EECS 485 Project 3: Client-side Dynamic Pages

Due 9pm Saturday, Feb 20, 2016

A prominent example of client-side dynamic pages is the Facebook activity feed, which updates live without clicking a button or reloading the page. Client-side applications impact the frontend, (that is, the HTML/CSS/JS), and also the backend. Backends use REST API principles, which creates a common interface (the API) used simultaneously by frontend web applications, mobile applications, internet of things (IoT) devices, etc.

In this project, you will continue working on the photo album website developed in project 2. You will modify a portion of project 2 to use client-side instead of server-side dynamic pages. You will also add a new caption feature. In this assignment, you will implement a REST API and program in JavaScript.

Part 1: Implementation Preliminaries

Please do not touch the files in the pa2 sub-directory. Make another sub-directory called pa3, and copy the files from pa2 into the pa3 sub-directory and work on the files there.

We are reusing your PA1 databases for this project, databases named: groupXXpa1.

More importantly, you should have received an email with a new secret key to be used for website deployment. So, be sure to change your routes to `{new-secret-key}/pa3/`. In general, we recommend using the blueprint argument url_prefix for specifying these routes.

Part 2: Build the website

In this project, we will be taking your project two and converting it to a Javascript-based application. This means that the page updates dynamically by fetching data in the background via AJAX and inserting the new data into the DOM via Javascript. The routes that require this functionality are listed below. You will begin with getting some XHR/AJAX experience with the user, login and logout functionality before moving on to turning your album and picture routes into its own small Single Page Application. You will also be updating captions through API polling (as part of a single-page Ember app, which is the last part of the project). Use of libraries is allowed.

Implement API and HTML Pages

You will be separating your backend (Flask-python) from your front-end (HTML, JavaScript). Your Flask-python app will now offer API endpoints that only return JSON responses, instead of returning a full template via the render_template function. JSON is an object notation language that lets us return key-value pairs of data in order to serve as a "black box" for the JavaScript to interact with.

What follows is a list of the url endpoints that you should create in your application. You should have created some of these for PA1/PA2. If a page is not listed here, presume it should behave the same way as in Project 2. **All API routes should be prefixed with `/api/v1`.** POST requests to your API and responses from it should specify a MIMETYPE/Content-type of `application/json`. Flask's `jsonify` function will automatically do this for your API responses.

All of your PA2 endpoints should only accept GET requests that will render HTML templates. You should move your controller logic from these controllers to your new API controllers. For example, your /user/edit Flask controller should just render a template with an HTML form. It should not perform session checking. That will be handled in your PUT /api/v1/user controller. Your Javascript will query your /api/v1/user route and insert any errors it encounters with session (see error checking below).

Please note that 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.

New User page: `/user` [public]

This page is for user creation. It contains a form that should `POST` it's 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 use **server-side and client-side Javascript validation** for all data validation. Follow the validation rules in the section later in this document. Please note that you are **not** allowed to use HTML input tag validation.

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 PA2.

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).

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.

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 be `PUT` request to `/api/v1/user.` Validation errors should be dynamically inserted as described in `/user`. Unlike PA2, these fields should not be separate forms. It should all be one form. The HTML form need not be inserted dynamically with Javascript. However, the initial data loading into your HTML form's values should be handled with Javascript and AJAX.

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

The user should be able to change his/her firstname, lastname, password and email address (but not username). Updated information should be `PUT` 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. It should perform the same client-side and server side validation checks as specified in `/user`. Hint: you can use `GET /api/v1/user` to fetch the current session's user.

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. ~~**In addition to** the server-side validation of credentials you performed in project 2, you should perform the same validation checks in Javascript. If an error occurs, you should insert the error node into the DOM with the proper IDs. Client-side~~ checking of /login is not required.

Upon successful login, redirect. 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 helpful. If a user is already logged in, redirect to /user/edit (via Flask).

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

Used in `/login`. A `POST` request to `/api/v1/login` with a JSON object of \`{"`username": "username`" and "`password": "password"}`\` should sign in the user. If successful, it should return a JSON object \``{"username": "username"}`\` where username is the signed-in user.

Logout: `All Pages` [sensitive]

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

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

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

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

This page displays the thumbnail view of an album’s pictures just like the previous assignments. 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?id=<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 so that a user can navigate back to this page (refer to URL History 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?id=<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()` helpful)
4. User clicks a picture to `/pic?id=<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?id=<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 PA2 (it need not modify data dynamically client-side).

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",
  "lastupdated": "2016-02-02",
  "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.

View picture page: `/pic?id=<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. Clicking them should not change the URL, however.

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.

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.

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/public]

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 Album.lastupdated field for the album that the pic is in.

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 standard has been proposed and implemented in all major browsers. For a brief introduction, you should read this 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()` function when updating your DOM and register an onpopstate handler for when the browser's history state changes.

Part 4: Authentication and Session Management

Just as before, we will be implementing sessions. For this project, you should perform the following checks for the following routes listed above: `pic` and `album`. All API routes should also do session checking and return the appropriate error message specified in Error Checking You can reuse the same session checking as from PA2 for the other routes above.

1. If a session exists, render the page via Flask and display the name of the user currently signed in.
2. 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.
3. 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.
4. If a session does not exist and the page is public, perform necessary AJAX requests to render the content into the page.
5. If a session does not exist and 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. 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."
        }
   ]
}
```

Please note that errors is an array of all errors that occur for the page. For example, a user cannot receive a 422 error if they are not first authenticated. Errors denoted with an asterisk (*) signal that if you encounter the error, you should return immediately with that error message not continue processing the other validation checks. 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*
- ~~`GET /api/v1/user/<username>`~~
  - ~~A valid session is required, “You do not have the necessary credentials for the resource”, 401*~~
  - ~~Username does not exist, “Username does not exist”, 404*~~
- `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. Empty strings are valid for firstname and lastname
    - 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 follow the error rules specified for 404. “The requested resource could not be found”~~
  - 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`
    - Except for password, all fields (`username`, `firstname`, `lastname`, and `email`) 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 follow the error rules specified for 404
    - 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?id=<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?id=<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?id=<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 follow the error rules specified for 404
    - 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 into your password database (which means you do not need to check things beyond above rules).

Part 6: Here Comes the Framework

After going through the previous parts, one can quickly realize the complexity involved with creating a single page applications. For large-scale applications that have multiple components (and components more complex than just a simple input and text), the code can quickly become messy. Since every group implemented this in their own way, it would be difficult to switch groups and understand their codebase. Enter frontend (client-side) MVC frameworks whose aim is to provide a consistent way to interact with your server, provide a consistent source of truth data store and push those updates to all of the components that need the data. This also goes back the other way (from view component to server).

In this part, we will use Ember to provide a unified way to address these problems. Before heading any further, it is suggested you read through Ember’s documentation to understand some of the terminology and what is happening. While not necessary, reading through Ember’s guides may prove helpful (~2-3 hrs of reading), but most importantly understanding the Core Concepts. There are guides on templates, controllers, components, models, etc.

Part 6 will be using Ember, a frontend Javascript MVC framework for single page web applications. We’ve chosen Ember instead of other frameworks due to its terminology being easier to understand, good documentation, and its widespread popularity. It will be your group’s responsibility to read documentation to meet the project’s requirements. Remember, this class, especially this project, are not about learning the framework. It’s about learning why the framework exists (what problems does it solve) and how does it solve them. Much of the code is given, so this portion of the project will be weighted to reflect that.

Part 6's web page will live separately from our traditional application at the url /secretkey/pa3/live. Flask need not render a Jinja template, as we will be serving Handlebars templates that are rendered client-side by Ember's Handlebars compiler. In order to accomplish this, you should use Flask's send_file function which skips the Jinja process. **Because of this, there are no necessary session/page access checks to be performed. You also should not need to modify the current user's session when entering this page (including the lastupdated session time). This page is public** Here is an example Flask controller:

```
    @main.route('/secretkey/pa3/live')
    def live_route():
        return send_file('static/html/live.html')
```

Please note that the directory where this lives may be different for your project. It is not required to live in the views folder.

Within live.html, we will be building our Ember view. This view will have the following features:
* Display the image for the picid passed in as a query parameter
* Display the caption and the ability to edit that caption using an AJAX request

Set-up
First, create a view/template called live.html with the following code:

```
<!-- static/html/live.html or perhaps templates/live.html -->
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN">
<html lang="en">
<head>

  <link rel="stylesheet" href="/static/css/style.css" />
  <script src="//code.jquery.com/jquery-2.1.4.min.js"></script>
  <script type="text/javascript"
src="https://cdnjs.cloudflare.com/ajax/libs/handlebars.js/4.0.3/handl
ebars.js"></script>
  <script type="text/javascript"
src="http://builds.emberjs.com/release/ember-template-compiler.js"></
script>
  <script type="text/javascript"
src="http://builds.emberjs.com/release/ember.debug.js"></script>
  <script type="text/javascript"
src="http://builds.emberjs.com/release/ember-data.js"></script>
```

```
  <script type="text/javascript"
src="https://cdnjs.cloudflare.com/ajax/libs/moment.js/2.10.6/moment.m
in.js"></script>
  <script type="text/javascript"
src="/static/js/app/app.js"></script>
  <title>EECS485</title>
</head>
<body bgcolor="#66FFCC">
  <script type="text/x-handlebars" data-template-name="pic">
  </script>
</body>
</html>
```

In this file we download the Handlebars templating library, the Ember template compiler module, a Ember version meant to be help with development and the ember data module. Next, we set-up three scripts in our body that will encapsulate the Ember template code for our view. We will have a “pic” template that will render the information about the picture. Within it, we will render the comment-submit component which will handle comment submission. Before the page is rendered (client-side), Handlebars will take the Handlebars template code within those scripts and compile them to native HTML that the browser can render.

The last step that may prove valuable is installing the Ember Inspector for Google Chrome. Utilize this to inspect your routes, data, etc.

Creating the Ember Application
Next, let’s create the app.js which will contain our Ember application.

```
// /static/js/app/app.js
window.App = Ember.Application.create();

App.Router = Ember.Router.extend({
  rootURL: '/secretkey/pa3/live'
});
```

We create an Ember application and attach it to the window object with the name App. This means we can refer to it at any point by using “App”. You can even type it into your web browser’s developer Javascript console to modify it or inspect it. The next thing we do is run the initializer. Whenever the Ember app is created, it will run this function. ~~What we have specified here will play a role in allowing username validation. We have created an Ember “service” and injected it within our other parts of the codebase (controllers and routes). We’ve also taken our main data store (see below) and made it available to the username-validator service so that it can query the data store.~~ After creating the Ember app, we set a configuration option on its router called the rootURL, which let's Ember know what page is the root page for our Ember application.

```
App.Store = DS.Store.extend({});

App.ApplicationAdapter = DS.JSONAPIAdapter.extend({
  namespace: '/secretkey/pa3/jsonapi/v1'
})
```

we let Ember's Data Store (DS) know where our data will be coming from. Ember has several adapters for how to serialize and deserialize data. By default, it uses the JSONAPI adapter. It expects data coming from our backend REST API to fit the JSONAPI specification, and our backend REST API will be sent data in the format of the JSONAPI specification.

```
App.Router.map(function() {
  this.route('pic', { path: '/pic/:pic_id' });
});
```

Ember's Router helps determine what page a user is on and what information should be loaded. Earlier client side MVC frameworks did not include routers, so when a user would navigate to view their profile, the url would not change. This meant users could not copy and share links and would lose their browser history. It hides the complexity of the HTML5 History API that you used in the previous part of the project. This problem is no longer an issue as major frameworks now provide a router that supports the API. Here we see the influence that client side frameworks have had on pushing the modern web browser APIs forward.

Our routes will be located at a URL similar to the following:
`http://hostname:portno/secretkey/pa3/live#/pic/football_s3`

Here we have defined one route for our picture.

Let's now create our client side Ember data store for the picture. We want to retrieve the picture's file path (pic), the previous and next pictures in the album based on the sequence number, the picture caption and all associated favorites.

```
// app.js
App.Pic = DS.Model.extend({
  picurl: DS.attr('string'),
  prevpicid: DS.attr('string'),
  nextpicid: DS.attr('string'),
  caption: DS.attr('string'),
});
```

```
App.PicRoute = Ember.Route.extend({
  model: function(params) {
    var pic = this.store.findRecord('pic', params.pic_id);
    return pic;
  },

  actions: {
    save: function() {
      var pic = this.modelFor('pic');
      var caption = this.modelFor('pic').get('caption');
      this.set('caption', caption);
      this.modelFor('pic').save();
    }
  },

  renderTemplate: function() {
    this.render('pic');
  }
});
```

~~Here we define the model Favorite. Also specified are the models for the routes (what should happen when a route requests a model). In this case, pic will find the record in our data store. The data store will in turn query the server if it does not have the information present.~~

For a given model, the data store should be the single source of truth. Why is this? You can imagine we add on another route that will display all the pictures that two friends are tagged in together. Instead of having a third data store that repeats the code for getting both users' pictures, it would be wise to just fetch from the two separate data stores. Any other store that depends on them will have the correct information most recently fetched from the server. There will not be inconsistencies between your templates.

Notice how we've established an action called save on our "pic" route. This will be called when someone triggers the action from our template (as you'll soon see).

Now that we have our models and routing set-up, let's modify our templates to render the necessary information from our data stores.

```
<!-- live.html -->
<script type="text/x-handlebars" data-template-name="pic">
  <h1>Picture</h1>
  <img src="/static/images/{{model.picurl}}">
  <br>
  {{#link-to "pic" model.prevpicid}}Previous{{/link-to}}
  {{#link-to "pic" model.nextpicid}}Next{{/link-to}}
  <br>
  <b>Caption: </b>{{input type="text" value=model.caption
enter="save"}}
</script>
```

We've created a template where our model information can be rendered. It will display the picture and links to the previous and next picture models will be dynamically generated. Unlike our previous server-rendered set-up, Ember will load the data models for the previous and next pictures asynchronously. No full-page reload required!

This template is also responsible for displaying the current caption and allowing it to be modified (or added if there is no current caption set on a pic). The "enter" attribute specifies that on enter our "save" action should be fired off. The save action will then modify our data store which then sends a PATCH or a POST request to the server.

Your template does not need the ability to view all of an album's pictures. Visitors will be viewing an album's pictures through the picid and prev/next links.

So what does the API backend look like? It will be similar to the set-up from before, but this time it must follow the JSONAPI specification. An example ember_api.py ~~api.py~~ file has been provided for you in the repo to help with implementing the API routes. Please modify as necessary to get working for your group's project. We will not be testing your API routes for this portion of the project, only the frontend interactions.

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?id=<albumid>` should have id `album_<albumid>_link`
- `/albums/edit`
    - Each link to `/album/edit?id=<albumid>` should have id `album_edit_<albumid>_link`
- `/album`
    - Each "link" to `/pic?id=<picid>` should have id `pic_<picid>_link.` An href should not be present.
- `/album/edit`
    - Each link to `/pic?id=<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 in 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

Submit the following files to the autograder:

- `source.tar.gz` A tar archive containing your application source code.
- `tbl_create.sql`
- `load_data.sql`

How to create a tarball of your source code, from your git repository:

```
    git archive --format tar.gz HEAD > source.tar.gz
```

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 pa3 directory for your new secure photo album website. Based on PA2, 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 pa1 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 pa2 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.