EECS 485 Project 3:   
Client Side Dynamic Pages

Due 9pm Thursday, Oct 27, 2016

Client-side applications impact the frontend (that is, the HTML/CSS/JS), and also the backend. Backends use REST API principles, which create a common interface (the API) used simultaneously by frontend web applications, mobile applications, internet of things (IoT) devices, etc.

In this project, you will revise the web application that was built in P2 to become a hybrid between a [single-page application](https://en.wikipedia.org/wiki/Single-page_application) and the route-based application that currently exists. This means that for some routes of your website, you will not get the entire HTML contents from the server. Instead, you will create client-side JavaScript that will fetch data in the background using AJAX, and then insert the content into the page. You will be adding a caption feature for photos, as well as a REST API in Flask that will be used by your JavaScript to dynamically create pages.

[Part 1: Implementation Preliminaries](#_ubmuq27tu0mn)

[Part 2: Build the website](#_3znysh7)

[Part 3: URL Navigation and History](#_fqw5fwkxzf5u)

[Part 4: Authentication and Session Management](#_17dp8vu)

[Part 5: Error Checking](#_2z9epg6nc4b)

[Part 6: Organization Tips](#_fjk3mvkvjria)

[Part 7: Grading and Deliverables](#_aiaamufi2u1h)

# Part 1: Implementation Preliminaries

Please do not touch the files in the P2 sub-directory. Make another sub-directory called P3, and copy the files from P2 into the P3 sub-directory and work on the files there. In addition, the only JavaScript library you are permitted to use is [jQuery](https://jquery.com/).

# Part 2: Build the website

## Implement API and HTML Pages

You will be separating your backend (Flask-python) from your front-end (HTML, JavaScript). Many routes in your Flask-Python app will now only offer API endpoints that return JSON responses, instead of returning a full template via the render\_template function.

Your base.html template can render the navigation links discussed in Grading and Deliverables. These do not need to be inserted with Javascript on every page. This template can check the session to determine if the logout or login navigation link should be displayed.

What follows is a list of the url endpoints that you should create in your application. You should have created some of these for P1/P2. If a page is not listed here, presume it should behave the same way as in Project 2.

### API Routes

The functionality for most of these routes is the same as your controllers in the previous projects. You should be moving functionality from there into new API routes. New to these routes is [variable routes](http://flask.pocoo.org/docs/0.10/quickstart/#variable-rules). These look like:

@app.route(‘/testroute/<testvariable>’)

def test\_route(testvariable):

The name of the variable in the route must match the name of the parameter in your function.

Note that POST requests to and responses from your API should specify a [MIMETYPE/Content-type](https://en.wikipedia.org/wiki/Media_type) of application/json. Flask’s [jsonify](http://flask.pocoo.org/docs/0.11/api/#module-flask.json) function will automatically do this for your API responses. If you’re using jQuery for your requests, look into changing the contentType setting for the [ajax()](http://api.jquery.com/jquery.ajax/) function.

For testing the API routes, try using [cURL](http://www.codingpedia.org/ama/how-to-test-a-rest-api-from-command-line-with-curl/), [python requests](http://docs.python-requests.org/en/master/) or [Postman](https://www.getpostman.com/): these will let you hit your API routes and get feedback without writing any JavaScript.

#### New User API: `GET /api/v1/user` [sensitive]

This API route is for fetching the current user. It should GET to /api/v1/user. A JSON object of the following format should be returned if the user has a valid session:

{

“username”: “sportslover”,

“firstname”: “Paul”,

“lastname”: “Walker”,

“email”: “sportslover@hotmail.com”

}

#### New User API: `POST /api/v1/user` [public]

This API route is for user creation. It should POST to /api/v1/user with a JSON object of the following:

{

“username”: “sportslover”,

“firstname”: “Paul”,

“lastname”: “Walker”,

“password1”: “paulpass93”,

“password2”: “paulpass93”,

“email”: “sportslover@hotmail.com”

}

If successful, a JSON object with the same fields specified in `GET /api/v1/user` should be returned and an HTTP status code of 201.

#### Update User API: `PUT /api/v1/user` [sensitive]

The user should be able to change their firstname, lastname, password and email address (but not username). Updated information should sent via PUT request to /api/v1/user. Your PUT request should have all of the same fields as required in POST /api/v1/user. If password1 and password2 are the empty strings, then assume they are not being modified.

#### User Login API: `POST /api/v1/login` [public]

The user should be able to POST to /api/v1/login with a JSON object of the following:

{

“username”: “username”,

“password”: “password”

}

This should sign in the user. If successful, it should return a JSON object `{“username”: “username”}` where username is the signed-in user.

#### User Logout API: `POST /api/v1/logout` [sensitive]

Used for logging out. A POST request to /api/v1/logout with no body should logout the user and end the session. If successful, return no response body with a status code of 204.

#### Album API: `GET /api/v1/album/<albumid>` [sensitive/public]

This API route retrieves the information for an album (specified by albumid) and its pictures. It should return a JSON object of the following format:

{

"access": "public",

"albumid": 1,

"created": "2016-01-01 00:00:00",

"lastupdated": "2016-02-02 00:00:00",

"pics": [

{

"albumid": 1,

"caption": "",

"date": "2016-01-01",

"format": "jpg",

"picid": "5c00dd3598ce621105cb7062a59e7931",

"sequencenum": 0

},

],

"title": "I love sports",

"username": "sportslover"

}

Please note that we have omitted the remaining album pictures from the example JSON output for brevity purposes.

#### Pic API: `GET /api/v1/pic/<picid>` [sensitive/public]

This API route retrieves the picture information for picid. It should return a JSON object such as the following:

{

"albumid": 1,

"caption": "Pelle Pelle",

"format": "jpg",

"next": "568ab398af3555d9c991c62b0e4d024c",

"picid": "63b1f8027b1cdc739ac89b2dd62cb108",

"prev": "933b775e7ea1d6575271103b00e7e965"

}

#### Update Picture API: `PUT /api/v1/pic/<picid>` [sensitive]

This API route updates the picture information for picid. It should accept a JSON object with the same format as the JSON response of `GET /api/v1/pic/<picid>`. Only captions are modifiable. On successful update, return a JSON object of the updated resource (same fields as those sent in the request) and a status code of 200. This should also update the Album.lastupdated field for the album that the pic is in.

### Static Page Routes

All you should be doing in these routes is rendering an HTML template, or checking for a valid session and redirecting if applicable. The actual logic for these actions should be in your new API controllers. For example, your /user/edit Flask controller should either render a template with an HTML form, or redirect to login if there is no session. The actual editing of user information will be done by the /api/v1/user controller.

#### New User page: /user [public]

This page is for user creation. It contains a form that should use AJAX to POST its information in JSON format to /api/v1/user with username, firstname, lastname, password1, password2, and email. The HTML form need not be inserted dynamically with Javascript.

You should validate both on the **server side in Python and on the client side in JavaScript.** Follow the validation rules in the section later in this document. Please note that you are **not** allowed to use HTML [input tag validation](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/input).

If a user provides all valid information, create the User by inserting their information into your database. Use the same password encryption and user accounts as in P2.

If a session already exists, redirect the user to /user/edit (in Flask). Otherwise, after adding the user to the database, redirect to /login (in Javascript) and allow the new user to log in with their new credentials (do not log them in automatically after signup).

#### Edit Account page: /user/edit [sensitive]

The user should be able to change his/her firstname, lastname, password and email address (but not username). Updated information should use AJAX to send a PUT request to /api/v1/user. Unlike P2, these fields should not be separate forms, it should all be one form. The HTML form need not be inserted dynamically with Javascript. However, you should load data into your HTML form using Javascript and AJAX. Hint: you can use GET /api/v1/user to fetch the current session’s user.

If the request is successful, stay on the page. You may show a success message if the request is successful, but this is not required.

#### User Login page: /login [public]

Here, a non-authenticated user can enter their username and password to login to a specific user account and become authenticated. Client-side checking for this page is not required.

Upon successful login, redirect using JavaScript. If there is a url query parameter (for example: /login?url=/the/prev/url) then redirect to the URL. Otherwise, redirect to /. You may find [window.location](https://developer.mozilla.org/en-US/docs/Web/API/Window/location)helpful. If a user is already logged in, redirect to /user/edit (via Flask).

#### Logout: All Pages [sensitive]

If a user is currently logged in, they should be able to logout by clicking the logout button. This should destroy the session and redirect to the default home page. This should be handled with an AJAX request to /api/v1/logout.

### Single Page Application Routes

For all of these routes, **all that will be in your Python is a call to return a template**. You will be populating the page dynamically using JavaScript.

#### View Album page: /album?albumid=<albumid> [sensitive/public]

This page displays the thumbnail view of an album’s pictures just like the previous assignments. Just like in P2, you can view this page if the album is public, you own the album, or the album is private and you have been given explicit access. However, this should be checked client-side (see Authentication and Session Management below).

The album title should be at the top, along with the album's owner. The photos should be displayed in sequence order, each with its date, and a caption (including the empty string). Similar to project 1, clicking each photo should take you to /pic?picid=<picid>. Your links to the photos should not have an href attribute because navigation will be handled with the Javascript onclick event handler.

This page must insert the album’s contents via Javascript. You should also implement [window.History](https://developer.mozilla.org/en-US/docs/Web/API/Window/history) so that a user can navigate back to this page (refer to [URL History](#_fqw5fwkxzf5u) section). For example, if a user clicks a picture while in an album, they should be able to hit the back button and return to the album. This should not cause a full page re-render, but instead should be handled dynamically. Album information should be fetched with a `GET` AJAX request to your `/api/v1/album/<albumid>` route.

Here is a sample process;

1. User navigates to /album?albumid=<albumid>
2. Flask returns an HTML document only with the necessary script includes and <div id=”content”></div> with no inside divs.
3. Javascript executes fetching the album information from the API via AJAX and inserting the nodes (you may find [Node.appendChild()](https://developer.mozilla.org/en-US/docs/Web/API/Node/appendChild) helpful)
4. User clicks a picture to /pic?picid=<picid>
5. Javascript removes all inner nodes from the content div and inserts necessary picture information. Data is fetched from API via AJAX.
6. User hits back button and is navigated back to /album?albumid=<albumid> via Javascript. Album data is refetched via AJAX and reinserted into the DOM’s content div.

A link to edit the album should also be included, although it can use the same /album/edit route from P2 (it need not modify data dynamically client-side).

#### View picture page: /pic?picid=<picid> [sensitive/public]

This page displays a picture just like the previous assignment. It should have the caption, full-sized picture and “links” to previous and next picture. The links should **not** cause the user to go to the server for rendering a new HTML document with the picture. They should instead fetch the new picture information by issuing an AJAX request and update the DOM with Javascript. If the picture is the first or last in an album, you should still show the previous and next links with the proper IDs. However, clicking them should not change the URL.

You must be able to edit the caption. This should be updated by sending a JSON PUT request to /api/v1/pic/<picid> with the entire picture contents required specified. The caption should be an input text element with no submit button. Submission should occur when a user hits enter within the textbox. You can only edit a picture’s caption if it is part of an album that you own. In the case where there is no logged-in user or the user does not have permission to edit the caption, a paragraph element should be shown with the caption in its text. This means either the <p> tag is shown with the caption OR the input field for caption is shown, not both.

You can view this page if the picture is part of a public album, you own the album it is a part of, or it is in an album that is private for which you have been given explicit access. If the user does not have access to the album this picture is in, they should not be able to see the picture; an error 403 should be returned.

# Part 3: URL Navigation and History

When single page applications were first being implemented, a major issue was the lack of working browser URL navigation. Since your browser is never actually switching it’s URL, users could not share links and utilize native browser history functionality like going back and forward. To fix this, the [window.History](https://developer.mozilla.org/en-US/docs/Web/API/Window/history) standard has been proposed and implemented in all major browsers. For a brief introduction, you should read [this](https://developer.mozilla.org/en-US/docs/Web/API/History_API) walkthrough.

In our single page application, we require navigation back and forth between viewing an album and viewing an individual picture. To do so, you will need to make sure to use the [pushState()](https://developer.mozilla.org/en-US/docs/Web/API/History/pushState) function when updating your DOM and register an [onpopstate](https://developer.mozilla.org/en-US/docs/Web/API/WindowEventHandlers/onpopstate) handler for when the browser’s history state changes. To help you get started, here’s a small example of how to use these:

<div id="content"></div>

<script type=”text/javascript”>

window.onpopstate = function(event) {

document.getElementById("content").innerHTML =

event.state.stateVariable;

}

var state1Obj = {stateVariable: "I am currently in state 1!"}

var state2Obj = {stateVariable: "Now I am in state 2!"}

var state3Obj = {stateVariable: "Now I am currently in state 3!"}

history.replaceState(state3Obj, "title", "?state=3")

history.pushState(state2Obj, "title", "?state=2")

history.pushState(state1Obj, "title", "?state=1")

document.getElementById("content").innerHTML =

state1Obj.stateVariable

</script>

Try putting this in an HTML file and viewing it with your browser. Press the back/ forward buttons and look at how the URL and page change.

# Part 4: Authentication and Session Management

Just as before, we will be implementing sessions. For this project, you should perform the following checks for **only** pic and album (the Single Page Application routes). All API routes should also do session checking and return the appropriate error message specified in Error Checking. For all other routes, you should use the same error checking as P2.

1. If a session exists, render the page via Flask and display the name of the user currently signed in. Perform necessary client-side AJAX requests to render the content into the page. If your API request returns a 403 status code, render the response message into the page with class error. Otherwise, insert the response content.
2. If a session does not exist, render the page and display a link to the login page (with the url query parameter set to the current page url). This need not be determined via the API request. This can be rendered with the initial page load via Flask’s render\_template. See “Implement API and HTML Pages” above on navigation.
   1. If the page is public, perform necessary AJAX requests to render the content into the page.
   2. If the page is not public, render the unauthorized response message into the page with class error.

# Part 5: Error Checking

You are responsible for checking the errors listed below. These errors will be returned from your REST API when submitting or fetching data (with the exception of many checks for user creation and editing). To get a better understanding of the HTTP status codes chosen, read the descriptions on them [here](https://en.wikipedia.org/wiki/List_of_HTTP_status_codes). Other errors you think of that are not explicitly listed here or elsewhere in this spec you should handle responsibly with a proper status code and message of your choosing.

You must now validate data for your API requests. If an error occurs, you should return a JSON object of the following format:

{

"errors": [

{

"message": "Access forbidden."

}

]

}

The “errors” value is an array of all errors that occur for the page. **Errors denoted with an asterisk (\*) signal that if you encounter the error, you should return immediately with that error message. Do not continue processing the other validation checks.** For example, a user cannot receive a 422 error if they are not first authenticated. Errors are listed in order of decreasing importance. The only scenario where the errors array can have multiple entries is for HTTP status 422 errors (see below). All other error scenarios should have a single entry in the errors array.

All rules are specified in the format <description of error>: <error message to return to users>, <HTTP status code>

Your API and site must enforce the following rules:

#### Sensitive pages with no session

A user is not logged in and the resource is sensitive: “You do not have the necessary credentials for the resource”, 401\*

#### Sensitive pages with unauthorized access

A user is logged in but does not have permissions to fetch/modify the resource: “You do not have the necessary permissions for the resource”, 403\*

#### GET /api/v1/user

A valid session is required: “You do not have the necessary credentials for the resource”, 401\*

#### POST /api/v1/user and /user

* All fields are required: “You did not provide the necessary fields”, 422\*
  + Does **not** need to be checked client-side.
  + This only occurs if a JSON key is missing from the request: Empty strings are valid for firstname and lastname
* The username must be unique (though case insensitive): “This username is taken”, 422
  + Does **not** need to be checked client-side.
* The username must be at least three characters long: “Usernames must be at least 3 characters long”, 422
* The username can only have letters, digits and underscores: “Usernames may only contain letters, digits, and underscores”, 422
* The password should be at least 8 characters long: "Passwords must be at least 8 characters long", 422
* The password must contain at least one digit and at least one letter: "Passwords must contain at least one letter and one number", 422
* The password can only have letters, digits and underscores: "Passwords may only contain letters, digits, and underscores", 422
* The first and second password inputs must match: “Passwords do not match”, 422
* Email address should be syntactically valid: “Email address must be valid”, 422
  + Here is a regular expression which should check email validity well.
  + import re
  + if not re.match(r”[^@]+@[^@]+\.[^@]+”, email):
    - # handle an invalid email address
* The (username, firstname, lastname) fields have a max length of 20: "<field> must be no longer than 20 characters" (If lastname was too long, for example, the error would be “Lastname must be no longer than 20 characters”, 422.
  + email is allowed a max length of 40. The error messages should reflect this.

#### PUT /api/v1/user and /user/edit

* A valid session is required: “You do not have the necessary credentials for the resource”, 401\*
  + You may wish to use the /api/v1/user to check this
* A user is logged in but does not have permissions to fetch/modify the resource: “You do not have the necessary permissions for the resource”, 403\*
  + This occurs if the username sent in the request is not the same as the one in the current session
* Same validation rules as POST /api/v1/user

#### POST /api/v1/login and /login

* All fields are required: “You did not provide the necessary fields”, 422\*
  + This only occurs if a JSON key is missing from the request. If the key is there, but the value is the empty string, you should continue.
* Username does not exist: “Username does not exist”, 404\*
* Password is incorrect for the specified username: “Password is incorrect for the specified username”, 422

#### POST /api/v1/logout and /logout

A valid session is required: “You do not have the necessary credentials for the resource”, 401\*

#### GET /api/v1/album/<albumid> and /album?albumid=<albumid>

* The album must exist: “The requested resource could not be found”, 404\*
* Proper authorization handling as specified above for sensitive pages\*

#### GET /api/v1/pic/<picid> and /pic?picid=<picid>

* The pic must exist: “The requested resource could not be found”, 404\*
* Proper authorization handling as specified above for sensitive pages\*

#### PUT /api/v1/pic/<picid> and /pic?picid=<picid>

* All fields are required: “You did not provide the necessary fields”, 422\*
  + This only occurs if a JSON key is missing from the request. If the key is there, but the value is the empty string, you should continue.
* The pic must exist: “The requested resource could not be found”, 404\*
* Proper authorization handling as specified above for sensitive pages\*
* Only caption is modifiable: “You can only update caption”, 403

Appropriate validation errors must be found in your page in the following form:

<p class=”error”>

\*\*Error message goes here\*\*

</p>

Where \*\*Error message goes here\*\* is replaced with the appropriate error message. For example:

<p class=”error”>

Lastname must be no longer than 20 characters

</p>

**You can assume that the user is acting in good faith: your goal is to prevent users from adding bad usernames/passwords, not to guard against motivated attackers who want to sneak a** [**strange entry**](http://en.wikipedia.org/wiki/Code_injection) **into your password database (which means you do not need to check things beyond above rules).**

# Part 6: Organization Tips

Like P2, we will not be giving you any more starter code. It is up to you to decide how you want to organize all of your files. However, here are some tips that might make managing all the new files a bit easier:

* For your JavaScript files, don’t put all of the code into main.js! Modularize them, and make a different file for user, pic, album, etc.
  + You can import an external JavaScript file in your HTML with:
  + <script src=”myscript.js”></script>
* For your Python files, don’t put all of the new API routes into your old controller files! Make a new directory /api and put your API Python files in there. If you have trouble with registering blueprints and importing python files, use the other controllers as a reference.

# Part 7: Grading and Deliverables

Make sure that all these element IDs are present in your HTML templates:

* / (not logged in)
  + The link for login should have id home\_login
  + The link for account creation (/user) should have id home\_user\_create
  + The links to public albums should have ids album\_<albumid>\_link
* /user
  + The form should have an id of new\_user
  + The input for username should have an id new\_username\_input
  + The input for firstname should have an id new\_firstname\_input
  + The input for lastname should have an id new\_lastname\_input
  + The input for email should have an id new\_email\_input

The input for your first password field should have an id new\_password1\_input

* + The input for your second password field should have an id new\_password2\_input
  + The submit button for updating should have an id new\_submit
  + Follow the rules for validation input error as specified in the validation section; all error <p>’s need to have class error and the correct error message
* /user/edit
  + The form should have an id of update\_user
  + The input for firstname should have an id update\_firstname\_input
  + The input for lastname should have an id update\_lastname\_input
  + The input for email should have an id update\_email\_input
  + The input for your first password field should have an id update\_password1\_input
  + The input for your second password field should have an id update\_password2\_input
  + Follow the rules for validation input error as specified in the validation section; all error <p>’s need to have class error and the correct error message
  + The submit button for updating should have an id update\_submit
* /login
  + The input for username should have an id login\_username\_input
  + The input for password should have an id login\_password\_input
  + The submit button or link should have an id of login\_submit
* /albums
  + Each link to /album?albumid=<albumid> should have id album\_<albumid>\_link
* /albums/edit
  + Each link to /album/edit?albumid=<albumid> should have id album\_edit\_<albumid>\_link
* /album
  + Each “link” to /pic?picid=<picid> should have id pic\_<picid>\_link. An href should not be present.
* /album/edit
  + Each link to /pic?picid=<picid> should have id pic\_<picid>\_link
  + The radio button for public album toggle should have id album\_edit\_public\_radio
  + The radio button for private album toggle should have id album\_edit\_private\_radio
  + The submit button for toggling album access should have id album\_edit\_access\_submit
  + Each “remove access” submit button in the access table should have id album\_edit\_revoke\_<username>
  + The text input for granting new access should have id album\_edit\_grant\_input
  + The submit button for granting new access should have id album\_edit\_grant\_submit
* /pic
  + The <p> that shows the caption should have id pic\_<picid>\_caption
    - Only shown if does not have permission to edit caption (see /pic on captions)
  + The text input for editing the caption should have id pic\_caption\_input
    - Only shown if current user has permission to edit caption (see /pic on captions)
  + The link to the next picture in the album should be next\_pic
  + The link to the previous picture in the album should be prev\_pic
* <all signed in pages>
  + The “Nav Home” link should have an id nav\_home¨
  + The “Edit Account” link should have an id nav\_edit
  + The “My Albums” link should have an id nav\_albums
  + The “Logout” button should have an id nav\_logout

NOTE: While some of the ids from project 1 and project 2 were explicitly specified above, you should maintain **all** of the ids we gave you in project 1 and project 2 (except where noted as different); we can and will test you on them!

The autograder will be using Firefox 44 for its testing. You should be testing your application in this browser to ensure correctness.

### Code

You will not need to submit any files to the autograder. We will be testing your deployed site directly.

We will post a link to the autograder when it is ready for this project.

In the README.md at the root of your repository please provide the following details:

* Group Name (if you have one)
* List the contribution for each team member:   
  User Name (uniqname): "agreed upon" contributions
* Any need-to-know comments about your site design or implementation.

We will check the P3 directory for your new secure photo album website. Based on P2, your website should contain at least the following users with their albums. **Remember that in this project, we are hashing the passwords, so you will have to calculate the data for the initial input**.

* Username: sportslover, Password: paulpass93 - "I love sports" (public), "I love football" (private)
* Username: traveler, Password: rebeccapass15 - "Around The World" (public)
* Username: spacejunkie, Password: bob1pass - "Cool Space Shots" (private)

Please make sure your albums have the correct ordered id: these should be the same as P1 and consistent with the order shown above. Your website may contain other users and albums, but please ensure that the above users and albums exist. Do not touch the files in the P3 sub-directory after the deadline.

As mentioned before, **Remember to commit your code into GitHub and the server, please do not modify your code after the due date - either on the repo or the server**, or else we will assume your submission is late.