Programación web de CS50 con Python y JavaScript

OpenCourseWare

Donar (https://cs50.harvard.edu/donate)

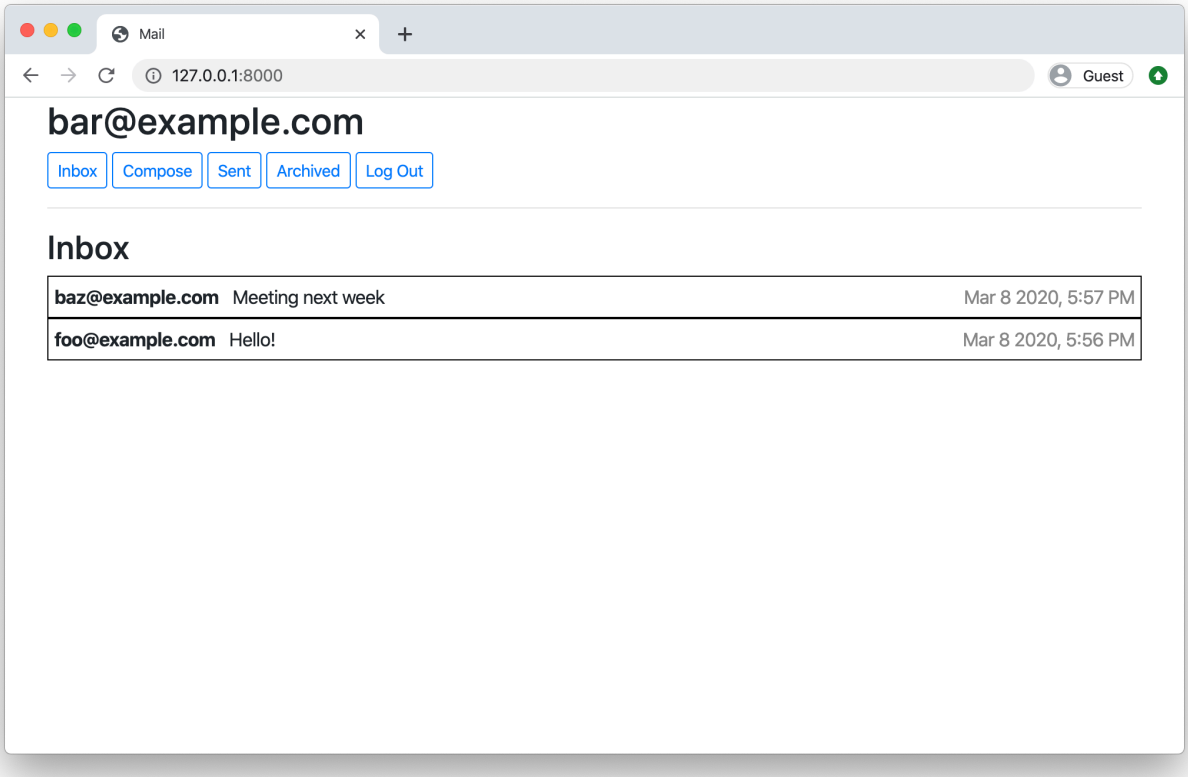
Brian Yu (https://brianyu.me)
brian@cs.harvard.edu

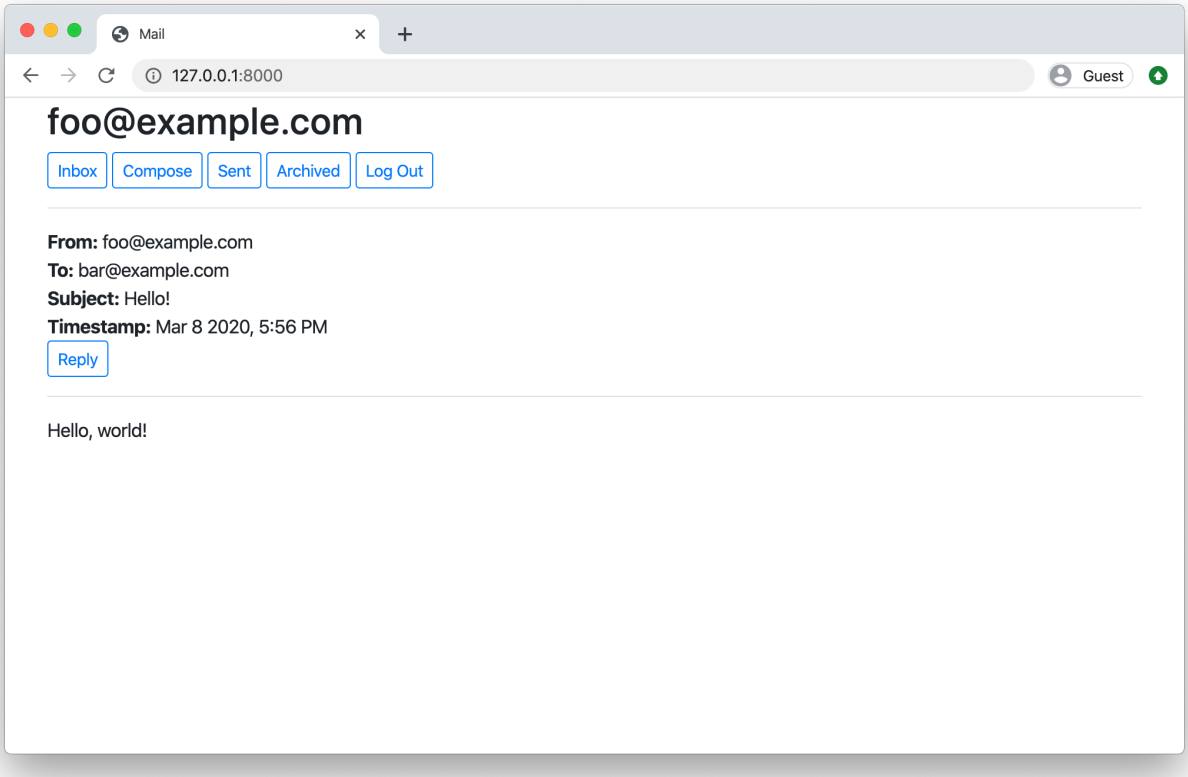
David J. Malan (https://cs.harvard.edu/malan/)
malan@harvard.edu
(https://www.facebook.com/dmalan) (https://github.com/dmalan)
(https://www.instagram.com/davidjmalan/) (https://www.linkedin.com/in/malan/)
(https://www.reddit.com/user/davidjmalan)
(https://www.threads.net/@davidjmalan) (https://twitter.com/davidjmalan)

Correo

> CS50W no presenta una correspondencia uno a uno entre conferencias y proyectos. Si está intentando este proyecto sin haber visto al menos la Conferencia 5, ¡lo está intentando demasiado pronto!

Diseñe una interfaz para un cliente de correo electrónico que realice llamadas API para enviar y recibir correos electrónicos.





Empezando

1. Descargue el código de distribución de https://cdn.cs50.net/web/2020/spring/projects/3/mail.zip

(https://cdn.cs50.net/web/2020/spring/projects/3/mail.zip) y descomprímalo.

2. En tu terminal, `cd` en el `mail` directorio.
3. Ejecute `python manage.py makemigrations mail` para realizar migraciones para la `mail` aplicación.
4. Ejecute `python manage.py migrate` para aplicar migraciones a su base de datos.

Comprensión

En el código de distribución hay un proyecto Django llamado `project3` que contiene una única aplicación llamada `mail`.

Primero, después de realizar y aplicar las migraciones para el proyecto, ejecute `python manage.py runserver` para iniciar el servidor web. Abra el servidor web en su navegador y utilice el enlace "Registrarse" para registrarse para obtener una nueva cuenta. Los correos electrónicos que enviará y recibirá en este proyecto se almacenarán completamente en su base de datos (en realidad, no se enviarán a servidores de correo electrónico reales), por lo que puede elegir cualquier dirección de correo electrónico (por ejemplo, `foo@example.com`) y contraseña que desee. Me gustaría para este proyecto: las credenciales no necesitan ser credenciales válidas para direcciones de correo electrónico reales.

Una vez que haya iniciado sesión, debería ser llevado a la página Bandeja de entrada del cliente de correo, aunque esta página está prácticamente en blanco (por ahora). Haga clic en los botones para navegar a sus buzones de correo Enviados y Archivados, y observe cómo esos también están actualmente en blanco. Haga clic en el botón "Redactar" y accederá a un formulario que le permitirá redactar un nuevo correo electrónico. Sin embargo, cada vez que hace clic en un botón, no se lo lleva a una nueva ruta ni se realiza una nueva solicitud web: en cambio, toda esta aplicación es solo una página, con JavaScript utilizado para controlar la interfaz de usuario. Ahora echemos un vistazo más de cerca al código de distribución para ver cómo funciona.

Eche un vistazo `mail/urls.py` y observe que la ruta predeterminada carga una `index` función en formato `views.py`. Así que abramos `views.py` y veamos la `index` función. Tenga en cuenta que, siempre que el usuario haya iniciado sesión, esta función representa la `mail/inbox.html` plantilla. Miremos esa plantilla, almacenada en `mail/templates/mail/inbox.html`. Notarás que en el cuerpo de la página, la dirección de correo electrónico del usuario se muestra primero en un `h2` elemento. Después de eso, la página tiene una secuencia de botones para navegar entre varias páginas de la aplicación. Debajo de eso, observe que esta página tiene dos secciones principales, cada una definida por un `div` elemento. El primero (con un `id` of `emails-view`) contiene el contenido de un buzón de correo electrónico (inicialmente vacío). El segundo (con un `id` of `compose-view`) contiene un formulario donde el usuario puede redactar un nuevo correo electrónico. Los botones en la parte superior, entonces, deben mostrar y ocultar selectivamente estas vistas: el botón redactar,

por ejemplo, debe ocultar `emails-view` y mostrar `compose-view`; Mientras tanto, el botón de la bandeja de entrada debería ocultar `compose-view` y mostrar el `emails-view`.

¿Cómo lo hacen? Observe que en la parte inferior de se incluye `inbox.html` el archivo JavaScript . `mail/inbox.js` Abra ese archivo, almacenado en `mail/static/mail/inbox.js` y eche un vistazo. Observe que cuando se carga el contenido DOM de la página, adjuntamos detectores de eventos a cada uno de los botones. Cuando `inbox` se hace clic en el botón, por ejemplo, llamamos a la `load_mailbox` función con el argumento `'inbox'`; mientras tanto, cuando `compose` se hace clic en el botón, llamamos a la `compose_email` función. ¿Qué hacen estas funciones? La `compose_email` función primero oculta `emails-view` (estableciendo su `style.display` propiedad en `none`) y muestra `compose-view` (estableciendo su `style.display` propiedad en `block`). Después de eso, la función toma todos los campos de entrada del formulario (donde el usuario puede escribir la dirección de correo electrónico del destinatario, la línea de asunto y el cuerpo del correo electrónico) y establece su valor en la cadena vacía `''` para borrarlos. Esto significa que cada vez que haga clic en el botón "Redactar", se le presentará un formulario de correo electrónico en blanco: puede probar esto escribiendo valores en el formulario, cambiando la vista a la Bandeja de entrada y luego volviendo a la vista Redactar.

Meanwhile, the `load_mailbox` function first shows the `emails-view` and hides the `compose-view`. The `load_mailbox` function also takes an argument, which will be the name of the mailbox that the user is trying to view. For this project, you'll design an email client with three mailboxes: an `inbox`, a `sent` mailbox of all sent mail, and an `archive` of emails that were once in the inbox but have since been archived. The argument to `load_mailbox`, then, will be one of those three values, and the `load_mailbox` function displays the name of the selected mailbox by updating the `innerHTML` of the `emails-view` (after capitalizing the first character). This is why, when you choose a mailbox name in the browser, you see the name of that mailbox (capitalized) appear in the DOM: the `load_mailbox` function is updating the `emails-view` to include the appropriate text.

Of course, this application is incomplete. All of the mailboxes simply show the name of the mailbox (Inbox, Sent, Archive) but don't actually show any emails yet. There's no view yet to actually see the contents of any email. And the compose form will let you type in the contents of an email, but the button to send the email doesn't actually do anything. That's where you come in!

API

You'll get mail, send mail, and update emails by using this application's API. We've written the entire API for you (and documented it below), so that you can use it in your JavaScript code. (In fact, note that we have written **all** of the Python code for you for this project. You should be able to complete this project by just writing HTML and JavaScript).

This application supports the following API routes:

`GET /emails/<str:mailbox>`

Sending a `GET` request to `/emails/<mailbox>` where `<mailbox>` is either `inbox`, `sent`, or `archive` will return back to you (in JSON form) a list of all emails in that mailbox, in reverse chronological order. For example, if you send a `GET` request to `/emails/inbox`, you might get a JSON response like the below (representing two emails):

```
[
    {
        "id": 100,
        "sender": "foo@example.com",
        "recipients": ["bar@example.com"],
        "subject": "Hello!",
        "body": "Hello, world!",
        "timestamp": "Jan 2 2020, 12:00 AM",
        "read": false,
        "archived": false
    },
    {
        "id": 95,
        "sender": "baz@example.com",
        "recipients": ["bar@example.com"],
        "subject": "Meeting Tomorrow",
        "body": "What time are we meeting?",
        "timestamp": "Jan 1 2020, 12:00 AM",
        "read": true,
        "archived": false
    }
]
```

Notice that each email specifies its `id` (a unique identifier), a `sender` email address, an array of `recipients`, a string for `subject`, `body`, and `timestamp`, as well as two boolean values indicating whether the email has been `read` and whether the email has been `archived`.

How would you get access to such values in JavaScript? Recall that in JavaScript, you can use `fetch` to make a web request. Therefore, the following JavaScript code

```
fetch('/emails/inbox')
.then(response => response.json())
.then(emails => {
    // Print emails
    console.log(emails);

    // ... do something else with emails ...
});
```

would make a `GET` request to `/emails/inbox`, convert the resulting response into JSON, and then provide to you the array of emails inside of the variable `emails`. You can print that value out to the browser's console using `console.log` (if you don't have any emails in your inbox, this will be an empty array), or do something else with that array.

Note also that if you request an invalid mailbox (anything other than `inbox`, `sent`, or `archive`), you'll instead get back the JSON response `{"error": "Invalid mailbox."}`.

`GET /emails/<int:email_id>`

Sending a `GET` request to `/emails/email_id` where `email_id` is an integer id for an email will return a JSON representation of the email, like the below:

```
{
        "id": 100,
        "sender": "foo@example.com",
        "recipients": ["bar@example.com"],
        "subject": "Hello!",
        "body": "Hello, world!",
        "timestamp": "Jan 2 2020, 12:00 AM",
        "read": false,
        "archived": false
}
```

Note that if the email doesn't exist, or if the user does not have access to the email, the route instead return a 404 Not Found error with a JSON response of `{"error": "Email not found."}`.

To get email number 100, for example, you might write JavaScript code like

```
fetch('/emails/100')
.then(response => response.json())
.then(email => {
    // Print email
    console.log(email);

    // ... do something else with email ...
});
```

`POST /emails`

So far, we've seen how to get emails: either all of the emails in a mailbox, or just a single email. To send an email, you can send a `POST` request to the `/emails` route. The route requires three pieces of data to be submitted: a `recipients` value (a comma-separated string of all users to send an email to), a `subject` string, and a `body` string. For example, you could write JavaScript code like

```
fetch('/emails', {
  method: 'POST',
  body: JSON.stringify({
      recipients: 'baz@example.com',
      subject: 'Meeting time',
      body: 'How about we meet tomorrow at 3pm?'
  })
})
```

```
.then(response => response.json())
.then(result => {
    // Print result
    console.log(result);
});
```

If the email is sent successfully, the route will respond with a 201 status code and a JSON response of `{"message": "Email sent successfully."}`.

Note that there must be at least one email recipient: if one isn't provided, the route will instead respond with a 400 status code and a JSON response of `{"error": "At least one recipient required."}`. All recipients must also be valid users who have registered on this particular web application: if you try to send an email to `baz@example.com` but there is no user with that email address, you'll get a JSON response of `{"error": "User with email baz@example.com does not exist."}`.

`PUT /emails/<int:email_id>`

The final route that you'll need is the ability to mark an email as read/unread or as archived/unarchived. To do so, send a `PUT` request (instead of a `GET`) request to `/emails/<email_id>` where `email_id` is the id of the email you're trying to modify. For example, JavaScript code like

```
fetch('/emails/100', {
  method: 'PUT',
  body: JSON.stringify({
      archived: true
  })
})
```

would mark email number 100 as archived. The body of the `PUT` request could also be `{archived: false}` to unarchive the message, and likewise could be either `{read: true}` or `read: false}` to mark the email as read or unread, respectively.

Using these four API routes (getting all emails in a mailbox, getting a single email, sending an email, and updating an existing email), you should have all the tools you now need to complete this project!

Specification

Using JavaScript, HTML, and CSS, complete the implementation of your single-page-app email client inside of `inbox.js` (and not additional or other files; for grading purposes, we're only going to be considering `inbox.js`!). You must fulfill the following requirements:

- **Send Mail**: When a user submits the email composition form, add JavaScript code to actually send the email.

    - You'll likely want to make a `POST` request to `/emails`, passing in values for `recipients`, `subject`, and `body`.
    - Once the email has been sent, load the user's sent mailbox.
- **Mailbox**: When a user visits their Inbox, Sent mailbox, or Archive, load the appropriate mailbox.
    - You'll likely want to make a `GET` request to `/emails/<mailbox>` to request the emails for a particular mailbox.
    - When a mailbox is visited, the application should first query the API for the latest emails in that mailbox.
    - When a mailbox is visited, the name of the mailbox should appear at the top of the page (this part is done for you).
    - Each email should then be rendered in its own box (e.g. as a `<div>` with a border) that displays who the email is from, what the subject line is, and the timestamp of the email.
    - If the email is unread, it should appear with a white background. If the email has been read, it should appear with a gray background.
- **View Email**: When a user clicks on an email, the user should be taken to a view where they see the content of that email.
    - You'll likely want to make a `GET` request to `/emails/<email_id>` to request the email.
    - Your application should show the email's sender, recipients, subject, timestamp, and body.
    - You'll likely want to add an additional `div` to `inbox.html` (in addition to `emails-view` and `compose-view`) for displaying the email. Be sure to update your code to hide and show the right views when navigation options are clicked.
    - See the hint in the Hints section about how to add an event listener to an HTML element that you've added to the DOM.
    - Once the email has been clicked on, you should mark the email as read. Recall that you can send a `PUT` request to `/emails/<email_id>` to update whether an email is read or not.
- **Archive and Unarchive**: Allow users to archive and unarchive emails that they have received.
    - When viewing an Inbox email, the user should be presented with a button that lets them archive the email. When viewing an Archive email, the user should be presented with a button that lets them unarchive the email. This requirement does not apply to emails in the Sent mailbox.
    - Recall that you can send a `PUT` request to `/emails/<email_id>` to mark an email as archived or unarchived.
    - Once an email has been archived or unarchived, load the user's inbox.
- **Reply**: Allow users to reply to an email.

- When viewing an email, the user should be presented with a "Reply" button that lets them reply to the email.
- When the user clicks the "Reply" button, they should be taken to the email composition form.
- Pre-fill the composition form with the `recipient` field set to whoever sent the original email.
- Pre-fill the `subject` line. If the original email had a subject line of `foo`, the new subject line should be `Re: foo`. (If the subject line already begins with `Re:`, no need to add it again.)
- Pre-fill the `body` of the email with a line like `"On Jan 1 2020, 12:00 AM foo@example.com wrote:"` followed by the original text of the email.

Hints

- To create an HTML element and add an event handler to it, you can use JavaScript code like the below:

```
const element = document.createElement('div');
element.innerHTML = 'This is the content of the div.';
element.addEventListener('click', function() {
    console.log('This element has been clicked!')
});
document.querySelector('#container').append(element);
```

This code creates a new `div` element, sets its `innerHTML`, adds an event handler to run a particular function when that `div` is clicked on, and then adds it to an HTML element whose `id` is `container` (this code assumes that there is a HTML element whose `id` is `container`: you'll likely want to change the argument to `querySelector` to be whichever element you'd like to add an element to).

- You may find it helpful to edit `mail/static/mail/styles.css` to add any CSS you need for the application.
- Recall that if you have a JavaScript array, you can loop over each element of that array using `forEach` (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/forEach).
- Recall that normally, for `POST` and `PUT` requests, Django requires a CSRF token to guard against potential cross-site request forgery attacks. For this project, we've intentionally made the API routes CSRF-exempt, so you won't need a token. In a real-world project, though, always best to guard against such potential vulnerabilities!

How to Submit

1. Visit this link (https://submit.cs50.io/invites/89679428401548238ceb022f141b9947), log in with your GitHub account, and click **Authorize cs50**. Then, check the box indicating that you'd like to grant course staff access to your submissions, and click **Join course**.
2. Install Git (https://git-scm.com/downloads) and, optionally, install `submit50` (https://cs50.readthedocs.io/submit50/).

> When you submit your project, the contents of your `web50/projects/2020/x/mail` branch should match the file structure of the unzipped distribution code as originally received. That is to say, your files should not be nested inside of any other directories of your own creation. Your branch should also not contain any code from any other projects, only this one. Failure to adhere to this file structure will likely result in your submission being rejected.
>
> By way of example, for this project that means that if the grading staff visits `https://github.com/me50/USERNAME/tree/web50/projects/2020/x/mail` (where `USERNAME` is your own GitHub username as provided in the form, below) we should see the two subdirectories (`mail`, `project3`) and the `manage.py` file. If that's not how your code is organized when you check, reorganize your repository needed to match this paradigm.

3. If you've installed `submit50`, execute

   ```
   submit50 web50/projects/2020/x/mail
   ```

   Otherwise, using Git, push your work to `https://github.com/me50/USERNAME.git`, where `USERNAME` is your GitHub username, on a branch called `web50/projects/2020/x/mail`.
4. Grabe un screencast (https://www.howtogeek.com/205742/how-to-record-your-windows-mac-linux-android-or-ios-screen/) que no exceda los 5 minutos de duración (y no lo cargue más de un mes antes de enviar este proyecto), en el que demuestre la funcionalidad de su proyecto. Asegúrese de que cada elemento de la especificación anterior se demuestre en su video. No es necesario mostrar su código en este video, solo su aplicación en acción; Revisaremos su código en GitHub. Sube ese video a YouTube (https://www.youtube.com/upload) (como no listado o público, pero no privado) o en otro lugar. En la descripción de su video, debe marcar la hora en la que su video demuestra cada uno de los cinco (5) elementos de la especificación. **Esto no es opcional** , los videos sin marcas de tiempo en su descripción serán rechazados automáticamente.
5. Envíe este formulario (https://forms.cs50.io/8f569b7d-bd6d-4446-82ac-d65b86d95bbb) .

Luego puede ir a https://cs50.me/cs50w (https://cs50.me/cs50w) para ver su progreso actual.