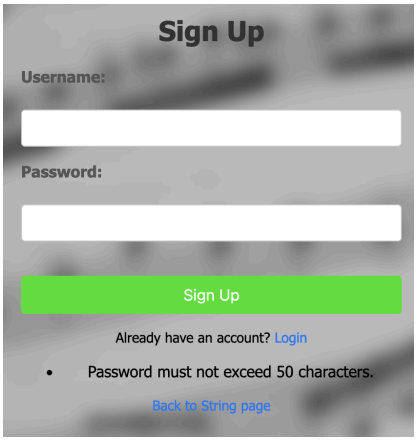
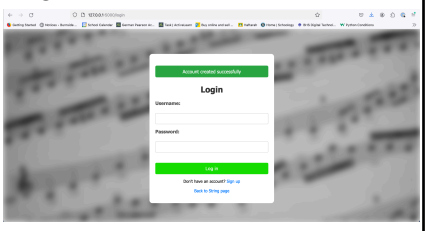
Valid input works, boundary testing; the minimum required length of the username (8)	Input “Username” for username and “TestPassword123” for password and click signup	Redirection to login page with flash message saying that the account was created succesfully	 Data integrity: Is it in the database? Yes The password is encoded so the administrator doesn't know	Pass
Empty username	Input no username but input a password and click signup	Stay on Signup page and message will display for not inputting anything for username	 Data integrity: No new user was added to the database 	Pass

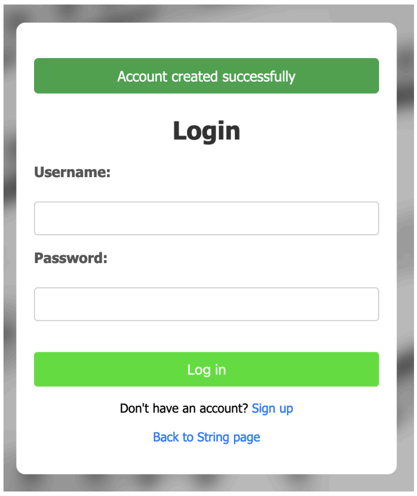
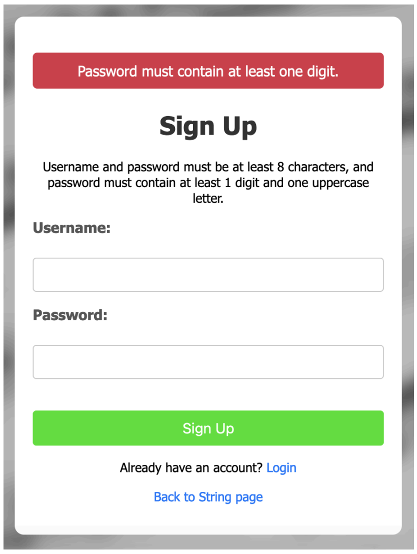
Empty Password	Input no password but input a username and click signup	Stay on Signup page and message will display for not inputting anything for password	 Data integrity: No new user was added to the database 	Pass
Both Empty	Input nothing and just click signup	Stay on Signup page and message will display for not inputting anything for username as that is the first thing that is required	 Data integrity: No new user was added to the database 	Pass

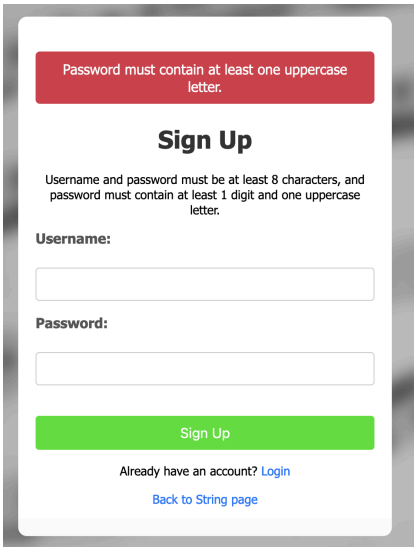
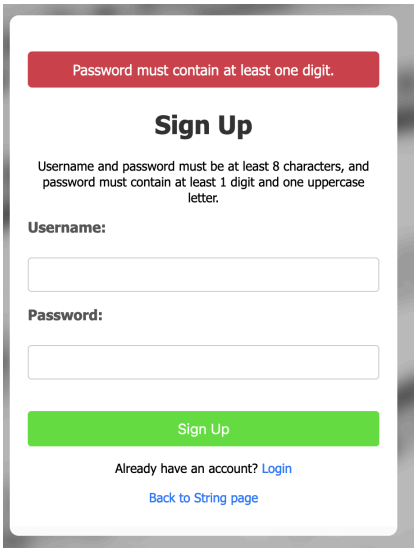
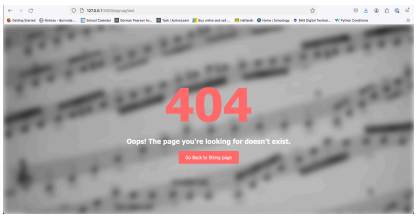
Boundary Testing: username parameters: a username below 8 characters (7)	Input “Test123” for username and “TestPassword123” for password and click signup	Stay on Signup page and message will display for the username having to be at least 8 characters long	 Data integrity: No new user was added to the database 	Pass
Boundary Testing: username parameters: a username above 50 characters	Input Lorem ipsum paragraph for username and “TestPassword123” for password and click signup	Stay on Signup page and message will display for the username having to be less than 51 characters long	 Data integrity: No new user was added to the database 	Pass
“” Modified	“”	“”	I made a physical boundary of 50 characters so that the user cannot input more than 50. So the account was created succesfully	Pass

			 Data integrity: New user was added to the database 	
Typing in special characters/Unicode.	Input “⁽↙(K)₂ʍ⁚┐ ⇡ᒻ□ ⇼ ಬ⁛.” in username and “TestPassword123” for password and click sign up	It should treat it like any other username, so it would redirect to the login page with the success mesage	 Data integrity: User was added to the database: 	Pass
Test username parameters: a username that is already taken	Input ‘????E2332’ (a already taken username) for username and “TestPassword123” for password and click signup	Stay on Signup page and message will display for the username being the same as another	 Data integrity: There is only one user with this username: 	Pass

SQL injection:	typing “ UNION SELECT null, username, password FROM user --” into a form. This will supposedly close off the quote marks from the query, execute the query “UNION SELECT null, username, password FROM user”. Then the “--” comments out the rest of the query.	It should treat it like a normal username	 (I am using the “?” tool which prevents sql injection.) "INSERT INTO Users (username, password) VALUES (?, ?)" Data integrity: is the user being added to the database? Yes 	Pass
Test password parameters: a password below 8 characters (7)	Type in “TestUsernam e” fro username and “Test123” for password	Stays on signup page and throws a flash message saying that the password must be at least 8 characters long	 Data integrity: The user is not present in the database	Pass

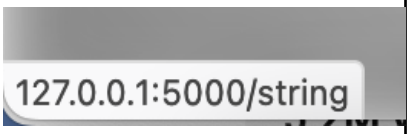
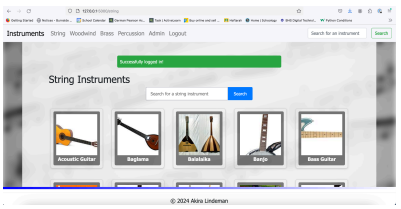
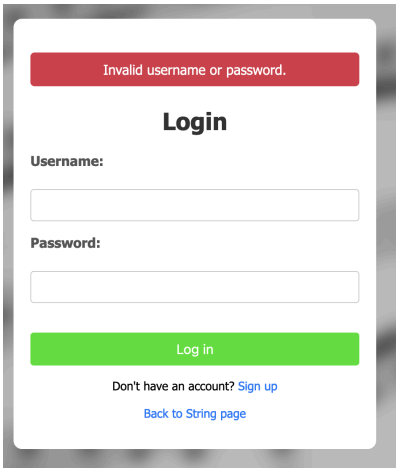
Test password parameters: a password above 50 characters	Type in “TestUsernam e1” fro username and “Test1234567 89101112131 41516171819 20212223242 52627282930 31323334353 63738394041 42434445464 ” for password	Stays on signup page and throws a flash message saying that the password must be fewer than 50 characters long	 This was before a physical boundary was added, so now that it is is it no longer possible to have a username or password greater than 50 characters. Data integrity: The user is not present in the database	Pass
“” Modified	“”	“”	I made a physical boundary of 50 characters so that the user cannot input more than 50. So the account was created succesfully with the shortened length of the password  Data integrity: New user was added to the database 	Pass

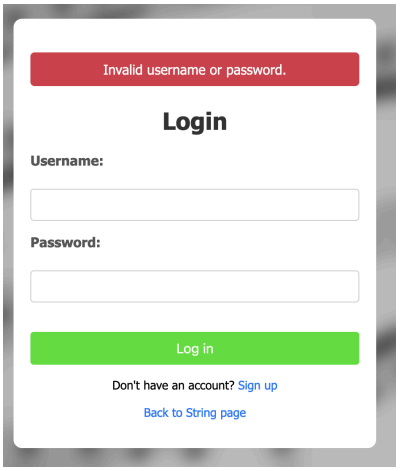
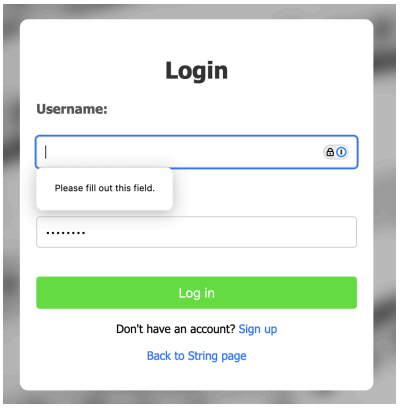
Test password parameters: typing in special characters/Unicode.	Type in “TestUsername2” for username and “⁅↙(K)2Ϻ⁖┐↟╹□ ⇹ഋ⁘” for password	Shouldn’t change the behaviour of the code at all	 Data integrity: User is added to the database	Pass
Test password with no digits	Type in “TestUsername” for username and “TestTest” for password	Stays on signup page and throws a flash message saying that the password must include at least one number	 Data integrity: The user is not present in the database	Pass

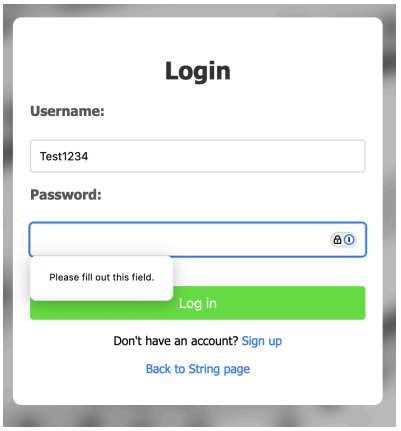
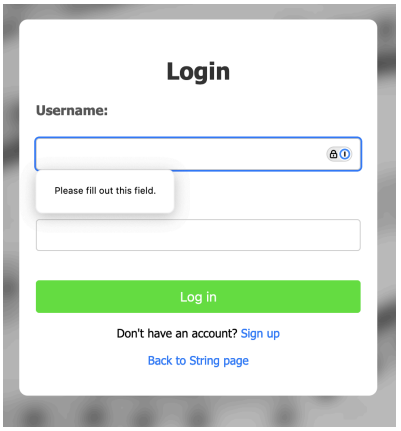
Test password with no uppercase letters	Type in “TestUsername3” for username and “test1234” for password	Stays on signup page and throws a flash message saying that the password must include at least one uppercase letter	 Data integrity: The user is not present in the database	Pass
Test SQL Injection attempts	Type in “TestUsername4” for username and input: '; DROP TABLE Users; – into the password space	Should treat it as any other password so will throw an error for not containing a digit		Pass
Add something to the end of the url	Type in “/test” at the end of the url	Should redirect to 404 page as it doesn't exist		Pass

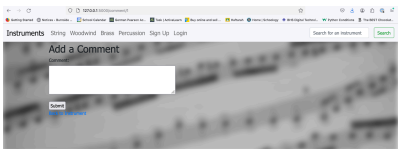
Login page

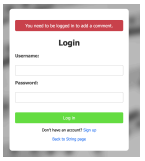
What am I testing	How am I testing it	Expected result	Actual result	Did it work? If not,

				what went wrong and how will you fix it
Does the Signup button work	Hovering over the button to see the route	The url: http://127.0.0.1:5000/signup should show		Pass
Does the ‘Back to string page’ button work	Hovering over the button to see the route	The url: http://127.0.0.1:5000/string should show		Pass
Valid Login	Test with a valid username and password to ensure the user can log in successfully.	Should redirect to the string page with success message		Pass
Invalid Username:	Enter a non-existent username to check if the system prevents login and displays the correct error message.	Should display an error message saying that the credentials are invalid. It doesn’t give the specifics (such as username doesn’t exist) because if someone		Pass

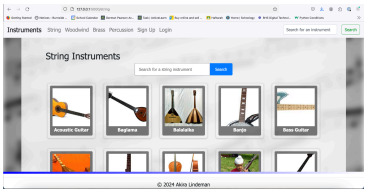
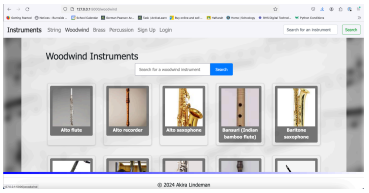
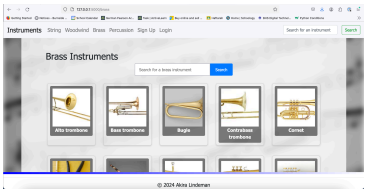
		with malicious intent is trying to hack in or log in then they would gain more information by knowing if one or the other was correct.		
Invalid Password	Type in a username “korea” with the password “Seoul1234” (“Seoul123”) is the correct password	It should flash the invalid credentials message		Pass
Empty username	Leave the Username field empty and type in “Test1234” for password and submit	It should just tell you to fill in the username as this is a mandatory field to fill in		Pass

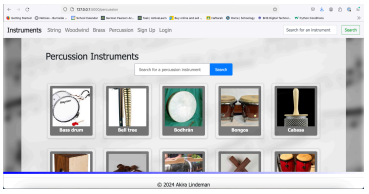
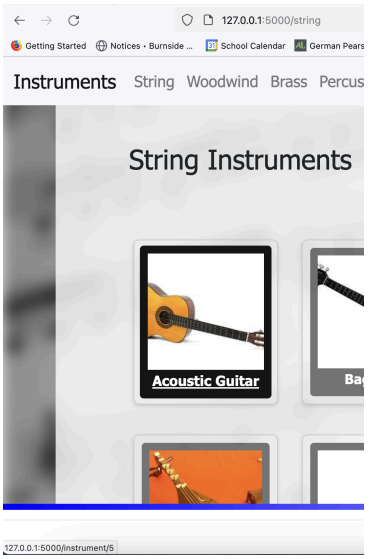
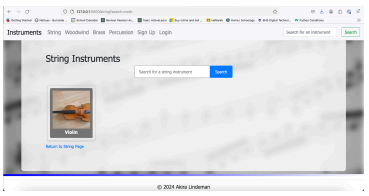
Empty password	Leave the password field empty and type in “Test1234” for username and submit	It should just tell you to fill in the password as this is a mandatory field to fill in		Pass
Both Empty	Leave both empty and click login	Should tell you to fill in the username as this is a mandatory field to fill in		Pass

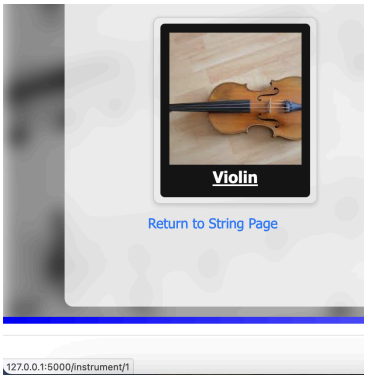
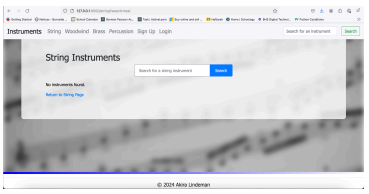
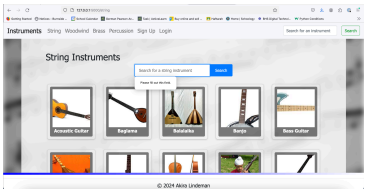
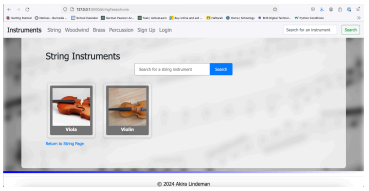
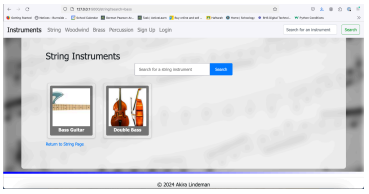
Trying to access comments when you arent logged in.	Type in http://127.0.0.1:5000/comment/1 in the url while not logged in	It should redirect you back to the login page with a message saying that you have to be logged in to add a comment		Fail
				This didn’t work so to fix it I will change the order of my code
				Before:
				 
				After:
				 
				This:
				`# Check if the user is logged in`
				`user_id = session.get('user_id')`
				`if not`

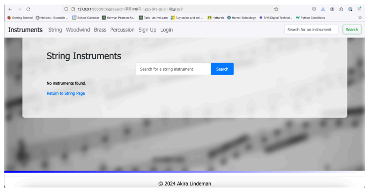
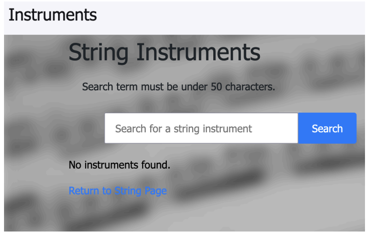
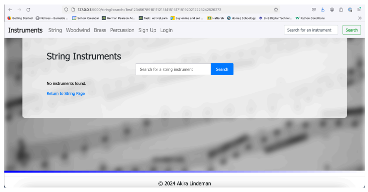
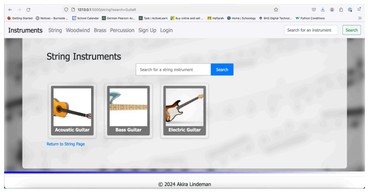
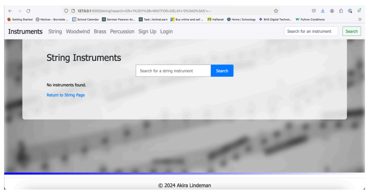
				user_id: flash("You need to be logged in to add a comment.", "error") return redirect(url_for('login')) Was added before the if request.method == 'POST': comment_text = request.form.get('comment') This ensures that you immediately get redirected to the login page with an error telling you that you must be logged in to add a comment.  Pass

Instrument Family pages

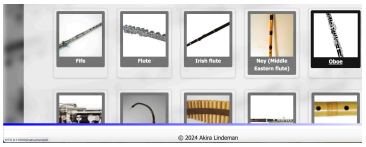
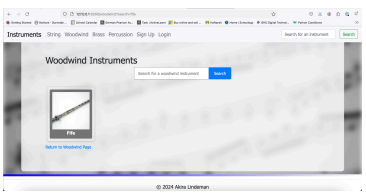
What am I testing	How am I testing it	Expected result	Actual result	Did it work? (if not, what went wrong)
Is the string page displaying the correct information	Going to the url: http://127.0.0.1:5000/string	Nav bar at the top, “String instruments title”, family search bar, individual instrument buttons, footer.		Pass
Is the woodwind page displaying the correct information	Going to the url: http://127.0.0.1:5000/woodwind	Nav bar at the top, “Woodwind instruments” title, family search bar, individual instrument buttons, footer.		Pass
Is the Brass page displaying the correct information	Going to the url: http://127.0.0.1:5000/brass	Nav bar at the top, “brass instruments” title, family search bar, individual instrument buttons, footer.		Pass

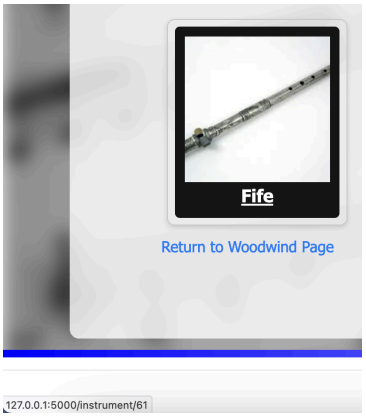
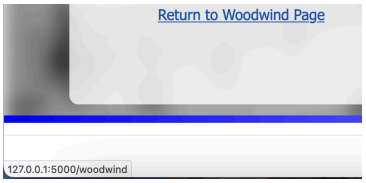
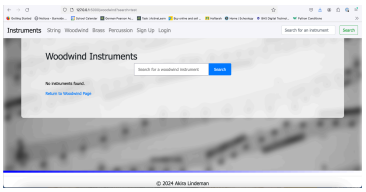
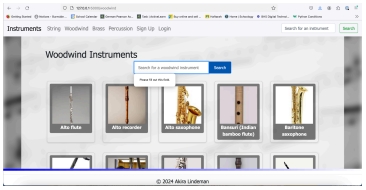
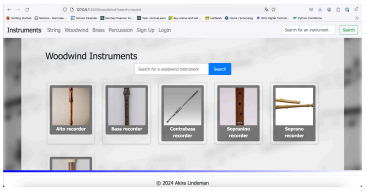
Is the Percussion page displaying the correct information	Going to the url: http://127.0.0.1:5000/percussion	Nav bar at the top, “brass instruments” title, family search bar, individual instrument buttons, footer.		Pass
String Page				
Are the individual instrument buttons working (I am only testing one because because all instrument pages follow the same template and logic, so fixing or confirming one ensures consistency across all others.	Hover the violin button	The button leads to the instrument page. In this case, the violin has the index 1 in my database.		Pass
String search bar: Valid instrument name	Search for “violin”	Should just display the button for violin		Pass

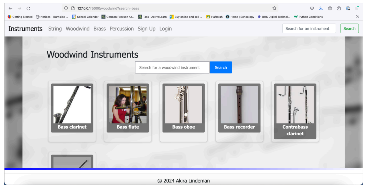
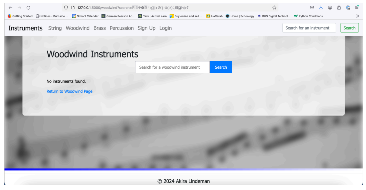
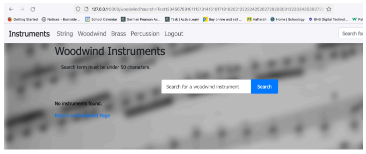
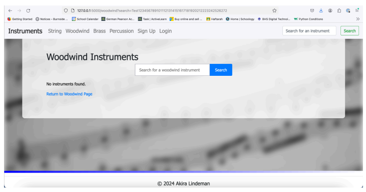
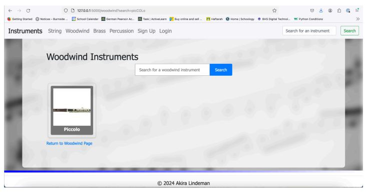
Instrument button after searching works	Hover the “Violin” button	Should show the route: http://127.0.0.1:5000/instrument/1		Pass
“Return to String page” after searching works	Hover the button	Should show the route: http://127.0.0.1:5000/string		Pass
String search bar: Search for an instrument that doesn’t exist	Search for “test”	Should just display the message for the instrument doesn't exist		Pass
String search bar: Search for nothing	Click the button without inputting anything	Nothing should happen so it should stay on the string page		Pass
String search bar: Search for a partial part of an instrument	Search for “vio”	All instrument with “vio” in their name should display		Pass
String search bar: Seach for mixed case	Search fro “bass”	Should display all instrument with “bass” in their name	 (Same logic as previous	Pass

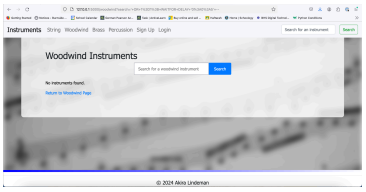
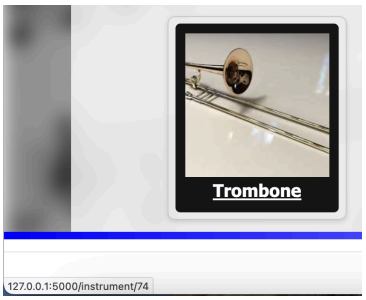
			row)	
String search bar: Seach unicode	Search “□□ᛘ❹□ ⁚)≸⫼⋟②⁝◦ 㐀⒃⁖毋◪┣⑪ 夕”	Should just display nothing as it isn’t an instrument		Pass
String search bar: search above 50 characters	Search, “Test1234567891011121314151617181920212223242526272829303132333435363738394041424344445464”	Shouldn’t break and should display a message saying that the search term is too long		Pass
“” Modified	“”	“”	I made a physical boundary of 50 characters so that the user cannot input more than 50. So it searched for, “Test12345678910111213141516171819202122232425262 72” 	Pass
String search bar: Different letter casing	Search for “GuItaR”	Should display all results for guitars		Pass
String search bar: SQL injection tesing	Input ' OR 1=1; WAITFOR DELAY '0:0:5' -- In theory this should	It will treat it as any other input so would throw the message saying that this		Pass

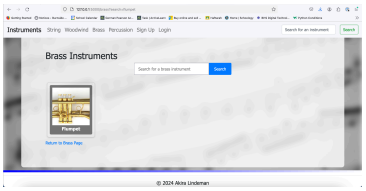
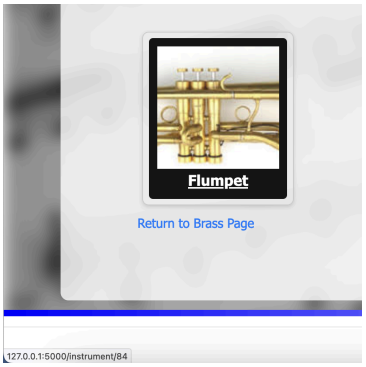
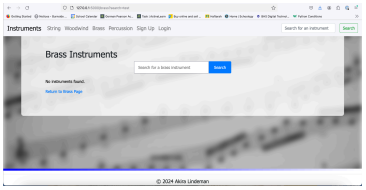
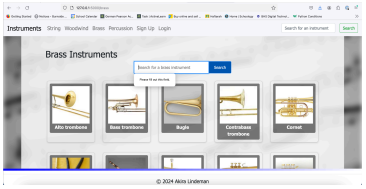
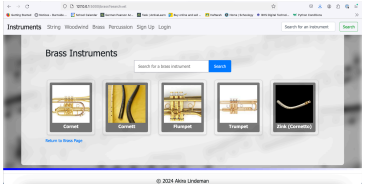
	create a time delay	instrument doesn't exist		
Woodwind Page				

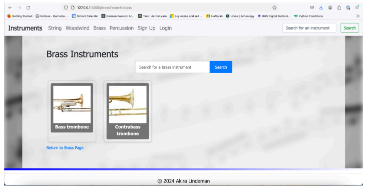
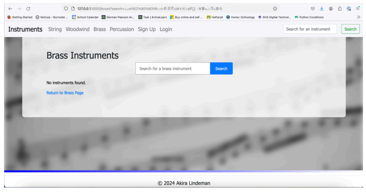
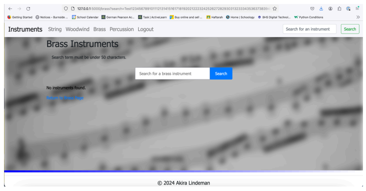
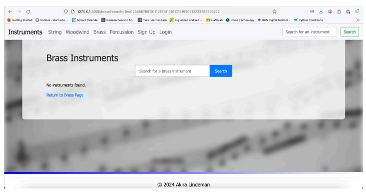
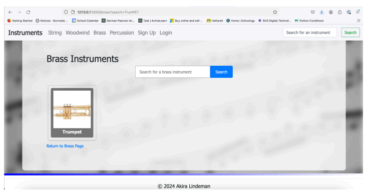
What am I testing	How am I testing it	Expected result	Actual result	Did it work? (if not, what went wrong)
Are the individual instrument buttons working (I am only testing one because because all instrument pages follow the same template and logic, so fixing or confirming one ensures consistency across all others.	Hover the oboe button	http://127.0.0.1:5000/instrument/40 should display		Pass
Woodwind search bar: Valid instrument name	Search for "fife"	Should just display the button for fife		Pass

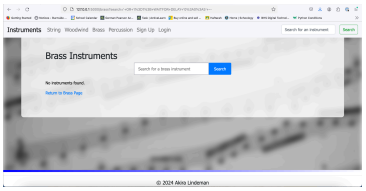
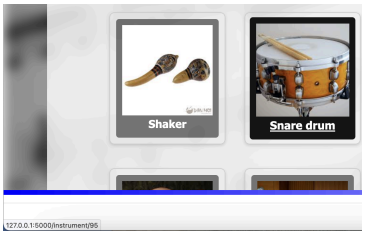
Instrument button after searching works	Hover the “Fife” button	Should show the route: http://127.0.0.1:5000/instrument/1		Pass
“Return to Woodwind page” after searching works	Hover the button	Should show the route: http://127.0.0.1:5000/woodwind		Pass
Woodwind search bar: Search for an instrument that doesn’t exist	Search for “test”	Should just display the message for the instrument doesn't exist		Pass
Woodwind search bar: Search for nothing	Click the button without inputting anything	Nothing should happen so it should stay on the woodwind page		Pass
Woodwind search bar: Search for a partial part of an instrument	Search for “record”	All instrument with “record” in their name should display		Pass

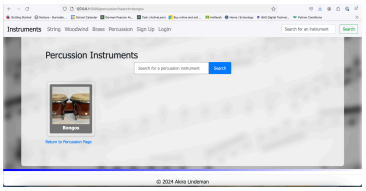
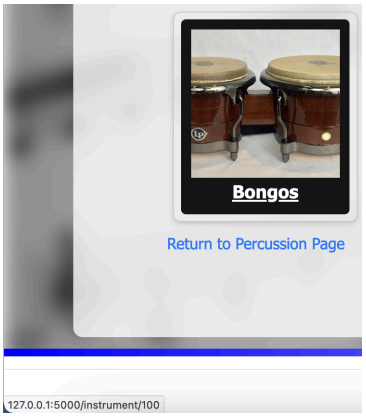
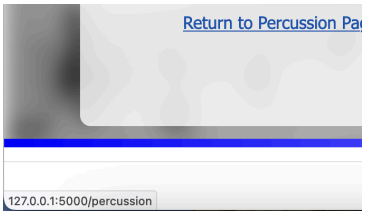
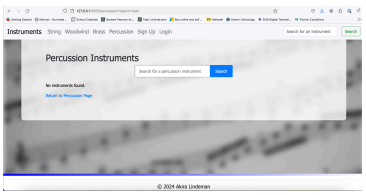
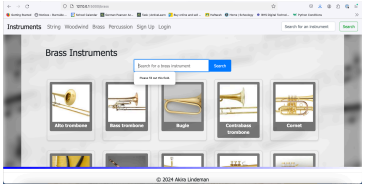
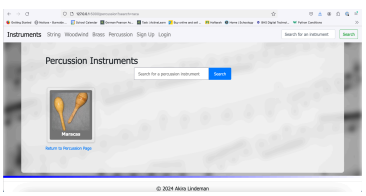
Woodwind search bar: Seach for mixed case	Search fro “bass”	Should display all instrument with “bass” in their name		Pass
Woodwind search bar: Seach unicode	Search “□□ᛉ❹□⠉⟩≴⦀⋟②⡇∘⫫⒃⠪毋◪┣⑪夕”	Should just display nothing as it isn’t an instrument		Pass
Woodwind search bar: search above 50 characters	Search, “Test12345678910111213141516171819202122232425262728293031323333435363738394041424344454 64”	Shouldn’t break and should display a message saying that the search term is too long		Pass
“” Modified	“”	“”	I made a physical boundary of 50 characters so that the user cannot input more than 50. So it searched for, “Test123456789101112131415161718192021222324 2526272” 	Pass
Woodwind search bar: Different letter casing	Search for “picCOLo”	Should display results for the piccolo		Pass

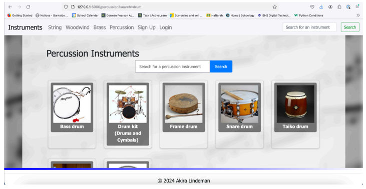
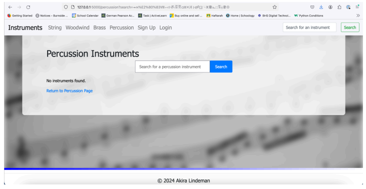
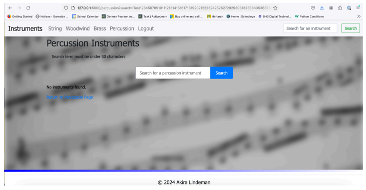
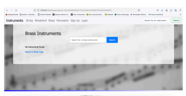
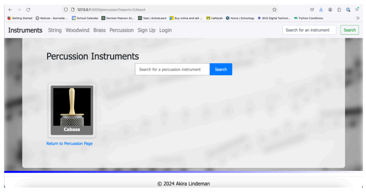
Woodwind search bar: SQL injection tesing	Input ' OR 1=1; WAITFOR DELAY '0:0:5' -- In theory this should create a time delay	It will treat it as any other input so would throw the message saying that this instrument doesn't exist		Pass
Brass Page				
What am I testing	How am I testing it	Expected result	Actual result	Did it work? (if not, what went wrong)
Are the individual instrument buttons working (I am only testing one because because all instrument pages follow the same template and logic, so fixing or confirming one ensures consistency across all others.	Hover the trombone button	http://127.0.0.1:5000/instrument/74 should display		Pass

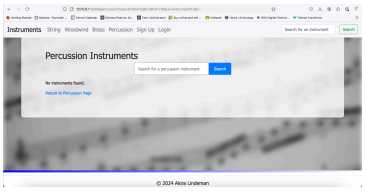
Brass search bar: Valid instrument name	Search for “fulmpet”	Should just display the button for flumpet		Pass
Instrument button after searching works	Hover the “Violin” button	Should show the route: http://127.0.0.1:5000/instrument/1		Pass
“Return to Brass page” after searching works	Hover the button	Should show the route: http://127.0.0.1:5000/brass		Pass
Brass search bar: Search for an instrument that doesn’t exist	Search for “test”	Should just display the message for the instrument doesn't exist		Pass
Brass search bar: Search for nothing	Click the button without inputting anything	Nothing should happen so it should stay on the Brass page		Pass
Brass search bar: Search for a partial part of an instrument	Search for “et”	All instruments with “et” in their name should display		Pass

Brass search bar: Seach for mixed case	Search fro “bass”	Should display all instrument with “bass” in their name		Pass
Brass search bar: Seach unicode	Search “┼⇴ɴ VIII ⥪☆ሸ◻◻ ᗡ჻✗丮 ⦚⩽⊩▯⊡╘▯韋 ∾::▯╋⾀◎”	Should just display nothing as it isn’t an instrument		Pass
Brass search bar: search above 50 characters	Search, “Test12345678910111213141516171819202122232425262728293031323334353637383940414243444546 4”	Shouldn’t break and should display a message saying that the search term is too long		Pass
“” Modified	“”	“”	I made a physical boundary of 50 characters so that the user cannot input more than 50. So it searched for, “Test123456789101112131415161718192021222324 2526272” 	Pass
Brass search bar: Different letter casing	Search for “TrumPET”	Should display results for the trumpet		Pass

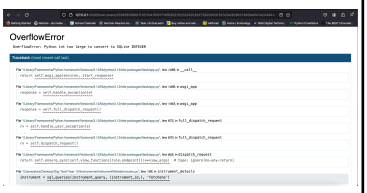
Brass search bar: SQL injection tesing	Input ' OR 1=1; WAITFOR DELAY '0:0:5' -- In theory this should create a time delay	It will treat it as any other input so would throw the message saying that this instrument doesn't exist and will causes zero delay in doing so		Pass
Percussion Page				
What am I testing	How am I testing it	Expected result	Actual result	Did it work? (if not, what went wrong)
Are the individual instrument buttons working (I am only testing one because because all instrument pages follow the same template and logic, so fixing or confirming one ensures consistency across all others.	Hover the snare drum button	http://127.0.0.1:5000/instrument/95 should display		Pass

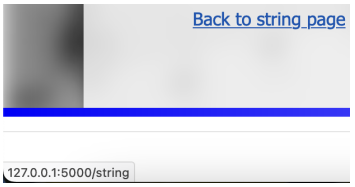
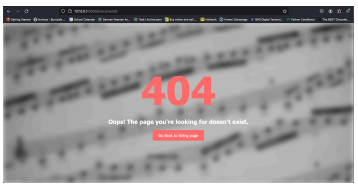
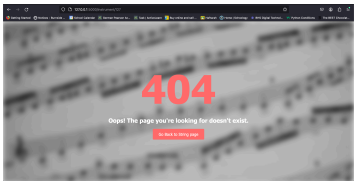
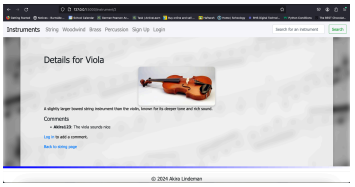
Percussion search bar: Valid instrument name	Search for “bongos”	Should just display the button for bongos		Pass
Instrument button after searching works	Hover the “bongos” button	Should show the route: http://127.0.0.1:5000/instrument/1		Pass
“Return to Percussion page” after searching works	Hover the button	Should show the route: http://127.0.0.1:5000/percussion		Pass
Percussion search bar: Search for an instrument that doesn’t exist	Search for “test”	Should just display the message for the instrument doesn't exist		Pass
Percussion search bar: Search for nothing	Click the button without inputting anything	Nothing should happen so it should stay on the Percussion page		Pass
Percussion search bar: Search for a partial part of an	Search for “raca” (maracas)	All instruments with “raca” in their name should		Pass

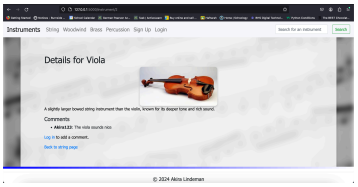
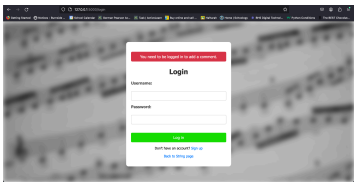
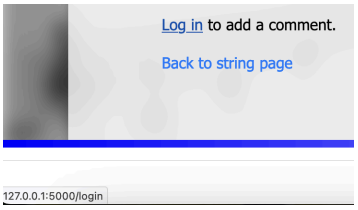
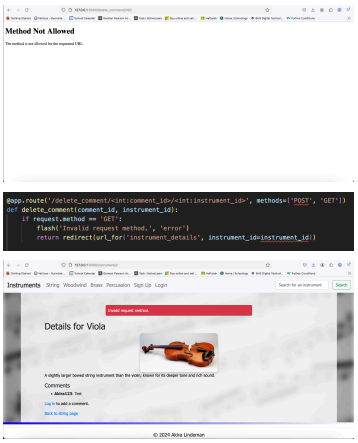
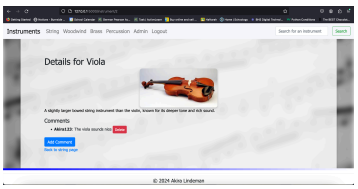
instrument		display		
Percussion search bar: Seach for mixed case	Search fro “drum”	Should display all instruments with “drum” in their name		Pass
Percussion search bar: Seach unicode	Search “⟴ ℵ Ⅷ ⟞ ☆ ᚈ ◻◻ᗡ ⛓✗爿 ⧘⊩▯⧠┗▯韋 ∾∷▯╋聿◎”	Should just display nothing as it isn’t an instrument		Pass
Percussion search bar: search above 50 characters	Search, “Test1234567891011121314151617181920212223242526272829303132333435363738394041424344454647”	Shouldn’t break and should display a message saying that the search term is too long		Pass
“” Modified	“”	“”	I made a physical boundary of 50 characters so that the user cannot input more than 50. So it searched for, “Test1234567891011121314151617181920212223242526272” 	Pass
Percussion search bar: Different letter casing	Search for “CAbasA”	Should display results for the trumpet		Pass

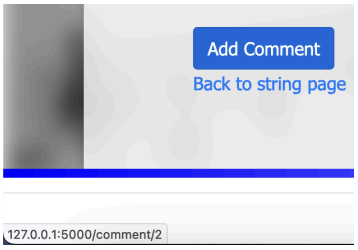
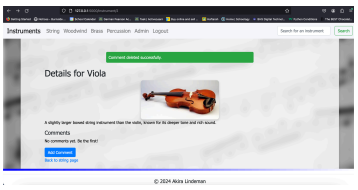
Percussion search bar: SQL injection tesing	Input “Dhol; DROP TABLE Instrument; –” In theory this should create a time delay	It will treat it as any other input so would throw the message saying that this instrument doesn’t exist and will handle the input securely without executing any harmful SQL commands.		Pass

Individual Insturment page

What am I testing	How am I testing it	Expected result	Actual result	Did it work? (if not, what went wrong)
Input a very long string of numbers in the url to see if it handles it without breaking	Input the numbers from 1 to 1000 into the url after the slash	It should redirect to ther error 404 page as the page doesnt exist		It did not work. I will add  These will handle the errors depending on what error is raised.  Now it works

‘Back to string page’ works	Hover over to check the route	http://127.0.0.1:5000/string Should display		Pass
Test boundary cases	Test 0 at the end of the url as the instrument id	It should redirect to the error 404 page		Pass
Test boundary cases	Test 127 at the end of the url as the instrument id	It should redirect to the error 404 page		Pass
When you are not logged in:				
All information is displayed correctly	Go to the viola page and check if the name, image, description, and comments are displaying properly	They will all be there		Pass
Data integrity: is it in the database?	Check the description for the viola in the database and on the webpage	They should be the same	A slightly larger bowed string instrument than the violin, known for its deeper tone and rich sound.	Pass
Data integrity: does this information match public information?	Check the description for the viola on the UK Philharmonia website https://philharmonia.co.uk/resources/instruments/viola/	They should be similar	A slightly larger bowed string instrument than the violin, known for its deeper tone and rich sound. “The viola’s tone is thicker and darker than the violin…The viola looks like a large violin and in terms of its construction”	Pass

You cannot add a comment if you are not logged in	The ‘Add comment’ button is not displayed and instead, the option to Log in is present	Looking on the viola page		Pass
You cannot add a comment if you are not logged in	Manually type in /comment/2 in the url to see if it is still accessible	It should not be accessible and redirect to the login page with an error message		Pass
Login button works	Hover over to check the route	http://127.0.0.1:5000/login Should display		Pass
Delete a comment when you aren’t logged in	If somehow the user knows the comment id and instrument id and they type into the route /delete_comment/24/2 which is the route to delete the ‘Test’ comment on the viola page	It should flash a message that the user must be logged in.		This failed because this function cannot be performed directly via the URL. So I changed the code so that if a user tries to use a GET request, they will be met with a flash error
When you are logged in (Account: Akira123):				
All information displays correctly	Go to the viola page	All information should display with the Delete button for the comment and the Add		Pass

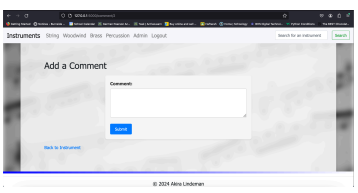
		Comment button		
Add comment button works	Hover over the add comment button	http://127.0.0.1:5000/comment/2 Should display		Pass
The delete button works	Click the delete button next to my comment	Comment should disappear with a message		Pass

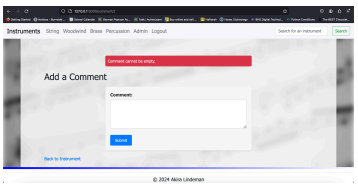
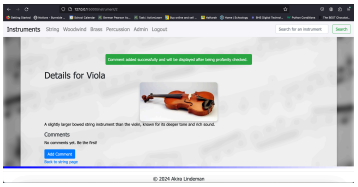
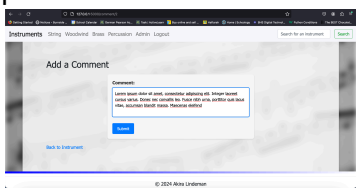
Why I didn't do every page

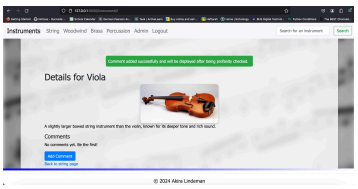
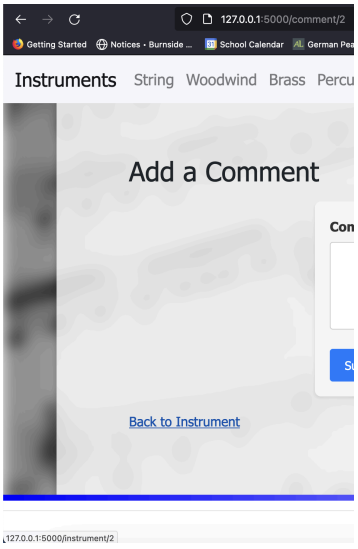
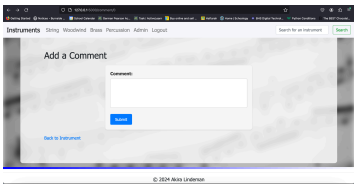
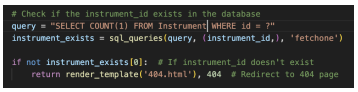
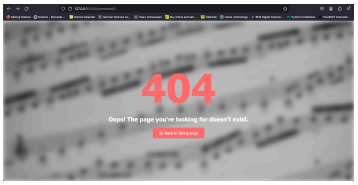
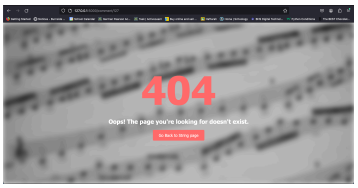
I don't really need to test every single instrument page on my site because, honestly, they're all pretty much the same underneath. Each instrument page pulls from the same template – it just swaps out the name, image, and a few other details. So if I test one, and everything works, it's safe to say the rest should work too. The layout and design don't change between instruments, so as long as the template behaves the way I expect, there's no point in wasting time testing each one individually.

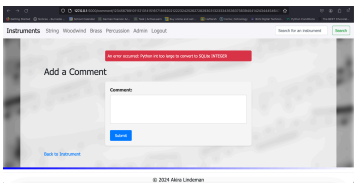
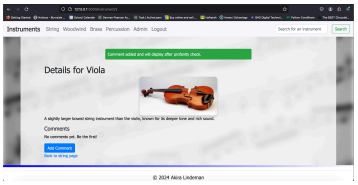
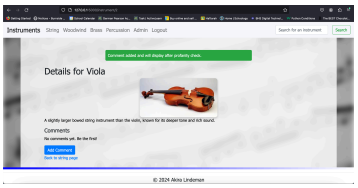
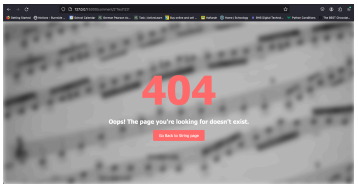
Testing every instrument page individually doesn't provide any additional value, because the template just takes whatever is in the database and displays it. So as long as the data integrity is solid (which I have checked and given one instrument as an example) and the template works, I can be confident that all the instrument pages will load correctly. The focus is on verifying the template logic and ensuring the database is storing the instrument data properly.

Add comment page

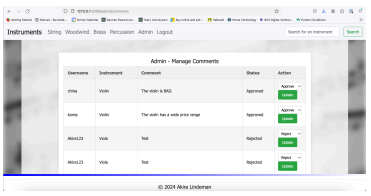
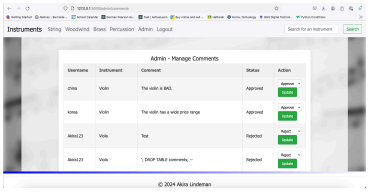
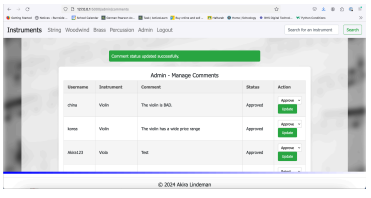
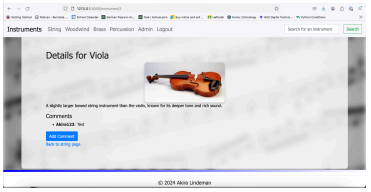
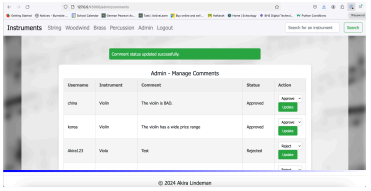
What am I testing	How am I testing it	Expected result	Actual result	Did it work? (if not, what went wrong)
The correct information is displaying	Going to the comment page for the viola	Correct information should display		Pass

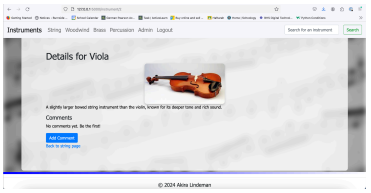
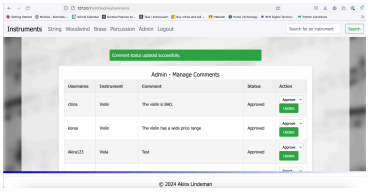
Inputting an empty comment	Click Submit without putting anything in the box	It should do nothing but tell the user that it needs to contain something		Pass
The max length of a comment	Copy and paste 5 paragraphs of lorem ipsum	It should throw an error that the comment is too long		Fail A limit needs to be set for the maximum length of the comment. I will add a physical limit of 200 characters to the I added <textarea id="comment" name="comment" rows="4" cols="50" maxlength="200" style="resize: none;"></textarea> This makes the max length 200 characters and the text area unalterable.
Solution	“”	“”	Only 200 characters are pasted  And the submission works	Pass

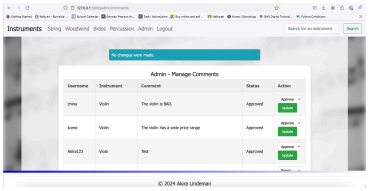
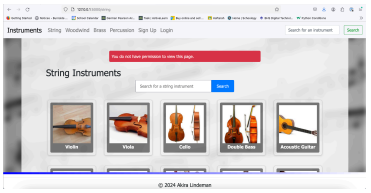
Submit a valid comment	Click submit after pasting 50 characters of Lorem ipsum	Should redirect back to the instrument page with a success message	 Data integrity: is it in the database? Yes	Pass
‘Back to Instrument’ button works	Hover the button	The link for the viola should display http://127.0.0.1:5000/instrument/2		Pass
Test boundary cases	Test 0 at the end of the url as the instrument id	It should redirect to the error 404 page		Fail
Solution	“”	“”	Add the code  This checks if the instrument exists in the database Result: 	Pass
Test boundary cases	Test 127 at the end of the url as the instrument id	“”		Pass

Input a very long string of numbers in the url to see if it handles it without breaking	Input the numbers from 1 to 1000 into the url after the slash	It should redirect to ther error 404 page as the page doesnt exist		It didn't break but it did not work either I will fix this by adding the same try and except statement from the instrument page  Now this works 
SQL injection	Input '; DROP TABLE comments; –	It should just use this as any other comment so it will just redirect to the instrument page with the success message		Pass
Special character testing	Input '💙😂 ĎσЌ𝓡𝓌Vⓦⓤℂ j1v4𝓌ⓜb⁎0 ȷ✻ ù3t4N〒1 ⓠ❾𝓟ⓐ𝔽𝒸⑤¥ ⓢ𝐅ⓗ𝕁o ❸%𝕋④𝒽ⓖℤĎ AbΔℍ'	It should treat it as any other type of string so it should redirect back to the instrument page with the success message		Pass
Edited url	Input 'Test123' into the url after the route	It should redirect to the error 404 page		Pass

Admin Comment page

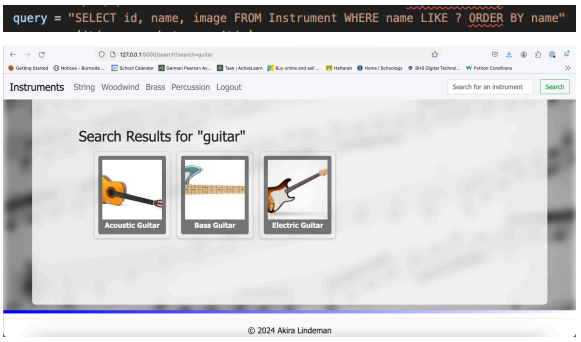
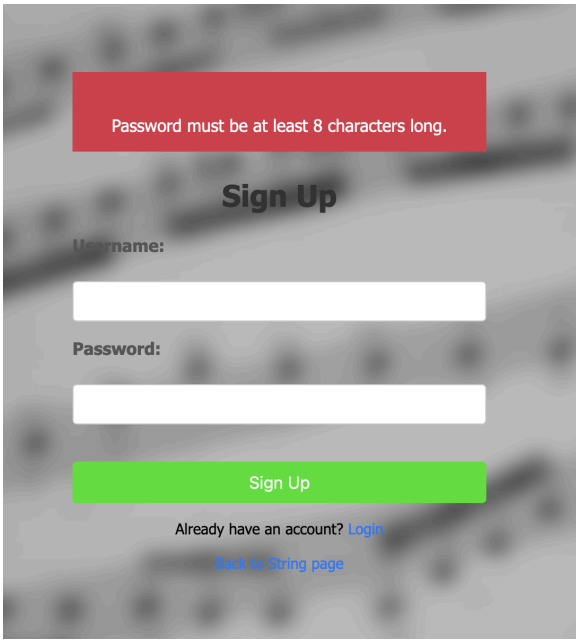
What am I testing	How am I testing it	Expected result	Actual result	Did it work? (if not, what went wrong)
Comments are displaying	Going to http://127.0.0.1:5000/admin/comments	The comments table is displaying with		Pass
See if comments just added are there	Looking for Test comment on viola that was submitted previously	Akira123 Viola Test Rejected Should show The status should be rejected.		Pass
Change the status	Change the status of the 'Test' comment to approve, and then click update	A flash message saying that it was successful. The comment should display on the viola page	 	Pass
Change the status	Change the status of the 'Test' comment to reject, and then click update	A flash message saying that it was successful. The comment should no longer be displayed on		Pass

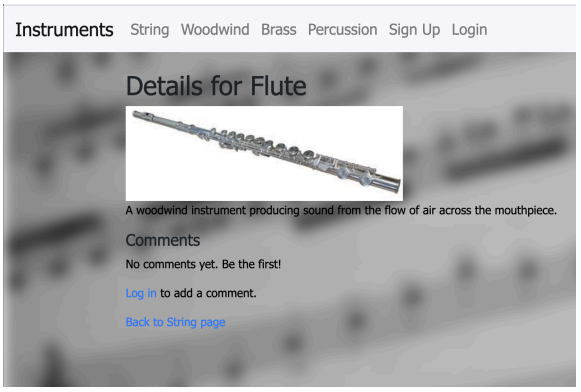
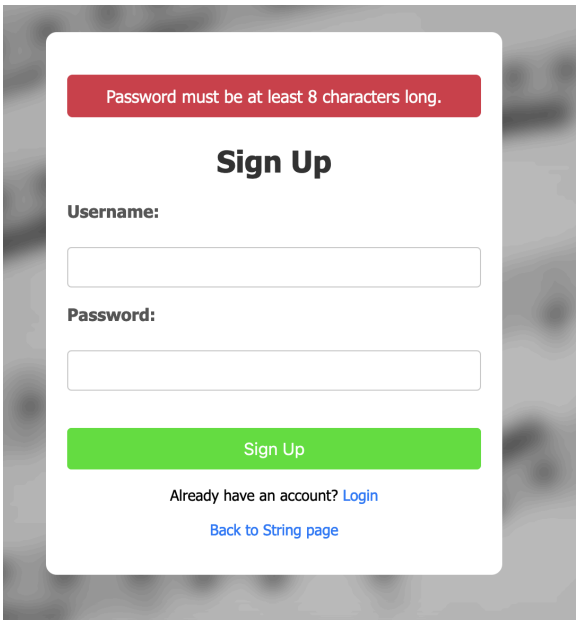
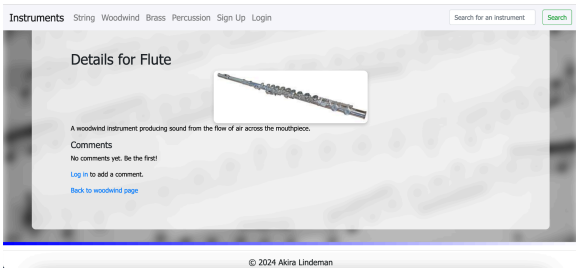
		the viola page		
When the comment is approved there is a 1 in the database	Check the current status of the ‘Test’ comment in the database	There should be a 0 in the status column		Pass
When the comment is rejected there is a 1 in the database	Change the status of the ‘Test’ comment to approve, and then click update Check the current status of the ‘Test’ comment in the database	There should be a 1 in the status column		Pass
Update nothing	Click update without changing anything	Nothing should happen		Nothing is updated but the flash message displays
“”	“”	“”	I will add code to fix this  **Fetch current status**: Before making an update, the code fetches the current status of the comment from the database. **Check for changes**: The if **str(current_status) != new_status** condition checks if the new status is different from the current status. If it's the same, the	Pass

			UPDATE query is skipped, and no update occurs. **Flash message**: If no changes were made, a flash message is displayed with the message "No changes were made.". If a change was made, it flashes "Comment status updated successfully.". 	
Access to the page when you aren't logged in	Logging out and going to this link http://127.0.0.1:5000/admin/comments	Reject and have a flash message		Pass

Tests from 3rd-party members

Name	Feedback/changes
Ron (programmer)	For search: ● Could maybe sort the results after searching alphabetically but not a big deal For admin page: ● Could maybe sort from most recent (the table) - maybe could add timestamp to database Maybe style bootstrap buttons? Not a big deal though because burnside does this too. Otherwise it's really good. The comment profanity check is really cool and all the buttons are very pro. Couldn't find any bugs either so overall well done. An E in my books.
Addressing	I can definitely sort the instruments so that they are in alphabetical order when searched.

	I will do this by implementing **ORDER BY name** in the queries. For example: 
Jessica Fu - Programming peer	The overall functionality and user interface of the interface was cool. The CSS could use some improvement especially with the background colour on different fonts. The comment page could be styled better. I think it looks pretty good, it is a website that I would use if I wanted to read about instruments. The padding on the error messages looks a bit weird. The back button on the individual instrument pages leads back to the strings page and not the corresponding pages. 

Addressing	 I changed the styling of the error messages to be more professional  I added a background layer in front of the background image to make the text and other things more readable. 
Robert Lindeman (father)	The images of the instruments that are used on the landing page are very attractive. I tried to add a comment before loging in. It took me through the login creation page, but then ended on the main page. It would be more convenient if it remembered where I was before the login procedure.

	I really like the way the name of the commenter is in **bold.** It would be helpful if the cursor was automatically set to the text-entry box when I first land on the Comment page, so that I don't have to click on the text-entry box first.
Adressing	To go back to the instrument page that you came from when you logged in, I will have to store the url of the instrument that the user was at previously.  Then in the login function I can have this if the login is valid  This was successful. To make the cursor already selected on the comment text box when on the comment page, I can simply put **autofocus** as the end of the opening tab for the textarea. 
Kaori Lindeman (mother)	(1) On the main page (e.g., string), since there are quite many instruments listed, I might make different groups, such as Europe (West), Asia, Middle Africa etc. (2) As for the "details" page, I would put little more space underneath the picture of the instrument before the description (3) The nav bar (Instrument, String…Search) is done very nicely. (4) The design of each button (each instrument) is nicely done, and it is nice that it changes the color when the cursle is on the top. (Good user interface)
Adressing	(1) This could be a good idea, but I don't think that sorting by country would be the most user friendly as they might not know where an instrument is from. In the future, one thing that I could add could

a a sort button with options on how to sort the instruments which could include, ‘by country’.

(2) Adding space space underneath the picture of the instrument before the description on the “details” page is an easy fix.

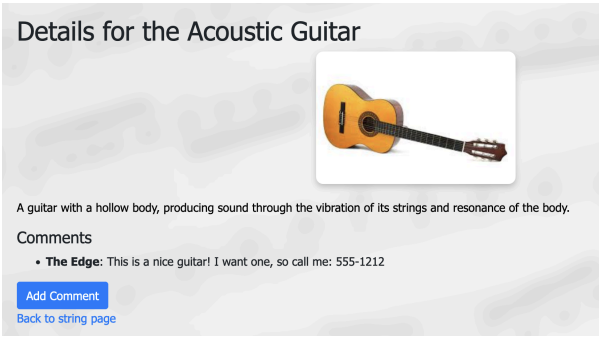
I will apply flex-container

```
.flex-container {
    display: flex;
    flex-direction: column;
    gap: 20px;
}
```

I apply it to the instrument image and description

```
<div class="flex-container">
  <img src="{{ instrument[3] }}" alt="Image of {{ instrument[1] }}" class="instrument-image">
  <p>{{ instrument[2] }}</p>
</div>
```

This fixed the problem



Improvements for the future

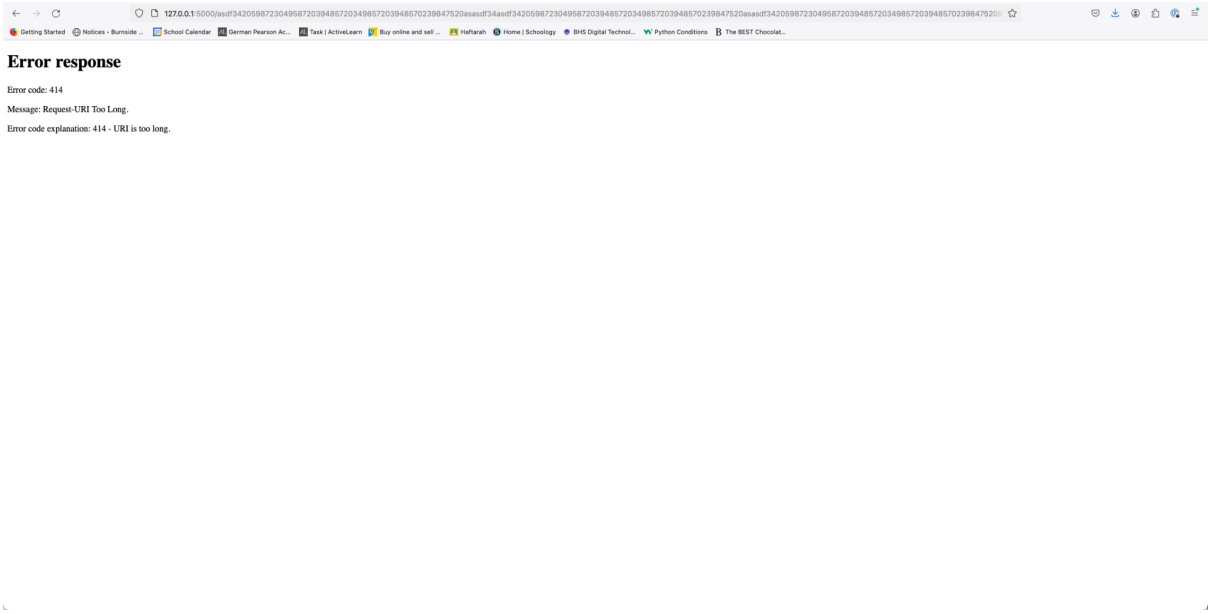
Here I will outline what I could improve and add on in the future if I kept working on my website.

As stated before in the third party testing area, I would add a sorting feature of the instruments that the user could chose. This would make it so that if they wanted to see the instruments of a certain category, they would be able to do so.
For example, world region.

The next this that I would add is a way for a user to request admin, if more people were to have this role. This is because at the moment, I have one admin account that I made and it was made into an admin account by changing the admin status in the database which is obviously not the ideal way to do it.

To fix this I would add a box on the signup page or add a profile page and make them able to request admin there which the ‘super admin’ could approve (and there would only be 1 super admin).

One bug that I am aware of is that if you type a string into the url that exceeds roughly 70,000 characters, the website breaks.



I am not sure of how to fix this, so if I had more time then I would look into it more in depth to try and solve this problem. But for the time being I think that my response which was documented earlier is apt enough as this is still just a highschool project and not a professional website (yet).

Another thing that might help is to add back a home page, which would be a more explicit page for what you can do on the website. At the moment, the user isn’t really guided with what to do but they have all of the options in the nav bar. Users visit website because they have a purpose that they want to fulfil, so if they have somewhat of an idea in mind that they want to achieve, then they will most likely be able to navigate through (evidence of this is the

third-party testing). I do think that if they don’t know what theta re doing or what thye want to do and are just browsing first-time users then they could have a hard time, so a home page could help them and guid them.

I would also change the admin page so that I can sort the comments by various things such as latest and alphabetical rather than the current instrument ID. I would also make it so that each row also functions as a button for the page where the comment is.

Relevant Implications

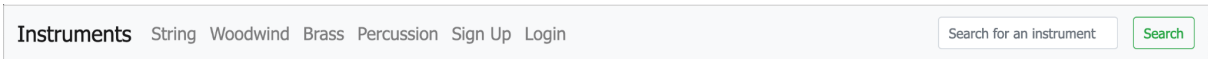
Website

Usability/Accessibility

Usability and accessibility are crucial for ensuring that users can easily interact with a website, which directly impacts their overall experience. A website should be intuitive and straightforward, incorporating established design heuristics that guide users in understanding how to navigate it. This includes clear menus, well-structured layouts, and easily identifiable calls to action. By minimising the effort required to find information or complete tasks, such as filling out forms or accessing different pages, websites can cater to a broader audience, including those with disabilities. Prioritising usability and accessibility allows everyone to engage with the content effectively.

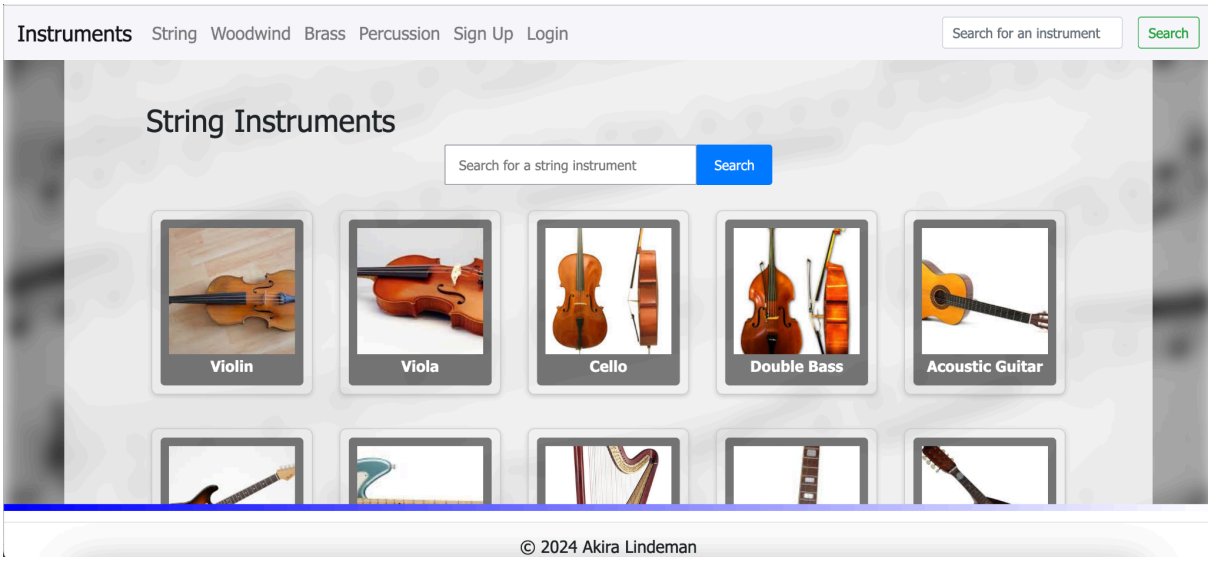
Some ways that I have addressed this relevant implication in my website is through the use of a navigation bar, and search bars.

The navigation bar provides clear pathways to different sections, allowing users to explore content intuitively without feeling lost. Meanwhile, the search bar offers a direct way to find specific information quickly, catering to users who may not want to browse through the different families. This combination helps accommodate various user preferences and behaviors, ensuring that everyone can access the desired information efficiently and without frustration, ultimately improving the overall user experience.



I also added a search bar on every instrument family page to make it even easier for the user to navigate as if they want to know if an instrument is in a certain family, they can search for it in the family search bar instead of looking through through whole page.

This can be seen here:



Aesthetic

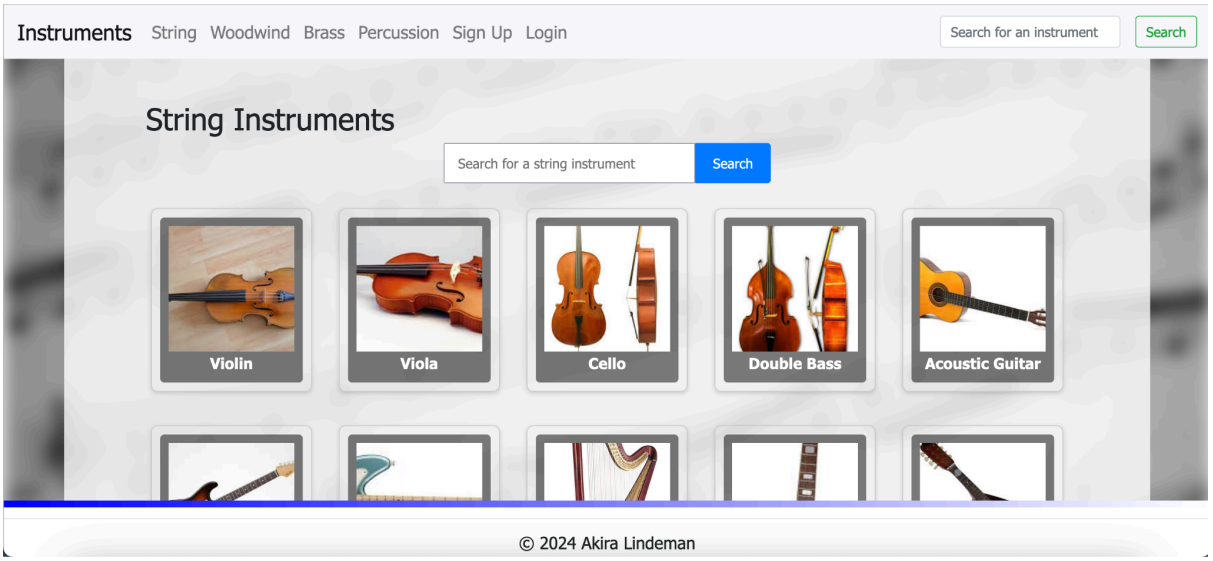
Aesthetics refer to the overall appearance of a website, encompassing its design, layout, colour scheme, and CSS elements. It includes everything that contributes to what users visually experience when they visit the site.

In my website, I used the font stack:

'Segoe UI', Tahoma, Geneva, Verdana, sans-serif

The font stack 'Segoe UI', Tahoma, Geneva, Verdana, sans-serif is a good choice for a professional website for several reasons. First, **Segoe UI** offers a clean and modern look, enhancing readability and user experience. **Tahoma** and **Verdana** are both widely recognized for their legibility on screens, making text easy to read across various devices. Additionally, this combination maintains a consistent and professional aesthetic, ensuring that the website appears polished and trustworthy. Finally, using a sans-serif font contributes to a contemporary feel, which is often preferred in professional settings.

Images on the grid for instruments 'card' layout



Displaying my instruments in the "card" format is a good design choice that enhances both the visual appeal and functionality of your web application. Cards offer a clean and organized layout by dividing content into distinct, containers. This structure makes it easy for users to look through multiple instruments without feeling overwhelmed. Visually, the card format helps the user by showing them images, names, and the name at a glance.

From an aesthetic standpoint, the card layout provides a good balance between text and visuals. Since images are a significant part of what draws user attention, placing each instrument's image within a dedicated card enhances the browsing experience.

Database

Security

Security refers to the measures a database employs to protect user data from hackers and unauthorized access. A well-secured database makes it challenging for malicious actors to compromise user accounts, ensuring that personal information remains safe from breaches and attacks.

In my website, I have implemented these measures to ensure that the website is protected from people with malicious intent and that the data of users which should be kept private is private.

My database is protected against SQL injection because I utilised parameterised queries with the "?" placeholder when inserting and updating data. Using parameterized queries with the "?" placeholder protects against SQL injection by separating SQL code from user input. This means that any data provided by users is treated solely as data, not executable code. When a query is prepared, the database engine knows to expect certain types of input, so even if a malicious user attempts to inject harmful SQL commands, they will be treated as plain text and not executed. This effectively safeguards the database from unauthorized manipulation and data breaches.

Here is one example of me doing so:

```
# Inserting comment into database
query = """
INSERT INTO Comments (instrument_id, user_id, comment, comment_status)
VALUES (?, ?, ?, ?)
"""
```

In this query, the use of the "?" placeholders allows for parameterized input when inserting a comment into the database. When the user submits a comment, the actual values for **instrument_id, user_id, comment,** and **comment_status** are provided separately from the SQL code. This means that the database treats these values as data rather than executable commands.

Privacy

Privacy centers on protecting users' confidential information stored in the database. It ensures that sensitive data, like emails and payment details, is kept private and not exposed to the public or shared without explicit consent. A database should uphold user privacy by only allowing the sharing of information that users have agreed to make public.

To prevent the administrator (me) from seeing the passwords of my users, I hashed the password on entry into the database so the passwords are encoded and unobtainable by me.

This is how I encoded them.
This is used in the signup and login function

```python
def hash_password(password):
    """Hashes a password using SHA-256."""
    return hashlib.sha256(password.encode()).hexdigest()


def check_password(stored_password, provided_password):
    """Checks if the provided password matches the stored password."""
    return stored_password == hash_password(provided_password)
```

This code securely handles passwords by hashing them with the SHA-256 algorithm. The **hash_password** function takes a plain text password, encodes it, and generates a unique hash, which is then stored in the database instead of the actual password. This enhances privacy because even if the database is compromised, the actual passwords remain protected, as attackers would only find hashed values. The **check_password** function compares a hashed version of a user-provided password against the stored hash to verify authenticity, further ensuring that sensitive data is never exposed or transmitted in its original form.

For exampl:

39	TestUsername	1af4607bec2f28278e9f67e8013f6fd52e64fa6dbe96e19b34e02b518d00b5a0	0

Sustainability and future proofing

Sustainability and future-proofing for a database mean making sure it can grow and adapt over time. First, the database should be scalable to handle more data and users as your needs change. Keeping data secure and intact is crucial to protect against future threats. The database should easily work with other systems to keep up with new technologies. Good data management practices help ensure data is handled responsibly. Finally, you need the database to support and updates, making the system more resilient for the future.

My database is sustainable and future proof because

Normalized and Structured for Growth

I've made sure that my database is properly normalised, meaning each table has a clear and distinct purpose without redundant data. For example, the **Users, Comments**, and **Instruments** tables are well-structured with relationships using foreign keys. This helps maintain data integrity and makes future updates or expansions easy to manage. If I decide to add more tables or features, the design will support that without significant changes.

Flexible Comment and Image Handling

The way I've designed the comments system with a **comment_status** column allows for easy scalability. Right now, I use simple approval or rejection statuses, but in the future, I could easily expand this to include more states, like "pending" or "flagged." Additionally, the flexibility in handling different instrument image sizes (both landscape and portrait) ensures that the layout remains adaptable as I add more instruments with varying image formats.

Scalable Search and Modular Routes

I've set up a search feature and modular routes like **/instrument/<int:instrument_id>** that make it easy to scale. In the future, I can add more filtering options to the search or even introduce new actions (like editing or reporting comments) without overhauling my route system. This structure makes my application ready to grow while maintaining a clean and logical flow.

These points make it easy for me to add new information in the future and edit the design of the database (if needed) without much change.