Web Systems and Technologies **Year 2020-2021**

Lab exercise: *Inbox*

Contents

1 Objectives

The objectives of this exercise are:

- Reinforce the Vue concepts seen in the lectures.
- Use the Vue library to develop the client side and use nodeJS in the server side to implement a single page application (SPA).

2 Functionality

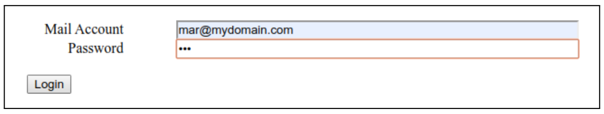
We will develop a web mail service. More specifically, we will implement an online inbox mail service. This online inbox mail service is similar to the `gmail` one from Google.
The inbox functionality to be implemented is the following:

- List the inbox mails.
- Read a mail from the inbox list.
- Compose new mails.
- Forward a mail.
- Reply to a mail.
- Delete a mail.

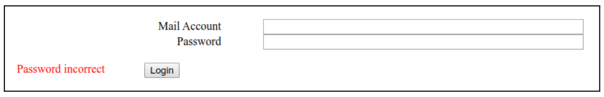
The whole web site will be implemented as a single page application. The following sections describe the application functionality in more detail.

2.1 Login

To access to the inbox it is necessary that the user sings in to the **`mailServer`** through a form as shown in figure 1.



(a) Login form.

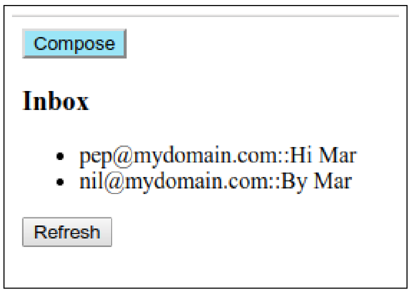


(b) Login error.

Figure 1: User login.

2.2 Inbox

Once the user has signed in it is displayed a list with all the messages she has in the inbox as shown in figure 2. For each



Inbox

- pep@mydomain.com::Hi Mar
- nil@mydomain.com::By Mar

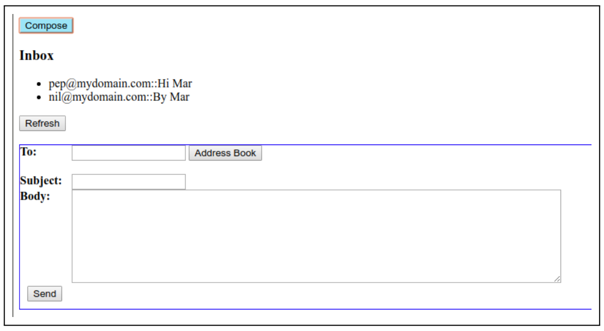
Figure 2: User inbox.

message it is displayed the sender mail address plus the subject. Each entry is clickable. From this view, the user can:

- Compose a message by clicking the compose button.
- Read the received mails.
- Refresh the list of received mails checking for new arrivals.

2.3 Compose a mail

By clicking on the **Compose** button in the **inbox** view it is shown a form to compose a mail as shown in figure 3. From

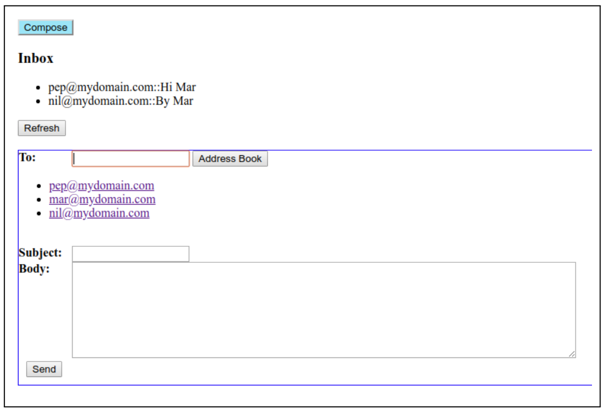


Inbox

- pep@mydomain.com::Hi Mar
- nil@mydomain.com::By Mar

To:
Subject:
Body:

(a) Form to compose a mail.



Inbox

- pep@mydomain.com::Hi Mar
- nil@mydomain.com::By Mar

To:

- pep@mydomain.com
- mar@mydomain.com
- nil@mydomain.com

Subject:
Body:

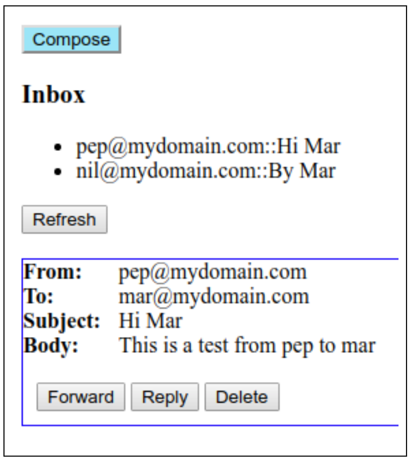
(b) Using the address book to get an address.

Figure 3: Form to compose a mail.

any form you might find along the functionality, by clicking on the **Address Book** button , it is displayed the list of existing mail addresses in the server as shown in 3b. This list is clickable. Once the user selects one mail address from the list, the input field associated with the button is populated with the selected address and the list disappears.

2.4 Read a mail

From the **inbox** view, when the user clicks on one of the mails, the mail information is displayed as shown in figure 4.



Inbox

- pep@mydomain.com::Hi Mar
- nil@mydomain.com::By Mar

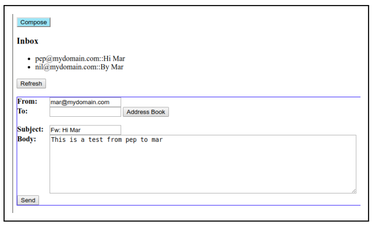
From:	pep@mydomain.com
To:	mar@mydomain.com
Subject:	Hi Mar
Body:	This is a test from pep to mar

Figure 4: Mail content.

From this view the user can forward the mail, reply to it or delete it.

2.5 Forward a mail

Once the forward button is clicked, the user access to a form ready to be fulfilled with the destination mail address as shown in figure 5.

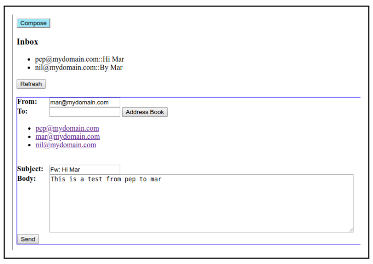


Inbox

- pep@mydomain.com::Hi Mar
- nil@mydomain.com::By Mar

From: mar@mydomain.com
To:
Subject: Fw: Hi Mar
Body: This is a test from pep to mar

(a) The field **To** needs to be set.



Inbox

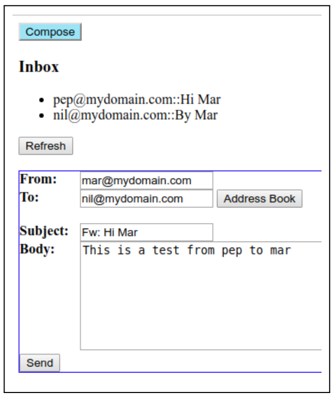
- pep@mydomain.com::Hi Mar
- nil@mydomain.com::By Mar

From: mar@mydomain.com
To:

- pep@mydomain.com
- mar@mydomain.com
- nil@mydomain.com

Subject: Fw: Hi Mar
Body: This is a test from pep to mar

(b) Using the address book to set the field **To**.



Inbox

- pep@mydomain.com::Hi Mar
- nil@mydomain.com::By Mar

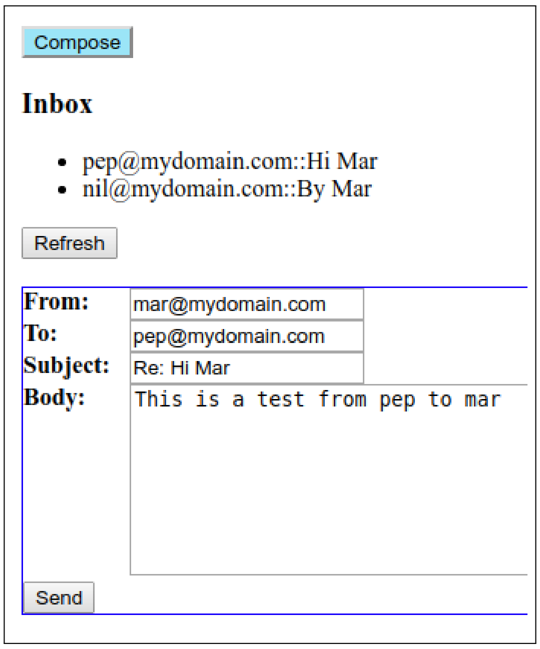
From: mar@mydomain.com
To: nil@mydomain.com
Subject: Fw: Hi Mar
Body: This is a test from pep to mar

(c) Field **To** set.

Figure 5: Forward form.

2.6 Reply to a mail

From the read view, once the reply button is clicked it is shown a form automatically fulfilled as shown in figure 6.



Inbox

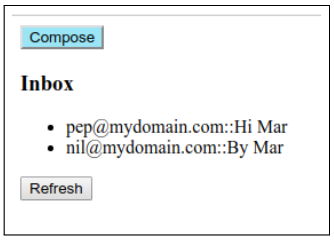
- pep@mydomain.com::Hi Mar
- nil@mydomain.com::By Mar

From: mar@mydomain.com
To: pep@mydomain.com
Subject: Re: Hi Mar
Body: This is a test from pep to mar

Figure 6: Replying to a mail.

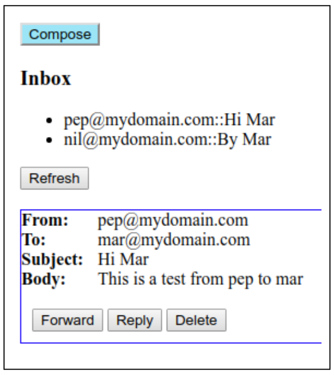
2.7 Delete a mail

From the read view, once the delete button is clicked, the mail is deleted from the user's inbox. The next images sequence show the steps to perform the delete operation.



- pep@mydomain.com::Hi Mar
- nil@mydomain.com::By Mar

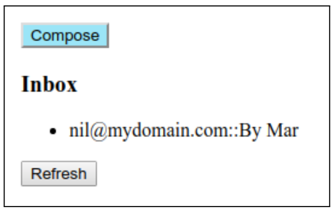
(a) User's inbox.



- pep@mydomain.com::Hi Mar
- nil@mydomain.com::By Mar

From: pep@mydomain.com
To: mar@mydomain.com
Subject: Hi Mar
Body: This is a test from pep to mar

(b) Accessing to the mail sent by pep.



- nil@mydomain.com::By Mar

(c) The mail from pep has been deleted.

Figure 7: Delete a mail.

3 Directory structure of the application

In this section we will explain which is the directory structure and the files needed to implement the application.

3.1 inbox/

This is the application's root directory.

3.2 package.json

This is a json file containing the name and the version of the application's required modules.
The skeleton of the lab provides you with this file. You just need to run the command:

```
npm install
```

to automatically get installed all the modules listed in the file. Those modules will be installed in the local directory **`node-modules`**.

3.3 app.js

This script implements the server. This file will contain:

- The creation of the HTTP server through the `Express` api.
- The middleware to automatically return the static content in the public directory.
- The middleware to implement the access control.
- The routing of the HTTP requests.

The server is executed using the command:

```
cd inbox
node app.js
```

3.4 public

All the static files, as *stylesheet*, the **`vue`** library, and the script containing the vue components go here, in the subdirectories `css` and `js` respectively.

3.5 public/login.html

This is the login form. See figure 1b. Through this form we will send by POST the user mail address and password. The POST request we will be generating by the form submission will be **`POST /login`**. As the form fields are sent by post, the query string goes in the body of the request.

3.6 index.html

It is the single page containing the DOM element where the vue application will be mounted on.

3.7 model/UserDB

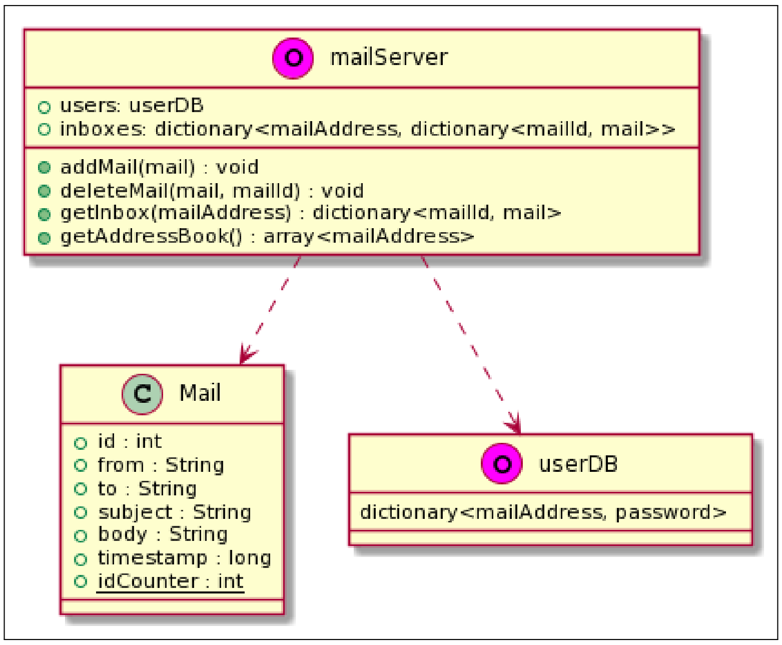
It contains the database with the users of the mail server. See section 4.2.

3.8 model/mailServer.js

It contains the constructor of a Mail object (see section 4.1) and the mail server (see section 4.3).

4 Back-End (server side) design

The back-end consists on the model of the application. It is developed in the **`app.js`** file. It is developed in JavaScript using nodeJS and Express libraries. **We provide you with the model fully coded in the skeleton of the exercise**.
Next figure shows the class-function diagram of the back-end.



O mailServer
users: userDB
inboxes: dictionary<mailAddress, dictionary<mailId, mail>>
addMail(mail) : void
deleteMail(mail, mailId) : void
getInbox(mailAddress) : dictionary<mailId, mail>
getAddressBook() : array<mailAddress>
C Mail
id : int
from : String
to : String
subject : String
body : String
timestamp : long
idCounter : int
O userDB
dictionary<mailAddress, password>

Figure 8: class/object diagram.

Here are some details about the classes and functions of the diagram.

4.1 `function Mail(from, to, subject, body)`

The **`Mail`** class is implemented as a constructor function. Note that the **`idCounter`** attribute is static. This class encapsulates a mail.

4.1.1 Class properties

idCounter: It is a static property. It is a number that is incremented by one each time it is used.

id: It is a number that identifies uniquely a mail. Its value is set in the constructor by using the static property **`idCounter`**.

from: String containing the address of the sender of the message.

to: String containing the address of the receiver of the message.

subject: String containing the subject of the message.

body: String containing the body of the message.

timestamp: A number with the time the mail was created.

4.2 `Single object userDB`

The **`userDB`** is a single object that encapsulates the users database. It is a dictionary. The dictionary is indexed by a user email address (String), for example, `'mar@dom.com'`. The value for each index is the user password (String). This is an example of a possible **`userDB`** object:

```
{
  'pep@dom.com': '123',
  'mar@dom.com': '123'
}
```

4.3 `mailServer`

The **`mailServer`** is a single object. It encapsulates all the functionality of a mail server. That is, contains a set of users and their inboxes, therefore the users' mails.

4.3.1 Object properties

users Contains the userDB single object.

inboxes It is a dictionary. It contains the inboxes of all the users of the domain. This dictionary is indexed by the user's mail address (String). The value for this index is another dictionary containing all the user's mails (inbox). The inbox dictionary is indexed by the mailID and its value is a mail object. Here is an example of a mailServer managing two users: **`pep`** and **`mar`**. **`Pep`** has two mails and **`mar`** just one.

```
{
  'pep@dom.com': {
    1: {
         from: 'mar@dom.com'
         to: 'pep@dom.com'
         subject: 'Hi'
         body: 'Hi Pep, how are you doing? '
         ...
    }
    2: {
         from: 'mar@dom.com'
         to: 'pep@dom.com'
         subject: 'Working'
         body: 'Hi Pep, we need to work in ...'
         ...
    }
  }
  'mar@dom.com': {
    3: {
         from: 'pep@dom.com'
         to: 'mar@dom.com'
         subject: 'Next holidays'
         body: 'Hi Mar, I would like to... '
         ...
       }
  }

} //end mailServer
```

4.3.2 Object methods

See the description of the methods directly from the skeleton code.

5 Server's middleware

The server will use some already built in *Express* middleware with the statement:

```
app.use(express.middlewareXX)
```

The following are the middleware we are going to use. **We provide you with all this middleware already coded in the server**.

5.1 `express.static(path.join(__dirname, "public")`

This Express middleware directly serves to the client the files in the public directory and its sub-directories. The following statement loads this middleware:

```
app.use(express.static(path.join(__dirname, 'public')));
```

To reference to the files in the public directory from an html or js file, you just need to specify the location of such file **relative to the `public` directory**. For instance, in an html file, to reference a stylesheet located in `public/css/style.css`, you will specify its route as follows:

```
<link rel="stylesheet" href="css/style.css">
```

5.2 `express.urlencoded({ extended: true })`

This middleware is used to get the query string parameters sent by POST. The following statement loads this middleware:

```
app.use(express.urlencoded({ extended: true }));
```

5.3 `express.json()`

This middleware enables receiving/sending data from/to the client in json format. The following statement loads this middleware:

```
app.use(express.json());
```

5.4 `express-session`

Express-session is a cookie-based configurable Express middleware. To access to this functionality it is necessary to install the Express module **`express-session`**. See how it works at [16].

```
const session = require('express-session')
```

This module doesn't save session data in the cookie itself, just the session ID. Session data is stored server-side.
By using the function **`session(options)`** you create a session middleware with the given options that configure the session ID cookie. Here is an example of how to use this middleware:

```
app.use(session({
  secret: 'keyboard cat',
  resave: false,
  saveUninitialized: true}));
```

Once loaded this middleware you can create session variables by directly adding a new property to the object **`request.session`**. In the following example we store in the session variable **`currentUser`** a user object:

```
request.session.currentUser = {name:'Mar', suname:'Dofi', age:20};
```

6 Server's REST API

The file `app.js` implements the REST API the server offers to the client to manage the user's inbox. The client's HTTP requests to the REST API will be handled by the `Express` framework through the `application` object. You will find examples of how to do the routing in Ian's master class notes of nodeJS, in the `Express` website in the *basic-routing* [4] and *routing* [5] sections and also in the `application` object api [6].
In the following subsections we describe the REST API the server offers.

6.1 GET /

This is a `GET` HTTP operation that can be handled by the `Express` `application::get(path, callback)` function (`app.get(...)`) [7]. The server needs to reply to this request by sending the `index.html` file. To do so you can use the `Express` function `res.sendFile(path)`. See how to use it at [12].

```
res.sendFile(path.join(__dirname,'index.html'));
```

6.2 POST /login

This functionality is fully developed in the skeleton of the exercise.
This is a `POST` HTTP operation that can be handled by the `Express` `application::post(path, callback)` function (`app.post(...)`) [8].
The client (browser) sends this request when the login form is submitted. The server needs to get from the query string the parameters sent by POST (the mail address and the password) to validate them.
You can access to the POST query string parameters as attributes of the object `request.body`. The following expression shows how to access to the mail address parameter: `request.body.mailAddress`.

Through the `mailServer` object you need to check if the combination of mail address and password exist. If so, the mail account is validated and you will need to do:

1. Store the user's mail address in a `session` variable so the client can perform operations over the inbox of the signed in user.
2. The server need to redirect the request to `GET /` to respond to the client with the `index.html` page. To do so you will need to use the `Express` function `response.redirect([status,] path)` (example: `response.redirect('/')`. See how it works at [13].

On the contrary, if the mail address and password combination does not exist, the server will redirect the request to `/login.html` specifying through a query string parameter the error code:

1: In case the account does not exist.

2: In case the password does not match.

For instance, if the user does not exist the server will redirect the response to the client to `/login.html?error=1`.

6.3 GET /inbox

This is a `GET` HTTP request that can be handled by the `Express` `application::get(path, callback)` function (`app.get(...)`) [7]. The server responds to this request by sending the json with the inbox of the signed in user. **The mail account of the signed in user is already stored in a session variable named `account`.** Through the `mailServer` object you can access to the user's inbox, which is a dictionary. You can send the dictionary to the client directly through the `response` object using the function `res.json([body])` where parameter can be any JSON type, including object, array, string, boolean, number, or null. See how it works at [14]. In the following example we send an object as a json to the client:

```
let user = {name:'Mar', suname:'Dofi', age:20};
response.json(user);
```

6.4 GET /addressBook

This is a `GET` HTTP request that can be handled by the `Express` `application::get(path, callback)` function (`app.get(...)`) [7]. The server sends back the `mailServer addressBook` property using the function `response.json([body])`. See how this function works at [14] and see an example in section 6.3.

6.5 POST /composedMail

This is a **POST** HTTP operation that can be handled by `Express` **`application::post(path, callback)`** function (**`app.post(...)`**) [8].
When the client sends this request includes in the HTTP body, as a parameter, a mail formated as a json object. In the server side, this json object is accessed through **`request.body`**. For instance in **`request.body.from`** you have the sender of the message.
Once you get the mail from the client you need to create an instance of the class **`Mail`** (see section 4.1) out of the received mail data. Remember that you can get the *from* field from the current signed in mail account stored in the session (see section 6.2). Afterwards you need to update the inbox of the receiver by adding the mail.
As you know, the HTTP protocol is an asymmetric stateless request-response client-server protocol. An HTTP client sends a request message to an HTTP server and **the server replies with a response message**. So, after processing the POST request **our server needs to send a response to the client**. The response we will be sending will be just the status code of the response without any data. This code will be **`200 OK`** (the request has succeeded). To do so you can use the *express* function **`response.status(code)`** [15] which is a chainable function. We can chain it to the **`response.end()`** function:

```
response.status(200).end()
```

6.6 DELETE /mail/:mailId

This is a **DELETE** HTTP operation that can be handled by `Express` **`application::delete(path, callback)`** function (**`app.delete(...)`**) [9].
This HTTP request includes a route parameter **`:mailId`**. See how it works in the section *Express* > *Routing* > *Route parameters* [5]. From this section:

”Route parameters are named URL segments that are used to capture the values specified at their position in the URL. The captured values are populated in the **req.params** object, **with the name of the route parameter specified in the path as their respective keys**.”

For example, if the client sends the following DELETE request to delete the mail with id 1 sent by *mar* to *pep*, the **DELETE** HTTP request the server will receive will be:

```
DELETE /mail/1
```

To access to the route parameter **`:mailId`** we would do:

```
request.params.mailId
```

Once the server has deleted the mail sends back to the client the new user’s inbox (in *json* format) which does not contain the deleted mail.

7 Access Control

We provide you with this functionality fully coded in the skel of the exercise.
We will provide a mechanism to avoid illegal access to the application for users that have not signed in. We will implement the logic of this mechanism in a middleware function.

8 Front-End

The following sections are devoted to the development of the client side of the application. This code will be executed in the browser (client). The front-end will be developed by using HTML, CSS, javscript and the vue library.

9 Vue application

In the file `public/js/components.js` we create the application that implements the inbox. We will mount that application into the DOM element identified as `"app"` in the `index.html`.
In the following sections we show you some of the attributes and methods of the vue application, and the communication between the vue application (client) and the server.

9.1 Communication between the Vue application and the server

From the client side we will access the server's REST API to interact with a user's inbox. To do so, from the client we will perform an asynchronous request (ajax) to the server's API by using the client side javascript function **`fetch`** which is similar to **`XMLHttpRequest`** but using promises. You can see how it works in [18]. In the following example, from the client, we perform the ajax **`GET /users`** request to the server to get a a list of users. The server returns the list of users in json format. The execution of line 1 returns a promise containing the response (a Response object). This is just an HTTP response, not the actual JSON. To extract the JSON body content from the response, we use the json() method. **This method also returns a promise.** To retrieve the actual value of the JSON you need to execute line 3.

```
1 fetch('/users')
2 .then(response => response.json())
3 .then(aJson => {console.log(aJson)});
```

In the next example we perform an ajax POST request to the server to send a json. Notice in line 7 how to build a json out of an object or a dictionary.

```
1  let user = {name:'Mar', suname:'Dofi', age:20};
2  fetch('/users', {
3    method: 'POST',
4    headers:{
5      'Content-Type': 'application/json',
6    },
7    body: JSON.stringify(user),
8 })
9 .catch((error) => {
10    console.error('Error:', error);
11  });
```

The following example shows a **`DELETE`** HTTP request which receives a response out of it. Notice in lines 4 and 5 how to receive a json.

```
1 fetch('/mail/1', {
2  method: 'DELETE',
3 })
4 .then(response => response.json())
5 .then(aJson => {console.log(aJson)})
6 .catch(err => {
7   console.error(err)});
```

9.2 Some data properties

In the following section we suggest you **some** of the properties of the **`data object`** of the vue application instance. Feel free to use them or not. You might need to define more.

9.2.1 userInbox

It is a dictionary containing the inbox associated to the user signed in. You get this dictionary from the server. The next example shows the json of **`pep@dom.com`** inbox:

```
{
1: {
from: 'mar@dom.com'
to: 'pep@dom.com'
subject: 'Hi'
body: 'Hi Pep, how are you doing? '
...
2: {
from: 'mar@dom.com'
to: 'pep@dom.com'
subject: 'Working'
body: 'Hi Pep, we need to work in ...'
...
}
}
```

The inbox is a dictionary indexed by the mail id. The value of each entry of the dictionary is an object that encapsulates a mail. Let's say we store the inbox in the variable **`inbox`**. To access to the field **`to`** of the mail in the index **`1`** we woul do:

```
inbox[1].to
```

9.2.2 selectedMail

It is the mail object the user has selected from the inbox by clicking on it.

9.2.3 pollingId

The id associated to the **`setInterval`** function used to do the polling over the mails in the inbox. See the section 9.3.1.

9.3 Vue instance lifecycle hooks

In the process of mounting and unmounting the vue application into a DOM element, we will need to take some actions in the **mounted** and in the **beforeUnmount** lifecycle hooks.

9.3.1 `mounted` lifecycle hook

This method is called when the vue application is mounted into a DOM element. In this method you might need to setup:

- The mail list of the signed in user. The setter of this mail list is the method **`refreshMailList`** 9.4.3.
- The address book of the **`mailServer`**. The setter of the address book is the method **`initAddressBook`** 9.4.6.
- The polling to automatically refresh the mail list each 5 seconds. To do so you can use the method **`setInterval(func|code, [delay])`**. See how it works at [19].

9.3.2 `beforeUnmount` hook

Before unmounting the vue application you can clear the **`setInterval`** to stop the polling over the mail list. You can use the function **`clearInterval(timerId)`**. See how it works at [19].

9.4 Some methods

The following methods are a guide. Feel free to delete them. You will need to define more methods.

9.4.1 `resetDisplay`

Method that sets to false all the components display-flag.

9.4.2 `displayComponent`

This method first of all calls the function **`resetDisplay`** to set all the components display flag to false. Afterwards it sets to true the flag of the component to be displayed. It might need to receive as a parameter an identifier of the component to be displayed.

9.4.3 `refreshMailList`

Method that sends a **`GET /inbox`** HTTP request to the client asking for the inbox of the signed in user. See how to perform an HTTP GET request in section 9.1.

9.4.4 `sendMail(mail)`

This method adds a mail to the inbox of the **`mail.to`** user (the receiver). To do so you will need to perform a **`POST /composedMail`** request to the server specifying in the body of the request the mail to be sent. The mail will be formatted as a json. See how to perform an HTTP POST request in section 9.1. Once the mail is sent, you need to display just the user inbox.

9.4.5 `deleteMail`

This function deletes the mail in the attribute **`selectedMail`**. To perform this operation you will perform an asynchronous **`DELETE`** HTTP request to the server specifying the id of the mail to be deleted as a route parameter:

```
DELETE /mail/mail-id
```

See how to perform an HTTP DELETE request in section 9.1.
The server, as a result of the the delete operation, will send back to the client the new user's inbox where the mail has been deleted. The inbox will be formatted in json. You will need to display the new user's inbox.

9.4.6 `initAddressBook`

This function sends the **`GET /addressBook`** HTTP request to the server to get an array with all the mail addresses the server manages. See how to perform an HTTP GET request in section 9.1.

10 Vue Components

Next sections describe all the vue components needed to manage the online inbox. This functionality is implemented as a single page application (SPA). That means that the **`index.html`** file uses all the vue components.
The components will be defined in the file **`public/js/components.js`** and will be used in the **`index.html`**.

Most of the components' templates are going to be an html form. **We will perform all the requests to the server using ajax (see section 9) which implies that the forms will never be submitted**. We will use the vue event modifier **`prevent`** to prevent the form to be submitted.

```
<form v-on:submit.prevent> </form>
```

All the javascript dictionaries received/sent from/to the server will be in json format.

Next sections describe the vue components we will develop. We use the kebab-case for the name of the components.

10.1 mail-list

This vue component renders the user's inbox view shown in the figure 2. Each mail is listed showing the fields *from* and *subject* of the mail. The component also includes a **compose** and **refresh buttons**. **This component is always visible in the SPA**.

The component might need to receive **`props`**.
Using the **`pep@dom.com`** inbox introduced in section 9.2.1:

```
{
  1: {
       from: 'mar@dom.com'
       to: 'pep@dom.com'
       subject: 'Hi'
       body: 'Hi Pep, how are you doing? '
       ...
  2: {
       from: 'mar@dom.com'
       to: 'pep@dom.com'
       subject: 'Working'
       body: 'Hi Pep, we need to work in ...'
       ...
     }
}
```

This inbox would be displayed as:

```
mar@dom.com::Hi
mar@dom.com::Working
```

Each of the above entries need to be clickable. By clicking on one of them, the mail information is displayed by using the **`mail-reader`** component (see section 10.2). If you use a tag anchor **`<a>`** in order not to reload the current page you need not to set the attribute **`href`**.

```
<a>Here goes the visible link text</a>
```

10.1.1 Component's behavior

You need to manage the following component's use cases:

The user clicks on one mail from the list: This action needs to hide any other visible component except this one and display the **`mail-reader`** component. You might need to use the function **`displayComponent`**, see section 9.4.2.

The user clicks on the compose button: This action needs to hide any other visible component except this one and display the **`mail-composer`** component (figure 3).

The user clicks on the refresh button: You might need to use the function **`refresMailList`** (see section 9.4.3) to refresh the list of mails in the inbox. See in 13.2 how to take out data back to the parent.

10.2 mail-reader

The component template shows, in **plain text** (readonly), the fields *from*, *to*, *subject*, and *body* of the selected mail. It also displays the **forward**, **reply** and **delete buttons** as shown in figure 4. Remember that you already have the signed in user's inbox in a property of the vue application.

The component might need to receive **`props`** for example the mail to be displayed.

10.2.1 Component's behavior

You need to manage the following component's use cases:

The user clicks on the forward button: This action needs to hide any other visible component except the **`mail-list`** one and display the **`mail-forwarder`** component (figure 5). You might need to use the function **`displayComponent`**, see section 9.4.2.

The user clicks on the reply button This action needs to hide any other visible component except the **`mail-list`** one and display the **`mail-replier`** component (figure 6). You might need to use the function **`displayComponent`**, see section 9.4.2 .

The user clicks on the delete button This action needs to hide any other visible component except the **`mail-list`** one (figure 7). You might need to use the function **`deleteMail`**, see section 9.4.5.

10.3 input-address

This component encapsulates (1) a label tag (2) an input text, (3) a button, and (4) a list of mail addresses. This list will only be displayed if the button is clicked. When the list is displayed, if you click over one address of the list, the input text is automatically fulfilled with the selected address and the address list hides. You can set an address either typing it directly in the input text, or by selecting one from the address list.
You will use this component within whatever component that uses an address field, as in the `mail-composer` and `the mail-forwarder` ones.

This custom *input* component will be used as a child component of any other one which will need an input for an address. Once the address would be set this address should be stored in a property of the parent component reactively. As you know, to do this binding we use the v-model directive. Using a v-model on a component is not straight forward. To check how to do it read the section *Using v-model on Components* in the vue tutorial [24].

10.3.1 Component's behavior

When the user clicks on the address button, the list of the mail addresses handled by the mailServer is shown. When the user clicks over one address of the list, the address is displayed in the input field and the list of addresses hides sliding up the rest of the tags. The address button is a toggle button. When you click the button you show the address list. If you click again the button while the list is being displayed you need to hide the list.

10.4 mail-composer

This vue component renders the input fields to compose a mail as shown in the figure 3.
The component template uses an `input-address` component for the *to* field, an `input text` for the *subject* and a `textarea` for the *body*. It also includes the **send button**.

The component might need to store the value of the *input* fields, let's say in an object called `mail`, and export this `mail` object to the `rootComponent`, so that the mail can be sent to the client. See in 13.2 how to take out data back to the parent.

10.4.1 Component's behavior

When the user clicks on the send button the client has to send the mail to the server. You might need to use the function `sendMail` of the `rootComponent`. Once the `send button` is clicked, the current component has to hide and the `mail-list` has to be displayed (figure 2).

10.5 mail-forwarder

This vue component renders a set of fields to forward the selected mail. **These fields are already fulfilled** with the `selectedMail` properties, except the `To` field as shown in the figure 5. The `from` field is read-only, the rest of the fields are editable, so the user can change their content.
The component template uses an `input-address` component for the `to` field, and an input text for the `from`, `subject` and `body` fields. It also includes the send button.

This component might need to receive props.

10.5.1 Component's behavior

When the user clicks on the send button you need to send to the server a json with the mail. Afterwards you need to hide the current component and display the `mail-list` one (figure 2).

10.6 mail-replier

This vue component renders a set of input fields to reply to the selected mail. **These fields are already fulfilled with the selectedMail properties as shown in the figure** 6. The `from` field is read-only, the rest of the fields are editable, so the user can change their content.

The component template includes an input text for the **`from`**, **`to`**, **`subject`** and **`body`** fields. It also includes the send button.

This component might need to receive props.

10.6.1 Component's behavior

When the user clicks on the send button you need to send to the server a json with the mail. Afterwards you need to hide the current component and display the **`mail-list`** one (figure 2).

11 How to deploy and test the project

To deploy the project you just need to go to the inbox directory and execute the command:
`node app.js`

To cancel the deployment you need to type `Ctrl + C`.

To test the application you need a browser and an incognito window to sign in two different users.

12 Sessions scheduling

As this is a big exercise we propose you to deliver each week a bit of functionality. This weekly delivery is optional but if you decide not to go for it you will lose its punctuation from the final exercise mark.

Weekly scheduling

The following table shows a summary of the scheduled tasks per week.

Session	**Week**	**Functionality**
Session 1	May 3rd	Client and server side for the the **`mail-list`** (mails are not clickable). Client and server side for the **`mail-composer`**.
Session 2	May 10th	**`mail-list`**: mails are clickable. Client and server side for the the **`mail-reader`**, **`mail-replier`**, and **`mail-forwarder`**.
Session 3	May 17th	Client and server side for the **`delete`** and **`input-address`**.
Session 4	May 24th	**Exercise delivery through gitHub**. There is no lab class.

Next we describe each session task.

12.1 Session 1: week on the 3rd of May

During this week you will be working on the **`mail-list`** and **`mail-composer`** components.

12.1.1 mail-list

The refresh of the mail list can be done either by clicking on the refresh button or with the automatic polling on the server over the user's inbox. See more information about it in section 9.3.1. Regarding to the server side, you might need to implement the API functions:

- **`GET /`**
- **`GET /inbox`**

Check the server's REST API in section 6.
See section 10.1 for more information about how to develop the client side.
For this delivery it is not necessary the mails in the mail list to be clickable.
Start by developing the server side.

12.1.2 mail-composer

At the moment to implement the **`to`** field use an HTML **`input text`**. Do not use yet the component **`input-address`**. In the server side you will need to implement the REST request **POST /composedMail** (see section 6.5). See section 10.4 for more details of the client side.

To test if both the *refresh* and the *compose* functionalities work, log in one browser as `mar@dom.com` and from an incognito log in as `nil@dom.com`. From `mar's` browser compose a mail to `nil`. Don't press the refresh button in `nil`'s browser, just wait until the new message appears in `nil`'s inbox. From `nil`'s browser compose a mail to `mar`. From `mar`'s browser click on the refresh button and verify that the message sent by `nil` appears in `mar`'s inbox.

This task is due for next week and it is worth **1** points.

12.2 Session 2: week on the 10th of May

First of all you will need to make clickable each mail in the mail list displayed by the **`mail-list`** component. See a tip of how to do it in section 10.1.

Secondly you will need to implement a way to hide a component when another one is displayed. This does not apply to the **`mail-list`** component as it is always visible.

Afterwards you will be working on the **`mail-reader`**, **`mail-replier`**, and the **`mail-forwarder`** components. For the latest two, you have already implemented the server side as you will be using the **`POST /composedMail`** REST request.
For the **`mail-forwarder`** component, at the moment, to implement the **`to`** field use an HTML **`input text`**. Do not use yet the component **`input-address`**.
Test **`mail-replier`** and **`mail-forwarder`** components between two users to check if the messages are sent correctly.

This task is due for next week and it is worth **1** points.

12.3 Session 3: week on the 17th of May

During this week you will be working on the **delete** functionality and on the **`input-address`** component.

12.3.1 delete a mail

To implement this functionality you need to implement the REST request **`DELETE /mail/mailId`** in the server side (see section 6.6). For the client side you just need to send a **`DELETE`** request to the server (see section 9.1).

12.3.2 `input-address`

You need to develop the server side (see section 6.4) and the client side (see section 10.3).
Once you have finished this component you will need to modify the **`mail-composer`** and **`mail-forwarder`** components to implement the **`to`** field using an **`input-address`** component.
Tip: Start by just developing an input text as the template of the component and make sure it works by sending a mail to a user. Once it works add the **`Address Book`** functionality.

With this last delivery the exercise will be finished. So the delivery date for the whole exercise is on the week of the 24th one hour before your lab class. To deliver the exercise you just need to push the final code to the **master** branch in gitHub. After this date you should not modify the repo.

13 Appendix: Some concepts of Vue components

In this section we will mention how the Vue application passes data to the component and gets data from it.

13.1 props

The component is like a mini-application of vue. If you need to access to properties of the vue aplication within the component you need to receive those properties as a custom property (**props**). That is the way the Vue application passes data to the component. See how it works in the vue tutorial **props** section [21]. In the following example we show you a component (**blog-post**) that needs to receive as a parameter a **title**:

```
const app = Vue.createApp({options});
app.component('blog-post', {
props: ['title'],
template: `<h3>{{ title }}</h3>`
})
```

Here is how we will use this component:

```
.....
<html>
<div id="app">
<blog-post title='Why Vue is so fun'></blog-post>
</div>
<html>
.....
```

To access to a **prop** from the component's **data** you need to refer to the prop as an instance property: **this.prop**, for example:

```
const app = Vue.createApp({options});
app.component('blog-post', {
props: ['title'],
data: function(){
return{
heading: this.title,
}
}
}
```

13.2 emits

When a component needs to communicate back up with its parent, it will emit an event. The parent will handle the event with the **v-on** directive. All the events a component can emit should be registered in the **emit** attribute of the component. This attribute is an array of strings. See how it works in [22]. In the following example the **blog-post** component can emit the events *postClicked* and *clear* events.

```
const app = Vue.createApp({options});
app.component('blog-post', {
  props: ['title'],
  emits: ['postClicked', 'clear'],
  data: function(){
   return{
     heading: this.title,
   }
  },
  template:`
    <input ....
    <button v-on:click="$emit('post-clicked',22)>Do something</button>"
    <button v-on:click="$emit('clear')>Clear</button>"
    `
}
```

The components' parent will listen to these events:

```
.....
<html>
<div id="app">
  <blog-post title='My post'
    v-on:post-clicked="myMethod($event)"
    v-on:clear="myClearMethod">
  </blog-post>
 </div>
<html>
.....
```

13.3 v-bind

You use the directive **v-bind** to set a value for an HTML attribute **bound** to a property or a method or a computed property of the vue application. This is a *one-way* binding. If the value of the vue application property changes it will change the value of the **v-bind** attribute. Not the other way round. For example:

```
<a v-bind:href="url">
....
const app = Vue.createApp({
data(){
url: "http://dom.com".
}
});
```

Also, the dynamic value of the **v-bind** attribute can come from a javascript expression:

```
<a v-bind:href="dom.com/file" + counter + ".html">
//The following expression is equivalent to the first one
<a v-bind:href=`dom.com/file${counter}.html`>
```

See how the directive **v-bind** works at [20].

13.4 Combining props and v-bind

Imangine that you need to pass to a component a property. The value of this property can be static (an specific value) or dynamic (somethig dynamically generated). The **blog-post** example in the 13.1 section is an example of passing an static property to a component.
In the following example we pass a dynamic property to a component. Anything that has to do with dynamic properties is handled by **v-bind**. In the following example the title of the **blog-post** component comes from a list.
The following code defines the **blog-post** component that receives the **title** prop.

```
const app = Vue.createApp({options})
app.component('blog-post', {
props: ['title'],
data: function(){
return{
heading: this.title,
}
}
}
```

This is the vue instance that contains the list of titles in the data property **posts**.

```
const app = Vue.createApp({
posts: ['Learning with Vue', 'I love Vue', 'Why Vue is so fun' ]
})
```

In here we generate as many tags **blog-post** as titles we have in the list **data.posts**.

```
<blog-post v-for="post in posts" v-bind:title="post"></blog-post>
```

As the prop **title** has a dynamic value, it is set by the **v-bind** directive. See how the **v-for** directive works in the section [23] in the vue tutorial.

References

[1] **NodeJS website** URL: `https://nodejs.org/en/`

[2] **npm website** URL: `https://www.npmjs.com/`

[3] **Express framework.** URL: `https://expressjs.com/`

[4] **Express basic routing.** URL: `https://expressjs.com/en/starter/basic-routing.html`

[5] **Express extended routing.** URL: `https://expressjs.com/en/guide/routing.html`

[6] **Object application** URL: `https://expressjs.com/en/5x/api.html#app`

[7] **Function app.get** URL: `https://expressjs.com/en/5x/api.html#app.get.method`

[8] **Function app.post** URL: `https://expressjs.com/en/5x/api.html#app.post.method`

[9] **Function app.delete** URL: `https://expressjs.com/en/5x/api.html#app.delete.method`

[10] **Object Request.query** URL: `https://expressjs.com/en/5x/api.html#req.query`

[11] **request.path property** URL: `https://expressjs.com/en/5x/api.html#req.path`

[12] **Function response.sendFile** URL: `https://expressjs.com/en/4x/api.html#res.sendFile`

[13] **Function response.redirect** URL: `https://expressjs.com/en/5x/api.html#res.redirect`

[14] **Function response.json** URL: `https://expressjs.com/en/4x/api.html#res.json`

[15] **Function response.status** URL: `https://expressjs.com/en/5x/api.html#res.status`

[16] **express-session middleware** URL: `https://www.npmjs.com/package/express-session`

[17] **How to write a middleware** URL: `https://expressjs.com/en/guide/writing-middleware.html`

[18] **Client fetch** URL:
`https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API/Using_Fetch`

[19] **setInterval function** URL: `https://javascript.info/settimeout-setinterval`

[20] **Binding a vue property to a tag attribute** URLs:
`https://v3.vuejs.org/guide/template-syntax.html#attributes`
`https://v3.vuejs.org/api/directives.html#v-bind`

[21] **Passing data to a child component** URL:
`https://v3.vuejs.org/guide/component-basics.html#passing-data-to-child-components-with-props`

[22] **Communication between a component and its parent** URL:
`https://v3.vuejs.org/guide/component-basics.html#listening-to-child-components-events`

[23] **List rendering** URL:
`https://v3.vuejs.org/guide/list.html#mapping-an-array-to-elements-with-v-for`

[24] **Using v-model on Components** URL:
`https://v3.vuejs.org/guide/component-basics.html#using-v-model-on-components`

[25] **Event handling** URL: `https://v3.vuejs.org/guide/events.html#listening-to-events`