PIMS: FE

The Patient Information Management System: Flask Edition

PIMS-Demo.HostedBy.JamesDev.co.gg

Project Development & Testing

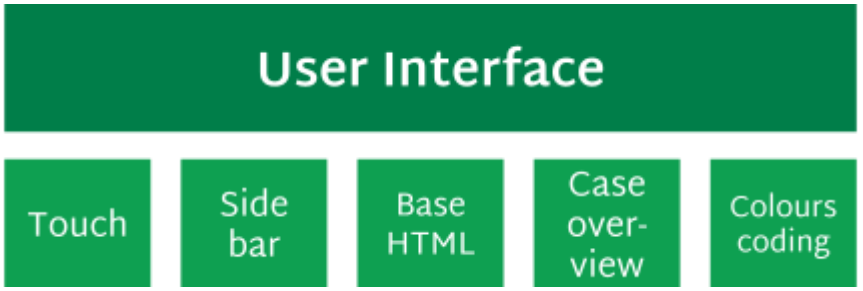
Contents:

Overall design	2
User interface	4
Authentication	20
Patient data	30
Export case	43
Triage	54
Admin	66
Utilities	83

Overall design

The project can be split into seven main parts: UI, authentication, patient data, export, triage, admin functions, and system utilities.

User interface



Touch

Justification

The software is designed for comfortable use with a mouse and keyboard or on a touchscreen device, such as a tablet or phone. This was an important feature the software needed - having that versatility allows the system to be portable while maintaining robust data collection and retrieval abilities. In the setting of a medical event, especially a busy triage event, having the software be functional on a small, portable device is essential for system utilisation. As such, the interface of PIMSFE features large clear buttons, with colour coding to show what is clickable and what is not.

The usability of the features are very simple by design. The buttons are designed to be prominent, easy to identify, and touch friendly. The buttons are `div` elements, clickable through the `onclick` attribute and a simple line of Javascript. By using `divs` over other elements, I am better able to customise their shape and appearance, making them larger, increasing usability on touchscreen devices and improving aesthetics on desktops.

Design & code

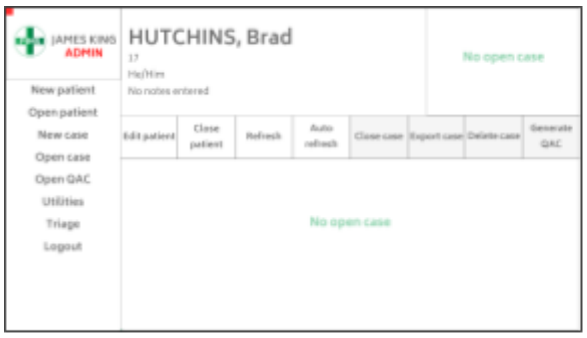


Fig 1: The home screen with no selected case

Fig 2: The home screen displaying a case

Figures one and two show the design on the main 'home' screen. Large touch buttons can be seen in the side menu, linking to other pages, and others can be seen across the central strip. These buttons link into commonly used functions.

```
<div id="homebar-item" class="homebar-item-active"
onclick="window.location.href = '/editpatient'">
    <h5>Edit<br>patient</h5>
</div>
```

```
#homebar-item, #auto-refresh {
    height: 100%;
    width: calc(100% / 9.5);
    display: inline-block;
}
```

Variables

1. **Button value**

   This is the name displayed on the button, it is set directly in the HTML.

2. **Button link or function**

   This is the link or Javascript function triggered when the button is clicked, this is set directly in the HTML.

Test plan

Test	Method	Expected output
Button rendering	The page will be loaded on different devices of varying size, and in varying browsers. The buttons should render in the correct order and size.	All the buttons in a section should either run across one row or down one column.
Button action	The page will be loaded and each button shall be pressed.	All the buttons should link to their associated page, or trigger their associated javascript function.

Test log

Test no.	Date	Pass / Fail	Outcome	Alterations
1	20 June 2023	Fail	Button rendered into their correct position on the display, but the hover effect which would change the colour of the text when the button is moused over only worked when over the text.	Alterations were made to the CSS code for homebar-item-active DIVs to cascade hover over events to children.
2	20 June 2023	Pass	Button rendered in correctly, hover event cascaded properly.	N/A

3	20 June 2023	**Pass**	When clicked, the javascript 'onclick' function activated redirecting to the destination page.	N/A

Sidebar

Justification

The sidebar provides quick access to the core pages and functions of the software, such as opening or creating a user and case. The sidebar also contains brief information about the user session, such as the name of the user account that has been used to login and the privilege level; a logout button can also be found at the bottom to allow the user a secure way to end the session.

As the sidebar is a navigation element, it is designed to be easy to use and intuitive. The sidebar will be displayed on all pages other than authentication pages (e.g., login), allowing quick access and navigation at all times. In addition, the CSS for the website will be coded to allow only the content of the page to scroll if they do not fit on the user's screen (`overflow-y: scroll;`). The sidebar, however, will remain fixed at all times (`overflow-y: hidden;`) - this will provide a reference point for the user to know where they are on the page, and will allow them to quickly navigate away at any time.

Design & code

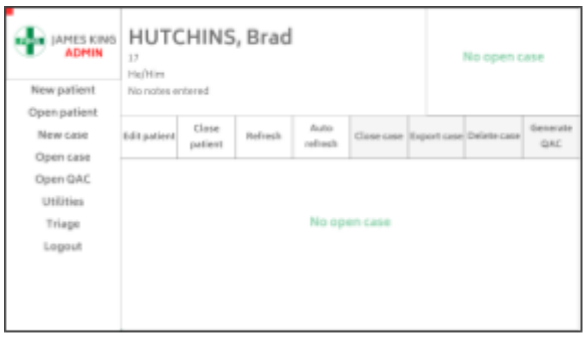


Fig 1: The home screen with no selected case

Fig 2: The home screen displaying a case

Figures one and two show the design on the main 'home' screen. The sidebar can be seen on the left hand side of the screen, with a list of large touch buttons and a small information panel at the top.

```
<div id="sidebar">
            <div id="sidebarlogocontainer">
                <div id="sidebarlogobundle">
                    <img id="sidebarlogo" src="/static/assets/logo.png"
onclick="window.location.href = &quot;/&quot;">
                    <div id="sidebarlogotextbundle">
                        <h3>JAMES KING</h3><br>

                         <h3 style="color: #FF0000;">ADMIN</h3>

                     </div>
                </div>
            </div>
            <div id="actionbuttons">
                <div class="actionbutton" onclick="window.location.href =
&quot;/newpatient&quot;">
                    <h3 class="actionbuttontext">New patient</h3>
                </div>
                <div class="actionbutton" onclick="window.location.href =
&quot;/openpatient&quot;">
                    <h3 class="actionbuttontext">Open patient</h3>
                </div>
                <div class="actionbutton" onclick="window.location.href =
&quot;/newcase&quot;">
                    <h3 class="actionbuttontext">New case</h3>
                </div>
                <div class="actionbutton" onclick="window.location.href =
&quot;/opencase&quot;">
                    <h3 class="actionbuttontext">Open case</h3>
                </div>
                <div class="actionbutton" onclick="window.location.href =
&quot;/openQAC&quot;">
                    <h3 class="actionbuttontext">Open QAC</h3>
                </div>
```

```
                <div class="actionbutton" onclick="window.location.href =
&quot;/utilities&quot;">
                    <h3 class="actionbuttontext">Utilities</h3>
                </div>
                <div class="actionbutton" onclick="window.location.href =
&quot;/triage&quot;">
                    <h3 class="actionbuttontext">Triage</h3>
                </div>
                <div class="actionbutton" onclick="window.location.href =
&quot;/logout&quot;">
                    <h3 class="actionbuttontext">Logout</h3>
                </div>
            </div>
        </div>
```

Variables

1. **Button innerHTML**

   This is the name displayed on the button, it is set directly in the HTML, and indicates what the button will do.

2. **Button action**

   For the sidebar, a small piece of Javascript code (`window.location.href='link goes here';`) is used for each button to redirect the user to the correct page.

Test plan

Test	Method	Expected output
Button rendering	The page will be loaded on different devices of varying size, and in varying	All the buttons in a section should either run

Base HTML

Justification

In order to keep a consistent design language across the website, and to improve efficiency, the site will utilise shared HTML bases. These files will set out the basic layout and structure of the page, including the meta data and sidebar, with other files inheriting from the bases to form whole webpages. This reduces repetition, reduces memory usage, improves consistency, and makes editing the source code easier.

The usability of the features are very simple by design. The buttons are designed to be prominent, easy to identify, and touch friendly. The buttons are `div` elements, clickable through the `onclick` attribute and a simple line of Javascript. By using `divs` over other elements, I am better able to customise their shape and appearance, making them larger, increasing usability on touchscreen devices and improving aesthetics on desktops.

Design & code

Fig 1: The home screen with a selected case. The HTML sources are indicated

Figure 1 shows the PIMS homescreen, divided into two sections. The left section, containing the sidebar, shows the rendered HTML from the `base.html` file. While

the right section shows the rendered HTML from the `home.html` file, which inherits from the base, forming a whole page.
For the final site, it is likely that there will be multiple base files for different sections of the website.

Variables

1. **Button value**
   This is the name displayed on the button, it is set directly in the HTML.
2. **Button link or function**
   This is the link or Javascript function triggered when the button is clicked, this is set directly in the HTML.

Test plan

Test	Method	Expected output
HTML rendering	The page will be loaded on different devices of varying size, and in varying browsers.	The different HTML templates should load in the correct order with the correct contents.

Test log

Test no.	Date	Pass / Fail	Outcome	Alterations
1	23 June 2023	Fail	With some pages, there were some initial problems with padding for the insert div. For some pages, the standard	This was rectified by removing the padding from the base template itself and

			padding caused issues with the visual layout of the page.	instead to the page inheriting from the template.
2	23 June 2023	**Pass**	The base template loaded successfully, and appeared as designed.	N/A

Project review:

The base design is coded to be flexible, allowing it to work on all common screen sizes. However, on some mobile devices the layout can look somewhat squashed - especially with the sidebar. A simple change would be to create a mobile-base and mobile-home alternative files, which would be optimised for smaller screens. Little further alterations would be needed, with all other pages inheriting from the base or having designs that would not need to be altered as drastically as the home page.

Patient & case overview

Justification

In a first aid scenario time is of the essence. For this reason, PIMS must be easy to navigate and must provide the most important, relevant information in a prominent position. One of the most crucial pieces of information stored regarding a case in the system is the last recording, or 'note'. This note contains the patient's most recent pulse, blood oxygen saturation, and observations.

The location was chosen to be prominent, with most users naturally first looking at the top of the page before moving down. In addition, the contents are designed to be eye catching, with bold fonts and bright colours.

Design & code

Fig 1: The home screen with a selected case. The patient overview is outlined in blue, and the case overview in red.

Figure 1 shows the PIMS home screen with a loaded case. The patient information is shown in a large panel at the top of the page (outlined in blue), while the case overview is displayed to the right (outlined in red). The patient overview includes core information regarding the patient, while the case shows the latest recorded statistics and a triage colour band.

Variables

1. **Patient object**

   Contains the patient name, age, pronouns, and a small note. The primary key of the patient serves as the foreign key for all associated case and note objects.

2. **Case object**

   Contains information about relevant case information, and serves as a reference point for all notes created for case. Includes data on when the patient entered and left care, the chief complaint, and who they were discharged by.

3. **Note object**

   Each note serves as an individual snapshot during a case, including visual observation data as well as pulse oximetry readings.

Test plan

Test	Method	Expected output
Overview rendering	The page will be loaded on different devices of varying size, and in varying browsers.	The overview should be clearly displayed at its location on the page, including all of the relevant patient and case data.
Colour bar colour coding	The page will be loaded with multiple cases, with a variety of readings in different triage categories.	When a case is loaded, the colour bar at the bottom of the case overview section should change colour based on pulse-oximetry readings.

Test log

Test no.	Date	Pass / Fail	Outcome	Alterations
1	23 June 2023	**Pass**	The overview bar loaded in, with a static placeholder being used to check spacing and layout.	The static placeholders were replaced with the Jinja variables.

Colour coding

Justification

Colour coding is a unique way of conveying important information clearly and quickly, while also creating a unique product aesthetic. PIMSFE utilises two main forms of colour coding; pulse oximetry (pulseox) and triage colour coding. Pulseox colour coding is used to clearly distinguish which readings relate to a patient's pulse rate (light blue) or blood oxygen saturation (red). The colours were chosen to match the standard colour schemes used on many pulse oximeter devices, helping to create a more intuitive experience. Triage colour coding links into PIMS' automatic triage method, which categorises patients into one of four categories (green, yellow, orange, red) based on their readings.

The location of each colour coded element was chosen with much care and thought. On each note bar, the triage colour coding is displayed on the left hand edge making it the first thing a person would see when scanning the line. This also allows the user to quickly scan down the page to see how the patient's condition has changed over the length of the case. This is also the case for the pulseox data, which is located at the right extreme of the note - being isolated from other data allows the figures to stand out, and for the operator to be able to scan up and down the screen. The case overview triage bar was actually a late addition to the design. Being near the top of the screen, it helps to bring attention to the patient's current condition quickly without the user having to skip down to the latest recorded note.

Design & code

Fig 1: The home screen with a selected case. PulseOx colour coding is outlined in red, triage colour coding in blue.

Figure 1 shows the PIMS home screen with a loaded case. Pulseox colour coding can be seen in the case overview panel as well as next to each note. Triage colour coding is also used extensively on this page, with each note being automatically interpreted, and given a colour bar on their left border. Additionally, the triage colour of the last recorded note can also be seen on the bottom border of the case information panel.

Variables

1. **Pulse rate**

   An integer supplied by the note object that represents the pulse of a patient, $0 \leq pulse \leq 220$.

2. **Blood oxygen saturation**

   An integer supplied by the note object that represents the percentage saturation of oxygen in the blood, $0 \leq spo_2 \leq 100$.

Test plan

Test	Method	Expected output
Overview rendering	The page will be loaded on different devices of varying size, and in varying browsers.	Colour coded elements should be displayed at the correct size in the

		correct location on each page.
Triage colour bars	The page will be loaded with multiple cases, with a variety of readings in different triage categories.	When a case is loaded, the triage colour bars should change colour based on pulse-oximetry readings.

Test log

Test no.	Date	Pass / Fail	Outcome	Alterations
1	23 June 2023	Pass	The pulseox colour coding was correctly applied, following the colour palette used by the pulse oximetry device.	Minor padding changes.
2	23 June 2023	Fail	The colour bar update was not triggered when the page loaded, as such, the bar was not displayed.	The function code was removed from the onload on the div, and was placed inline within script tags.
3	23 June 2023	Pass	The colour bar updated when the page was loaded, displaying an accurate	N/A

			classification.	

Authentication

Login control

Justification

With any system that handles sensitive user data - let alone medical data- it is important that access is thoroughly secured. As such, PIMSFE can only be accessed with an authenticated user account.

Logging in should always be the simplest step when using a system. PIMS takes an integer as a username, which is auto-assigned when the account is created, and a

password. The password is stored as a secure hash in the database, which prevents passwords from being leaked if a data breach were to occur. To further boost the security of stored passwords, PIMS uses a salt & hash algorithm, using the username to further obscure the data being hashed.

Design & code

```
def pwdHSH(username:int, password:str):
    return \
sha256(sha256(f"{username}{password}{username}".encode('utf-8')).hexdig
est().encode('utf-8')).hexdigest()
```

```
        username = int(request.form.get("username")) or None
        password = request.form.get("password") or None
        next = request.form.get("next") or None

        if None in [username, password]:
            flash("Form not complete")
            return render_template('login.html', next=next)

        user = OPS.query.get(username) or None
        if user == None:
            flash("User was not found")
            return render_template('login.html', next=next)
        if pwdHSH(username, password) == user.pwdhsh:
            [...]
            if next is None:

                return redirect(url_for('main.index'))
            else:

                return redirect(next)
        else:
            flash("Incorrect password")
            return render_template('login.html', next=next)
```

Variables

1. **Username**

   Supplied as an integer from a HTML form via POST. This is unique to each user in the system, and is generated when the account is created.

2. **Password**

   Supplied as a string from a HTML form via POST. This is used to authenticate the user, permitting them access to the system.

3. **Next**

   If the user was trying to access a specific page before they were asked to login, a GET argument with the page address will be included with the authentication request. This allows the system to redirect the user to their intended page once their account has been authenticated.

Test plan

Test	Method	Expected output
Invalid user	The algorithm will be checked by supplying the login system with the details of a non-existent account.	Access should be denied, with an error message displayed detailing that the account could not be found.
Valid user, invalid password	The algorithm will be checked by supplying the system with the account number of a user on the system, but with the incorrect password.	Access should be denied, with an error message displayed detailing that the password did not match what was stored in the system.
Valid	The algorithm will be checked by	Access should be granted

password and user	supplying the system with the account number of a user on the system with the correct password.	with the user directed to the home page or to the page they were trying to access before logging in.

Test log

Test no.	Date	Pass / Fail	Outcome	Alterations
1	28 June 2023	Fail	When an invalid user ID was submitted the program would not recognise this and would not abort the authentication function. As such, an internal server error would occur.	A catch 'if' statement was added that would check if the returned user object was a nonetype. If none was returned, it can be assumed the user could not be found and the function would abort and return the login template.
2	28 June 2023	Fail	When the username and/or password box were left empty and submitted they would be supplemented with a none object. This would cause issues during the	A check if statement using in list was added, if either values were none the function would abort and return the login

			authentication function as the program tried to call attributes that did not exist.	template.
3	28 June 2023	**Pass**	Valid users were correctly authenticated and logged in.	N/A

Privileges

Justification

To prevent the misuse of the system by authenticated users, PIMS utilises a ten-step permission scale, with ten being an admin account, and most users being at level five. PIMS uses this when checking to see if a user is permitted access to a certain action or section of the system, such as deleting a patient or editing users. The permission checking happens completely in the background, making it a worry-free experience for the users. If the user has the adequate permissions to access a part of the website or complete a task, the page will be loaded as usual. If the user does not have the required permissions, however, a simple error message will be displayed showing that they do not have access and they will be unable to proceed any further.

Design & code

```
{% if current_user.privs == 10 %}
<h3 style="color: #FF0000;">ADMIN</h3>
{% elif current_user.privs >= 5 %}
```

```
<h3 style="color: #4467a7;">USER</h3>
{% elif current_user.privs >= 1 %}
<h3 style="color: yellow;">EMA</h3>
{% endif %}
```

```
if current_user.privs < 10:
	return render_template('error.html', errormessage='403 Forbidden', errortext='This action is only accessible to admin accounts.')
else:
	[...]
```

Variables

1. **User privileges**

   An attribute of the operator object, it is an integer in the range $0 \leq privs \leq 10$.

Test plan

Test	Method	Expected output
Invalid user privileges	An user-only page will be loaded with an account with regular or admin user privileges ($p = 5$).	Access should be denied, with an error message displayed detailing that the account does not have sufficient permissions.
Valid user privileges	An admin-only page will be loaded with an account with admin user privileges ($p = 10$).	Access should be allowed, with the user sent to the requested page without

		issue.

Test log

Test no.	Date	Pass / Fail	Outcome	Alterations
1	29 June 2023	Pass	On admin only pages, users with privilege values of less than ten are refused entry, and are redirected to an error 403 page.	N/A

Project review:

At current, emergency medical access users are not implemented into PIMS:FE - users with a privilege value less than ten but greater than one will have the same access on the system. In future development, this is something that will need to be limited - however EMA users are rarely ever used.

Account arguments

Justification

Account arguments act as flags in the system that alters the login flow for a user. After a successful authentication, the system checks to see if there are any flags for

that account, then acts accordingly. There are two possible account flags: reset password, and disable account.

The process occurs completely in the background without requiring any extra input from the user, other than when an argument is selected when editing an account.

The output is clear, if a flag is present on the account, the user is immediately prompted on their next steps; either changing their password or contacting their admin. If no flags exist on the account (`accargs = 0`).

Design & code

```
match user.accargs:
                case 1:
                    return redirect(url_for('auth.changePassword',
username=username, next=next))
                case 2:
                    flash("Your account has been locked by an
administrator. " \
                          "Please contact your system manager for more
details.")
                    return render_template('login.html', next=next)
```

Variables

1. **Account arguments**

   An attribute of the operator object, it is an integer with a value of either 0, 1, or 2.

Test plan

Test	Method	Expected output
No account flags	A user account will be logged into with the correct credentials. This account will	Access should be permitted, with the user

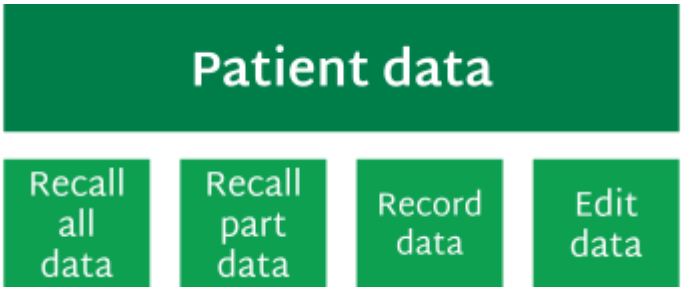
accargs = 0	not have any flags placed on it.	directed to the user page or to their redirect page. No further action should be taken.
Change password flag accargs = 1	A user account will be logged into with the correct credentials. This account will have the change password flag on it.	Access should be permitted, with the user redirected to the change password page. At this point, the user will not be logged in, and as such will need to re-enter their current password to proceed with the change and to login. This is done to prevent users from skipping a password change.
Account disabled flag accargs = 2	A user account will be logged into with the correct credentials. This account will have the account disabled flag on it.	Access should be denied, with an error message displayed that their account has been locked, and that they will need to contact their admin.

Test log

Test	Date	Pass / Fail	Outcome	Alterations

no.				
1	30 June 2023	**Pass**	accargs = 0 > the match statement was not triggered and the user logged in as normal.	N/A
2	30 June 2023	**Pass**	accargs = 1 > the match statement was triggered and the user was not logged in, but was redirected to the change password page.	N/A
3	30 June 2023	**Pass**	accargs = 2 > the match statement was triggered and the user denied entry to the system.	N/A

Patient data



Recall all data

Justification

PIMS, at its core, is a data storage and retrieval system. As such, one of the fundamental requirements for the system is to be able to retrieve all the stored data for a particular query.

Data retrievals are completed in the background, out of sight of the users. PIMS utilises cookies to store the IDs of the patient and case the user is currently accessing, this makes re-fetching the data when the user switches pages easy and hassle free - essentially allowing the system to remember what (or who) you were working on. From the point of view of the developer, data retrieval is made simple through the Flask-SQLalchemy module. This simplifies SQL statements into Python-style methods, and converts the records in the database into objects allowing for more intuitive handling of data.

Design & code

```
@main.route('/', methods=["GET"])
@login_required
def index():
    patient = request.cookies.get('patientID') or None
    caseObj = request.cookies.get('caseID') or None
```

```
    notes = []
    if patient is not None:
        patient = Patients.query.get(int(patient)) # patient object
retrieved
        if patient is None:
            resp = make_response(render_template('home.html',
current_user=current_user, patient=patient, case=caseObj, notes=notes))
            resp.delete_cookie('patientID')
            return resp
    if caseObj is not None:
        if patient is None:
            caseObj = None
        else:
            caseObj = Cases.query.get(int(caseObj)) # case object
retrieved
            if caseObj is None:
                pass
            elif caseObj.patientno != patient.patientid:
                caseObj = None
            else:
                notes = Notes.query.filter_by(caseno =
caseObj.caseno).all() # all notes for the case are retrieved
    return render_template('home.html', current_user=current_user,
patient=patient, case=caseObj, notes=notes)
```

Variables

1. **Object ID**

   Object IDs are the primary keys of the records stored in the PIMS database and are used to access particular data. The type will vary depending on what type of data the user is trying to access, be that patient, case, note, etc.

Test plan

Test	Method	Expected output

Retrieval of specific record	Sections of code that include data retrieval, especially where filters are used, will be tested with multiple inputs to check performance.	The expected object(s) should be returned.

Test log

Test no.	Date	Pass / Fail	Outcome	Alterations
1	1 July 2023	Fail	Improperly closing cases or cookie timeouts can from time to time result in there being a case cookie without a patient cookie. This is a situation that was not expected, causing issues with function and rendering when the home page loaded.	An additional check statement was added into the case data recall section - if there is no open patient (indicating there was not a patient cookie), the case cookie will be deleted for security.
2	1 July 2023	Pass	When a case cookie is present but not a patient cookie the system removes the case cookie and refuses to load the case data.	N/A
3	1 July	Pass	The correct data was recalled from the supplied ID. The	N/A

	2023		recalled data included all fields in the record.	

Partial data recall

Justification

A partial data recall is where only part of a record is retrieved from the database. This is helpful for system security, by only recalling the parts of the record needed it removes the possibility for a bad actor to access the full object and its data.
The usability is very similar to that of the full data recall, with the process appealing to be no different to the end user.

Design & code

```
@main.route('/utilities/operators', methods=['GET'])
@login_required
def operators():
    if current_user.privs < 10:
        return render_template('error.html', errormessage='403 Forbidden',
errortext='This action is only accessible to admin accounts.')
    allOps = OPS.query.with_entities(OPS.OPID, OPS.fname, OPS.lname).all()
    return render_template('operators.html', ops=allOps)
```

Variables

1. **Object ID**

   Object IDs are the primary keys of the records stored in the PIMS database and are used to access particular data. The type will vary depending on what type of data the user is trying to access, be that patient, case, note, etc.

Test plan

Test	Method	Expected output
Retrieval of specific record	Sections of code that include data retrieval, especially where filters are used, will be tested with multiple inputs to check performance.	The expected object(s) should be returned.
Retrieval of specific fields in a record	Sections of code that use .with_entries() to only retrieve specific fields from a record will be tested with multiple inputs to make sure only the correct fields are retrieved.	The expected object(s) should be returned with only the expected fields.

Test log

Test no.	Date	Pass / Fail	Outcome	Alterations
1	1 July 2023	Fail	The section of code exemplified in this document was copied from another area of PIMS. When copying it into its new area, the classes within the with_entities() function were not changed, as a result the SQL query was trying to pull fields from another object, causing an	The class declarations were changed to OPS, resolving the error.

			internal server error.	
2	1 July 2023	**Pass**	The correct data was recalled from the supplied ID. The recalled data includes only that specified within the with_entities() function.	N/A

Recording data

Justification

As a data storage system, being able to record data is one of the most vital functions of the software - without it, it is nothing but a glorified database browser. Storing data is made easy with Flask-SQLalchemy, which allows SQLite records to be interacted with as Python objects. As such, in PIMS, creating a new record is as easy as creating a new object.

Adding data is designed to be simple and intuitive. The input fields in the UI are distinguished from the rest of the website, and are clearly marked to show what data goes where. The HTML will have built in data checking to make sure all of the required fields are entered, with the data checking in the Python code serving as a backup in case the HTML is tampered with. Should the system need to refuse an entry, the user will be shown a clear error message instructing them on what they will need to correct to proceed. If all the data entered is correct the user will be notified through means of a redirect to the next step in the program, usually the home page.

Design & code

```
@main.route('/newcase', methods=["POST", "GET"])
@login_required
def newCase():
    if request.method == "GET":
        [...]
    if request.method == "POST":
        patient = request.cookies.get("patientID") or None
        patientID = request.form.get('patientid') or None
        if str(patient) != str(patientID):
            return render_template("error.html", errormessage="Forbidden",
errortext='The passed patient ID does not match patient ID in
cookies.',redirectURL='/newcase')
        patient = Patients.query.get(int(patient)) or None
        if patient is None:
            return render_template('error.html', errormessage='404 Patient not found',
errortext='The patient could not be located in the database, please reopen patient.')
        timeAdmitted = request.form.get('admittime') or None
        if timeAdmitted is None:
            return render_template('error.html', errormessage='Incomplete form',
errortext='Please complete all entry fields marked as required.',
redirectURL='/newcase')
        nC = Cases()
        nC.patientno = patient.patientid
        nC.timeadmitted = timeAdmitted
        nC.admittedfor = request.form.get('admitreason') or ''
        db.session.add(nC)
        db.session.commit()
        resp = make_response(redirect(url_for('main.index')))
        resp.set_cookie('caseID', str(nC.caseno))
        return resp
```

Fig 1: A sample "new case" window.

Variables

1. **Data entry fields**

   Depending on the type of data being entered, the number and types of data entry fields - HTML `<input>` elements will vary. For some pages, such as adding a case, this could be as few as two fields. While some others may need more than five, such as when adding a note.

Test plan

Test	Method	Expected output
Creation of record	Correct data will be entered into a data entry field.	The record is created without errors, but has not necessarily been committed to the database yet.
Insufficient data is rejected	Incorrect data will be entered into a data entry field.	The input is rejected, and an error message is displayed informing the user that more data is required.
Record is committed	Correct data will be entered into a data entry field. Then the page will be reloaded and the record viewed.	The record is created without errors, and can be recalled through the website after a reload.

Test log

Test no.	Date	Pass / Fail	Outcome	Alterations
1	3 July 2023	Fail	For text fields where an input was optional and not supplied, the text field would be filled with “None” - which is not ideal.	When the field is assigned, an or operator is used. This compares the input against an empty string. In Python, None equals False, and a string of any length True. As such, if the retrieved value is None, the system will instead use the blank string.
2	3 July 2023	Fail	The data was correctly submitted but the record was not stored in the database.	The line db.session.commit() was appended after the object was added to the database.
3	3 July 2023	Pass	When required values were not supplied from the HTML form, the function is aborted back to the template with a flash message instructing the user about the issue.	N/A

Project review:

Currently, PIMS does not specify what fields are required or missing when a user submits a form - a blanket “missing field” message is displayed. While this does work, its lack of specificity does not aid usability and might cause frustration when completing the form.

Editing data

Justification

The last major data procedure needed for PIMS is the ability to edit records. By design, some records within PIMS cannot be edited, to preserve their authenticity. The edit screens are designed to be highly usable; they use the same layout and design as data entry screens, however the data input boxes are already prefilled with the existing information stored on the system. The accepting Python code will then check that the data entered is valid before updating the record and redirecting the user. If the data is not valid, the code will return the user back to the edit screen with a message describing what they need to correct.

Design & code

```
@main.route('/editpatient', methods=['GET', 'POST'])
@login_required
def editPaitent():
   if request.method == 'GET': # the GET method recalls patient data from the database
and serves the HTML template
       patient = request.cookies.get("patientID") or None
       if patient is None:
           return redirect(url_for('main.index'))
       patient = Patients.query.get(int(patient)) or None
       if patient is None:
           return redirect(url_for('main.index'))
       return render_template('createeditpatient.html', endpoint='/editpatient',
patient=patient, new=False)
   elif request.method == "POST": # the POST method receives the edited data back from
```

```
the HTML page then adds them to the database
        patient = request.cookies.get("patientID") or None
        patientID = request.form.get("patientid") or None
        if (None in [patient,patientID]) or (patient != patientID):
            return render_template("error.html", errormessage="Forbidden",
errortext='The passed patient ID does not match patient ID in
cookies.',redirectURL='/editpatient')
        patient = Patients.query.get(int(patient)) or None
        if patient is None:
            return render_template("error.html", errormessage="404 Patient Not Found",
errortext="The patient could not be located within the database.")
        patient.name = request.form.get('name')
        patient.age = int(request.form.get('age'))
        patient.gender = request.form.get('pronouns')
        patient.notes = request.form.get('notes')
        db.session.commit()
        flash('Patient edits have been saved')
        return redirect(url_for('main.index'))
```

Fig 1: A sample "edit patient" window.

Variables

1. **Data entry fields**

   Depending on the type of data being entered, the number and types of data entry fields - HTML `<input>` elements will vary. For some pages, such as adding a case, this could be as few as two fields. While some others may need more than five, such as when adding a note.

Test plan

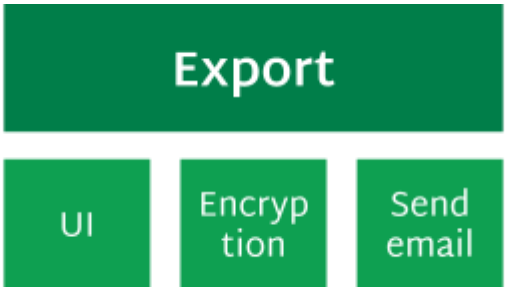
Test	Method	Expected output
Recalling the correct record	The edit form for a record will be completed and submitted.	The edits are applied to the specified record.
Incorrect data is rejected	Incorrect data will be entered into a data edit form.	The input is rejected, and an error message is displayed informing the user that more data is required.
Edits are committed	Correct data will be entered into a data entry field and submitted. Then the page will be reloaded and the record viewed.	The specific record is edited without errors, and can be recalled through the website after a reload.

Test log

Test no.	Date	Pass / Fail	Outcome	Alterations
1	4 July 2023	**Pass**	The correct data is recalled from the database, and the edits applied as instructed by the user.	N/A
2	4 July	**Pass**	The edits are committed to the database, this can be	N/A

	2023		checked by then recalling the data again to the edit page or another data recall screen.	

Export case



Export user interface

Justification

To facilitate exporting a case, a UI window is required. Two inputs are required: the email address to which the exported case will be sent, and the patient's date of birth or any other six digit number, used to encrypt the file. Having a window for exporting the case also gives the user one final chance to check that they have selected the correct case and patient, helping to protect the patient's data.
The confirmation screen is designed to be minimal yet information rich. The top bar of text contains the date and time the patient was admitted, used to identify the case, and the name of the patient. Beneath are two clearly labelled input boxes, one for the email and another for the date of birth or PIN. These boxes will utilise HTMLs built in input validation systems to make sure that the contents are valid before ever reaching the Python code.

Design & code

Fig 1: A sample "export" window.

Variables

1. **Patient email**

   The address the exported case pdf will be sent to.

2. **Patient date of birth**

   The patient's date of birth is used as a pin code to encrypt the exported case pdf file. This helps to protect the contents of the file during transit across the internet and on the patient's device.

3. **Patient ID**

   Used to recall patient data from the database.

4. **Case ID**

   Used to recall case data from the database.

Test plan

Test	Method	Expected output
Page rendering	A PIMS case will be loaded and the export page opened.	The page should render according to the design, the text should be in the correct location, correct

		size, and be easily readable.
Data validation	Incorrect values will be entered into the two export input boxes. For instance, a malformed email address in the email box, or text within the DOB field.	The input should be rejected. Depending on the user's browser, a small popup may appear indicating the specific nature of the issue.
Action	With correctly inputted data, the export page will be submitted.	The data should be sent using the HTML POST method to '`/exportcase`'

Test log

Test no.	Date	Pass / Fail	Outcome	Alterations
1	5 July 2023	**Pass**	The page rendered as expected according to the design.	N/A
2	5 July 2023	**Pass**	When the form was submitted, the data was correctly checked to ensure that both fields were filled. When submitted, the post request was forwarded to the	N/A

			correct route.	

Data recall and encryption

Justification

The PIMS export function's purpose is to draw upon a large amount of data stored in the database and present it in a way that is logical and easy to understand, even for those who may have no medical knowledge. As such, several database calls have to be made in order to collect all of the required data. Given the contents of the data, it must also be encrypted to secure the data. For this, the patient's date of birth is used as a six digit pincode.

The encryption process happens entirely in the background of the software. The program first uses a HTML template to construct a temporary HTML file with all of the patient's data in memory. This HTML file is then converted to a plain-text PDF file which is stored in the server's directory. This file is then opened, read, and encrypted before being resaved ready to be sent, the unencrypted copy is deleted for security and storage capacity saving.

Design & code

```
    htmlfile = open("/var/www/PIMSFE/templates/email-2023.html", "r")
    htmltop = htmlfile.read()
    htmlfile.close()
    dtstore = dt.now()
    #... HTML formatting logic goes here, placeholders within the template (e.g.,
{QAC}) are replaced with valued taken from objects
    htmltop, htmlbottom = htmltop.split('<br id="python-break">')
    #... SMTP client creation logic
    for note in notes:
```

```
        for user in users:
            if user.OPID == note.loggedby:
                loggedby = [user.fname, user.lname]
            else:
                pass
        htmltop+=f"""<tr>
<td style="border-left: 3px solid {triage(note.PR, note.SPO2)}; padding-left:
5px;">{note.noteno}</td>
<td>{loggedby[0][0]} {loggedby[1]}</td>
<td>{note.time}</td>
<td>{note.obsn}</td>
<td>{note.PR}</td>
<td>{note.SPO2}</td>
</tr>"""
    if len(caseObj.timedischarged) > 0:
        htmltop+="""<tr>
                <td class="discharged" colspan="6" style="text-align: center;">
                    <b style="width: 100%;"><i>DISCHARGED</i></b>
                </td>
            </tr>"""
    pdfkit.from_string(htmltop+htmlbottom,
f"/var/www/PIMSFE/case{caseObj.caseno}raw.pdf") # the PDF is generated from the HTML
and is saved unencrypted to the PIMS server
    with open(f"/var/www/PIMSFE/case{caseObj.caseno}raw.pdf", "rb") as in_file:
        input_pdf = PdfReader(in_file)
        output_pdf = PdfWriter()
        output_pdf.append_pages_from_reader(input_pdf)
        output_pdf.encrypt(DOB) # the PDF is encrypted
        with open(f"/var/www/PIMSFE/case{caseObj.caseno}.pdf", "wb") as out_file:
            output_pdf.write(out_file) # the encrypted PDF is saved
    remove(f"/var/www/PIMSFE/case{caseObj.caseno}raw.pdf") # the unencrypted PDF is
deleted
    with open(f"/var/www/PIMSFE/case{caseObj.caseno}.pdf", "rb") as attachment:
        p = MIMEApplication(attachment.read(), _subtype="pdf")
        p.add_header('Content-Disposition', "attachment; filename= %s" %
f"case{caseObj.caseno}.pdf")
        emsg.attach(p)
```

Variables

1. **Patient date of birth**

   The patient's date of birth is used as a pin code to encrypt the exported case pdf file. This helps to protect the contents of the file during transit across the internet and on the patient's device.

Test plan

Test	Method	Expected output
HTML file rendering	A PIMS case will be exported, a test line will be included in the code that will dump the value of `htmltop` and `htmlbottom` to a `.html` file.	The HTML file should contain a completed version of the HTML template file, with all the relevant patient details included.
PDF conversion	A PIMS case will be exported, a test line will be included in the program to halt the process before the unencrypted PDF file is opened and encrypted.	A pdf file should have been generated, with the file name `case{case number}raw.pdf`. The file should be readable, correctly formatted, and include all the relevant patient details and data.
PDF encryption	A PIMS case will be exported.	An encrypted PDF file should have been generated, with the password being the patient's DOB as provided. The file should be readable, correctly formatted, and include all of the relevant patient details and data. The unencrypted version of

		the file should be deleted from the server.

Test log

Test no.	Date	Pass / Fail	Outcome	Alterations
1	5 July 2023	**Pass**	When the filled HTML data was dumped, the file was correctly formatted with the correct data filled.	Halfway through development, the HTML template used - which was originally used in PIMS Tk - was updated. The reworked template has an easier to read design and includes more case data.
2	5 July 2023	**Pass**	Initial PDF conversion was completed without issue. The created file was opened to ensure correct conversion.	N/A
3	6 July 2023	**Pass**	PDF encryption was completed successfully. A stop gap was placed in the code allowing the created file to be retrieved manually. The	N/A

			file could only be accessed after the supplied DOB (or passcode) was entered.	

Send email

Justification

The final step of the PIMS case export process is for the encrypted PDF file to be sent to the patient. This is done by creating a SMTP client to connect to an mail server - in the example of the demo case

The encryption process happens entirely in the background of the software. The program first uses a HTML template to construct a temporary HTML file with all of the patient's data in memory. This HTML file is then converted to a plain-text PDF file which is stored in the server's directory. This file is then opened, read, and encrypted before being resaved ready to be sent, the unencrypted copy is deleted for security and storage capacity saving.

Design & code

```
context = ssl.create_default_context()
smtp_port = 465
smtp_username = "PIMS"
smtp_password = "**REDACTED**"
emsg = MIMEMultipart()
emsg["From"] = "PIMS <PIMS@jamesdev.co.gg>"
emsg["To"] = email
emsg["Subject"] = f"PIMS - Case no. {caseObj.caseno}"
emsg.attach(MIMEText(f"""{patient.name.split(", ")[1]},
Your case report is now ready, you can find it attached to this email.
```

```
The file is password locked to secure your data using the birthdate
your provided, in the format DDMMYY.
Thank you for using PIMS.

Note: this mailbox is not monitored, please do not respond to this
email."""))
# PDF creation logic...
with open(f"/var/www/PIMSFE-DEMO/case{caseObj.caseno}.pdf", "rb") as
attachment:
        p = MIMEApplication(attachment.read(), _subtype="pdf")
        p.add_header('Content-Disposition', "attachment; filename= %s"
% f"case{caseObj.caseno}.pdf")
        emsg.attach(p)
try:
        with smtplib.SMTP_SSL("jamesdev.co.gg", smtp_port,
context=context) as server:
                server.login(smtp_username, smtp_password)
                server.sendmail(emsg["From"], emsg["To"],
emsg.as_string())
                server.quit()

remove(f"/var/www/PIMSFE-DEMO/case{caseObj.caseno}.pdf")
except Exception as e:
        return e
```

Variables

1. **Patient email**

   The address provided on the confirmation page to which the exported case email will be sent.

Test plan

Test	Method	Expected output
SMTP connection	Without sending an email, the SMTP	The client should

established	client will be created and a connection attempt started.	establish connection with the server following SSL handshake and authentication.
Email & attachment sent	A case will be exported.	An email containing an encrypted PDF as an attachment should be sent to the recipient's email address.

Test log

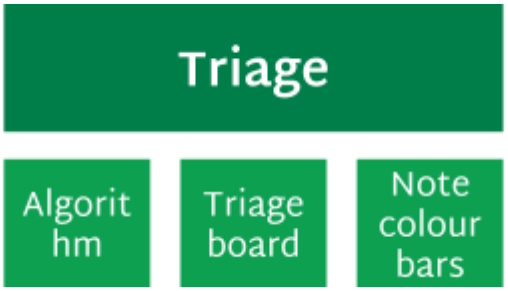
Test no.	Date	Pass / Fail	Outcome	Alterations
1	11 July 2023	Fail	SMTP communication could not be established on port 587.	This was less an issue with logic itself and more with the implementation. My email server *(which was the bane of my existence when I set it up)* uses SMTPS over port 465 - switching the port resolved this issue.
2	11 July	Fail	SMTP authentication failed - no user	This, once again, was less of an issue with

	2023		*"PIMS@JamesDev.co.gg"*	the logic and more with my email server. During authentication, the domain must be removed from the username.
3	11 July 2023	**Pass**	SMTP connection success. PIMS was able to login and authenticate with the jamesdev.co.gg mail server.	N/A
4	12 July 2023	**Pass**	The email object was successfully sent to the recipient email address. Attached was the encrypted case export PDF.	N/A

Project review:

Currently, the email to which the encrypted document is attached is a plain text. While this does work, it is a dated approach, with modern systems favouring HTML emails which feature designs and styling. I have implemented this on another Flask project of mine, who's email algorithms are based off of PIMS, so the process of implementing this would be fairly painless.

Triage



Triage algorithm

Justification

Triaging is a method through which first aiders and other medical personnel can determine priority amongst their patients when the demand for assistance rises above what the attending can comfortably provide. The PIMS triage algorithm is a simple way to determine the status of a patient based on recorded vital signs. From this, the patient is assigned onto a colour scale from green - meaning that their vitals do not show signs of distress - to yellow, orange, and finally red - being the most severe. This simple categorization can serve as a starting point to enable medical personnel to decide how to allocate their time and attention.
The triage algorithm happens entirely in the background. The categories chosen are clear to understand, and follow a widely used scale, with the reading boundaries derived from open clinical standards.

Design & code

```
def triage(pr, spo2):
    pr = int(pr)
    spo2 = int(spo2)
```

```
    level = "green"

    if (50 <= pr <= 110) or spo2 >= 95:
        level = "green"

    if (111 <= pr <= 120) or (90 <= spo2 <= 94):
        level = "yellow"

    if (121 <= pr <= 130) or (80 <= spo2 <= 89):
        level = "orange"

    if pr >= 130 or spo2 <= 80:
        level = "red"

    return level
```

Variables

1. **Pulse rate (pr)**

   The patient's recorded heart rate, as an integer.

2. **Blood oxygen saturation (spo2)**

   The patient's recorded saturation of oxygen in their blood, expressed as a percentage represented as an integer. No lower than zero or higher than one hundred.

Test plan

Test	Method	Expected output
Pulse and oxygen are in the same level and are categorised correctly	A variety of different values from different levels will be run through the algorithm. For each test, both the pulse rate and the SPO2 will be within the same level band.	The algorithm correctly assigns the correct colour band based on the provided readings.

Pulse and oxygen are not in the same level and are categorised correctly	A variety of different values from different levels will be ran through the algorithm. For each test, the pulse rate and the SPO2 will be within the different level band.	The algorithm should assign the patient the colour band of their most severe reading, either pulse rate or SPO2.

Test log

Test no.	Date	Pass / Fail	Outcome	Alterations
1	13 July 2023	Pass	When both the supplied pulse and blood oxygen saturation are independently at the same triage level the correct level is returned.	N/A
2	13 July 2023	Pass	When the supplied pulse and blood oxygen saturation are independently at different triage levels, the most severe level is returned.	N/A

Triage board

Justification

The triage board is a screen designed to quickly show all the currently triaged patients on the PIMS system in their respective colour band. The board automatically collects the data from the most recent recorded note for each case to determine where the patient should appear.

The triage screen is designed to be easily readable and understandable, each patient case appears in the respective colour coded category. A small amount of information is displayed for each case to keep the design simple and clean, the IDs of the patient and case along with the time they were triaged and had their last note record are displayed to aid in deciding who in each category should be prioritised. Lastly, there is a button which allows the operator to quickly open the respective case through the quick access code system. A small piece of javascript code embedded in the webpage will automatically refresh the screen every 30 seconds, allowing the board to reconfigure itself based upon any newly recorded notes without needing the intervention of a human operator.

Design & code

Fig 1: A sample "triage board" window.

```
@main.route('/triage', methods=['GET'])
@login_required
def triageBoard():
    tris = Triage.query.all()
```

```
    greenBundles = []
    yellowBundles = []
    orangeBundles = []
    redBundles = []
    for tri in tris:
        caseObj = Cases.query.get(tri.caseno)
        lastNote = Notes.query.filter_by(caseno =
tri.caseno).order_by(Notes.noteno.desc()).first()
        patient = Patients.query.get(tri.patientno)
        bundle = {'patientname': patient.name,
                  'patientid': patient.patientid,
                  'caseno': caseObj.caseno,
                  'addedat': tri.addedat,
                  'lastnotetime': lastNote.time}

        match triage(lastNote.PR, lastNote.SPO2):
            case "green":
                greenBundles.append(bundle)
            case 'yellow':
                yellowBundles.append(bundle)
            case 'orange':
                orangeBundles.append(bundle)
            case 'red':
                redBundles.append(bundle)

    return render_template('triageboard.html',
refreshtime=datetime.now().strftime('%H:%M:%S'),current_user=current_us
er, greenBundles=greenBundles, yellowBundles=yellowBundles,
orangeBundles=orangeBundles, redBundles=redBundles)
```

Variables

1. **Triage database table**

   The triage database table acts as a link sheet to all the currently triaged patients and their cases.

Test plan

Test	Method	Expected output
The screen renders correctly	The triage screen will be opened.	The triage screen should load as expected from the design, with all of the elements in the correct place.
Patients are correctly triaged	The triage screen will be loaded with triaged patients.	The triaged patients should appear within the right colour coded column on the screen.
Patient information is correctly displayed	The triage screen will be loaded with triaged patients.	The triaged patients should have their patient ID, case ID, time of triage, and time of last case displayed clearly below.
The open case button launches the correct case	The triage screen will be loaded with triaged patients, the open button will be pressed on a case.	The page should redirect back to the home page, with the relevant patient and case open.
The screen automatically refreshes	The triage screen will be loaded with triaged patients and left for 30 seconds.	The page should refresh, with any newly recorded information or patient changes displayed.

Test log

Test no.	Date	Pass / Fail	Outcome	Alterations
1	14 July 2023	Pass	The screen rendered correctly, with correctly formatted triage columns.	N/A
2	14 July 2023	Pass	Patient data was successfully extracted from the bundles supplied to each triage category. This data was then correctly formatted and displayed on the triage board.	N/A
3	14 July 2023	Fail	The open case button did not open the case.	The open case button on the triage board 'piggy-backs' off of the quick access code feature used elsewhere in the program, this required a POST request to be submitted. I hoped to do this through a javascript function, however that did not work correctly with the software. Instead, to each of the

				patients on the board, a small hidden form was added, with the quick open button being the submit for the form.
4	15 July 2023	**Pass**	When clicked, the open case button, using a hidden form, redirected back to the homescreen with the patient and case loaded.	N/A
5	15 July 2023	**Pass**	Every 30 seconds, the javascript timeout function embedded in the HTML forced the page to refresh. This happens 30s from the last time the page was loaded, either manually or automatically.	N/A

Note Colour Bars

Justification

The triage note colour bars appear automatically to the left of notes displayed on PIMS. They are designed to provide an easily readable indication to the patient's status at different times, and to be able to see at a glance how the patient's condition has trended.

With the colour bar being placed on the left of the element, it serves as the first part that the user will read. This gives the operator an instantaneous overview of the current condition of the patient; or to be able to run up and down the patient's history to get a glance on how the patient has progressed while in care.

Design & code

Fig 1: A sample case note. A green triage colour bar, displayed on the left of the note, indicates that the patient was a triage level green when this note was recorded.

```
<div id="NOTEID{{note.noteno}}" class="small-note-container"
onclick="openNote('{{note.noteno}}')">
                <div id="NOTEID{{note.noteno}}-colourbar"
class="small-note-colourbar">
                    <script>
                        var target =
document.getElementById('NOTEID{{note.noteno}}-colourbar');
                        switch (triage('{{note.PR}}', '{{note.SPO2}}'))
{
                            case 'green':
                            target.classList.add("triage-green");
                            break;
                        case 'yellow':
                            target.classList.add("triage-yellow");
```

```
                        break;
                    case 'orange':
                        target.classList.add("triage-orange");
                        break;
                    case 'red':
                        target.classList.add("triage-red");
                        break;
                    }
                </script>
            </div>
            <!-- note HTML goes here -->
</div>
```

```
.triage-green {
    background-color: #67C94D;
    color: #67C94D;
}

.triage-yellow {
    background-color: #FFFF54;
    color: #FFFF54;
}

.triage-orange {
    background-color: #F19F39;
    color: #F19F39;
}

.triage-red {
    background-color: #E93323;
    color: #E93323;
}

.triage-grey {
    background-color: #9A9D9F;
    color: #9A9D9F;
}
```

Variables

1. **Pulse rate (pr)**
   The patient's recorded heart rate, as an integer.
2. **Blood oxygen saturation (spo2)**
   The patient's recorded saturation of oxygen in their blood, expressed as a percentage represented as an integer. No lower than zero or higher than one hundred.

Test plan

Test	Method	Expected output
The colour bar renders correctly.	A patient and case will be opened with notes recorded.	The colour bar should appear within the note shape, on the left edge.
The correct triage level is assigned to the correct note.	A patient and case will be opened with recorded notes. The notes should be of different triage categories in order to correctly test the system.	Each note is assigned a colour bar that corresponds with that note's triage level.

Test log

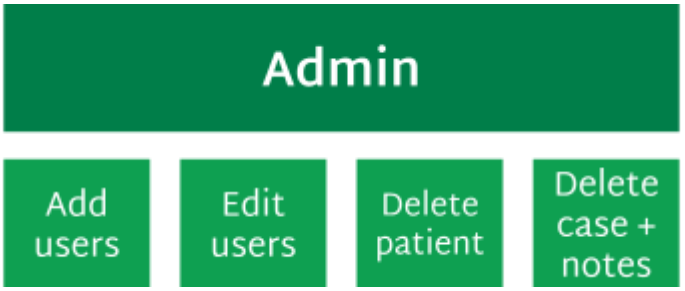
Test no.	Date	Pass / Fail	Outcome	Alterations
1	16 July	Partial pass	The colour bar rendered correctly on the side of the	N/A

	2023		note. However, some other text within the HTML note was moved around due to the element's CSS.	
2	16 July 2023	**Pass**	To each note, the correct colour bar was assigned according to their triage level.	N/A

Project review:

Due to a slight issue with the CSS of the note bar, the addition of the colour bar caused a 'bumping' effect on the text contained within. As such, certain elements of the notes have not been added according to the design, such as the name of the author of the note. This could be resolved with future development, however this might require a complete reworking of the HTML element and its CSS.

Admin



Add users

Justification

Operators are the users on PIMS. Maintaining separate accounts for each operator on the system improves security and safeguards patient data through permission levels. Separate user accounts also allows actions a user takes on the system to be monitored for data security and traceability.

The add operator window uses the common form template, easing the use of this feature by clearly distinguishing between entry fields and buttons. When adding an operator, flags can be placed on the account which will affect how it functions. These flags include the ability to temporarily disable the account, preventing access, or requiring the user to change their password when they next login.

Design & code

Fig 1: A PIMS window, opened to the 'Add operator screen'.

```
@main.route('/utilities/operators/addop', methods=['POST', 'GET'])
@login_required
def addOp():
    if current_user.privs < 10:
         return render_template('error.html', errormessage='403
Forbidden', errortext='This action is only accessible to admin
accounts.')
    if request.method == "GET":
        return render_template('addop.html')
    elif request.method == "POST":
        if current_user.privs < 10:
            return render_template('error.html', errormessage='403
Forbidden', errortext='This action is only accessible to admin
accounts.')
        op = OPS()
        fname = request.form.get('fname') or None
        lname = request.form.get('lname') or None
        privs = request.form.get('privs') or None
        password = request.form.get('password') or None
        password2 = request.form.get('password2') or None
        WANumber = request.form.get('WANumber')
        accargs = request.form.get('accargs') or None
        if None in [fname, lname, privs, accargs, password, password2]:
            return render_template('error.html',
errormessage='Incomplete form', errortext='Please complete all entry
fields marked as required.', redirectURL='/utilities/operators/addop')
        if password != password2:
            return render_template('error.html',
```

```
errormessage="Passwords do not match", errortext="Password must match,
please try again.", redirectURL='/utilities/operators/addop')
        op.fname = fname.upper()
        op.lname = lname.upper()
        op.privs = int(privs)
        op.pwdhsh = ""
        op.WANumber = WANumber
        op.accargs = int(accargs)
        db.session.add(op)
        db.session.commit()
        op.pwdhsh = pwdHSH(op.OPID, password)
        db.session.commit()
        flash(f'Operator {op.fname} {op.lname} has been added, username
{op.OPID}')
        return redirect(url_for('main.operators'))
```

Variables

1. **First and last name**

   The operator's first and last names.

2. **Privileges**

   The operator's privilege level, and integer from 1-10, with 10 being an admin account.

3. **Pager address**

   Links to a deprecated feature, that allowed operators to be quickly contacted using a Discord bot. This feature has now become outdated, and is not integrated into PIMS:FE.

4. **Password**

   Used to authenticate the account. For error detection, this must be entered twice.

5. **Account flag**

   An optional field that modifies the functionality of the user account.

Test plan

Test	Method	Expected output
Form validation works correctly.	The HTML form will be filled out several times correctly and incorrectly.	When submitted, the form should be checked both on the client and server side to make sure the form has been filled correctly. If any required fields are left blank or added incorrectly, the user will be notified.
The user is created and committed to the database.	The HTML form will be filled correctly and submitted.	A user instance is created with the information supplied on the HTML form, the object is then committed to the database.

Test log

Test no.	Date	Pass / Fail	Outcome	Alterations
1	18 July 2023	Pass	Form validation - both within the Python algorithm and HTML code - works correctly,	N/A

			when a field is missed the user is prompted. If the HTML form is bypassed, the code itself will recognise when a field is missed and will return an error page.	
2	18 July 2023	**Pass**	When the form is submitted correctly, a new user record is created in the database and committed.	N/A

Edit users

Justification

Over time, it may become necessary for a user account to be edited to reflect changes to a user's name or privilege level. This could be achieved through manually recalling the database from the server, editing it, then re-uploading it. However, this would require the server to be disabled while the necessary changes were made and presented a risk of the data in the database becoming corrupted or falling into the wrong hands. A cleaner solution was required.

The edit operator window's design is heavily linked to that of the add operator window, this aids in usability by keeping a clear and consistent design across similar functions. The form allows the user's name, privilege level, pager details, and account flags to be edited; for security reasons, the function to change a user's password is kept in a separate window, minimising the risk of an accidental edit.

Design & code

Fig 1: A PIMS window, opened to the 'Edit operator screen'.

```
@main.route('/utilities/operators/editop', methods=['POST'])
@login_required
def editOp():
    if current_user.privs < 10:
        return render_template('error.html', errormessage='403
Forbidden', errortext='This action is only accessible to admin
accounts.')
    op = request.form.get('op')
    op = OPS.query.get(op) or None
    if op is None:
        flash("Operator could not be found or was not selected")
        return redirect(url_for('main.operators'))
    return render_template('editop.html', op=op,
endpoint='/utilities/operators/editop/commit')

@main.route('/utilities/operators/editop/commit', methods=['POST'])
@login_required
def editOpCommit():
    if current_user.privs < 10:
        return render_template('error.html', errormessage='403
Forbidden', errortext='This action is only accessible to admin
accounts.')
    op = request.form.get('OPID')
    op = OPS.query.get(op) or None
    if op is None:
```

```
        flash("Operator could not be found or was not selected")
        return redirect(url_for('main.operators'))
    fname = request.form.get('fname') or None
    lname = request.form.get('lname') or None
    privs = request.form.get('privs') or None
    WANumber = request.form.get('WANumber')
    accargs = request.form.get('accargs') or None
    if None in [fname, lname, privs, accargs]:
        return render_template('error.html', errormessage='Incomplete
form', errortext='Please complete all entry fields marked as
required.', redirectURL='/utilities/operators')
    op.fname = fname.upper()
    op.lname = lname.upper()
    op.privs = int(privs)
    op.WANumber = WANumber
    op.accargs = int(accargs)
    db.session.commit()
    flash('Operator edits have been saved')
    return redirect(url_for('main.operators'))
```

Variables

1. **First and last name**

   The operator's first and last names.

2. **Privileges**

   The operator's privilege level, and integer from 1-10, with 10 being an admin account.

3. **Pager address**

   Links to a deprecated feature, that allowed operators to be quickly contacted using a Discord bot. This feature has now become outdated, and is not integrated into PIMS:FE.

4. **Account flag**

   An optional field that modifies the functionality of the user account.

Test plan

Test	Method	Expected output
The correct user is selected, and the data is recalled.	The edit operator window is opened to a specific user.	The correct user is selected from the database, and all of their information is filled into the correct input boxes on the HTML form.
Form validation works correctly.	The HTML form will be filled out several times correctly and incorrectly.	When submitted, the form should be checked both on the client and server side to make sure the form has been filled correctly. If any required fields are left blank or added incorrectly, the user will be notified.
The edits are applied to the correct user.	The HTML form will be filled correctly and submitted.	The selected user's record on the database will be updated to reflect the inputted values.

Test log

Test no.	Date	Pass / Fail	Outcome	Alterations

1	19 July 2023	**Fail**	When recalling user data, all fields are filled with the data stored accept for the "other options drop down menu"	Unlike with most HTML elements, who's value can be set through the element's attribute, select elements must have an individual option child marked as selected. This was accomplished using Jinja2 and if statements to match values.
2	19 July 2023	**Pass**	User data can be recalled successfully to the edit form.	N/A
3	19 July 2023	**Pass**	Data validation occurs both on the HTML form and in the POST route, to ensure that all the required data has been completed on the form.	N/A
4	20 July 2023	**Pass**	When the form is submitted, the updated data is stored in the database, and the changes committed. This was checked by then recalling the data back to the edit form - showing the contents of the	N/A

			database.	

Delete patient

Justification

The PIMS system stores personal data linkable to patients that would be categorised as “sensitive” under GDPR, and as such patients have the right at any time to have their data permanently deleted. To comply with this, a delete function was required.

To comply with GDPR’s requirement for total data deletion and to maintain referential integrity within the PIMS database, deleting a patient will also remove all associated notes and cases linked to their patient ID.

Design & code

Fig 1: A PIMS window, opened to the ‘Delete a patient’.

```
@main.route('/utilities/deletepatient', methods=["POST", "GET"])
@login_required
```

```
def deletePatient():
    if current_user.privs < 10:
        return render_template('error.html', errormessage='403
Forbidden', errortext='This action is only accessible to admin
accounts.')
    if request.method == "GET":
        return render_template("deletepatient.html",
endpoint="/utilities/deletepatient",
patients=Patients.query.with_entities(Patients.patientid,
Patients.name).all())
    elif request.method == "POST":
        patient = request.form.get('patient')
        patient = Patients.query.get(patient) or None
        if patient is None:
            flash("Patient could not be found or was not selected")
            return redirect(url_for('main.deletePatient'))
        cases = Cases.query.filter_by(patientno =
patient.patientid).all()
        notes = Notes.query.filter_by(patientno =
patient.patientid).all()
        for case in cases:
            db.session.delete(case)
        for note in notes:
            db.session.delete(note)
        db.session.delete(patient)
        flash(f'Patient {patient.name} has been deleted')
        db.session.commit()
        return redirect(url_for('main.index'))
```

Variables

1. **Patient (patient ID)**

   The target patient's ID.

Testing

Test	Method	Expected output

The correct patient is selected.	A patient is selected from the dropdown menu and the form is submitted. A debug halt is placed within the code that will return the selected patient's ID without deleting their data.	The desired patient's ID is returned.
The patient, their cases and notes are deleted.	A patient is selected from the dropdown menu and the form submitted.	The patient, their cases and notes are deleted from the database.

Test log

Test no.	Date	Pass / Fail	Outcome	Alterations
1	21 July 2023	**Pass**	A stop gap was placed in the code to allow debugging by returning the name of the user that the database had recalled based on the HTML form input. This confirmed that the system was selecting the correct patient to delete.	N/A
2	21 July 2023	**Pass**	With the stop gap removed, when the form was submitted the system correctly removed the patient, their cases, and	N/A

			all associated notes from the database.	

Project review:

A commonly included feature when deleting a user's data from a system is a confirmation email and "take-out box". This functions to confirm with the subject that their data has been removed from the system, and to provide them with a personal copy of everything that was stored regarding them on the system. For PIMS, which handles special category personal data, this could be a useful feature to implement, improving transparency and providing the patient with a final record of their medical history.

Delete case

Justification

The deletion of a single case from a patient's history is considered a non-routine operation in PIMS.

To select a case the user is prompted to enter the quick access code (fig. 1), the system then recalls basic information on the patient and case and asks the user to confirm their selection (fig. 2). This allows the operator to ensure that they have selected the correct case and avoids accidental deletion. Once the user confirms their selection, the system will delete the case and all associated notes - in keeping with GDPR and maintaining referential integrity.

Design & code

Fig 1: A PIMS window, opened to the 'Delete a case.

Fig 2: A PIMS window, opened to the 'Confirm delete case'.

```
@main.route('/utilities/deletecase', methods=['POST', 'GET'])
@login_required
def deleteCase():
    if current_user.privs < 10:
        return render_template('error.html', errormessage='403
Forbidden', errortext='This action is only accessible to admin
accounts.')
    if request.method == "GET":
        return render_template('deletecase.html',
endpoint='/utilities/deletecase/confirm')
    if request.method == "POST":
        patient = request.form.get('patientID')
        caseObj = request.form.get('caseID')
        patient = Patients.query.get(patient) or None
        caseObj = Cases.query.get(caseObj) or None
        if None in [patient, caseObj]:
            return render_template('error.html', errormessage='404 Not
found', errortext='Patient or case could not be found in the database,
please try again')
        notes = Notes.query.filter_by(caseno = caseObj.caseno).all()
        for note in notes:
            db.session.delete(note)
        db.session.delete(caseObj)
        db.session.commit()
        flash('Case has been deleted successfully')
        return redirect(url_for('main.index'))

@main.route('/utilities/deletecase/confirm', methods=['POST'])
```

```
@login_required
def deleteCaseConfirm():
    if current_user.privs < 10:
        return render_template('error.html', errormessage='403
Forbidden', errortext='This action is only accessible to admin
accounts.')
    qac = request.form.get('qac') or None
    if qac is None:
        return render_template("error.html", errormessage="Forbidden",
errortext='The passed patient ID does not match patient ID in
cookies.',redirectURL='/utilities/deletecase')
    qac = qac.split('-')
    if  (len(qac) == 3) and (qac[0].upper() == "PIMS"):
        try:
            pid = int(qac[1])
            cid = int(qac[2])
        except ValueError:
            return render_template("error.html", errormessage="400 Bad
request", errortext='The QAC entered is not valid, please check QAC. \
                                QACs are two numbers separated with a
hyphen.',redirectURL='/utilities/deletecase')
        qac.pop(0)
    elif len(qac) != 2:
        return render_template("error.html", errormessage="400 Bad
request", errortext='The QAC entered is not valid, please check QAC. \
                                QACs are two numbers separated with a
hyphen.',redirectURL='/utilities/deletecase')
    patient = Patients.query.get(int(qac[0])) or None
    if patient is None:
        return render_template("error.html", errormessage="404 Patient
not found", errortext='The patient from the QAC entered could not be
found in the database, please check
QAC.',redirectURL='/utilities/deletecase')
    case = Cases.query.get(int(qac[1])) or None
    if case is None:
        return render_template("error.html", errormessage="404 Case not
found", errortext='The case from the QAC entered could not be found in
the database, please check QAC.',redirectURL='/utilities/deletecase')
    if case.patientno != patient.patientid:
        return render_template("error.html", errormessage="Forbidden",
errortext='The QAC entered is not valid, please check
```

```
QAC.',redirectURL='/utilities/deletecase')
    return render_template('deletecaseconfirm.html', patient=patient,
case=case,endpoint='/utilities/deletecase')
```

Variables

1. **Quick Access Code (QAC)**

   The target patient's case ID combined with their user ID.

Test plan

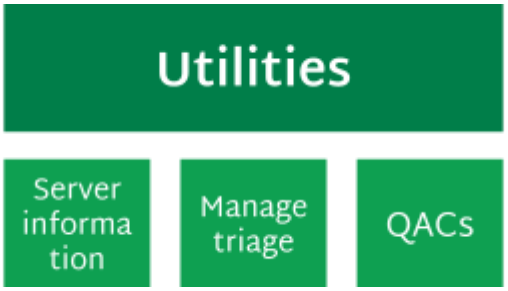
Test	Method	Expected output
The correct patient is selected.	A QAC is entered on the delete case window and the form submitted.	The confirm deletion window should open, displaying relevant data on the correct case.
The case and notes are deleted.	From the confirm deletion window the delete button is pressed.	The case and notes are deleted from the database.

Test log

Test no.	Date	Pass / Fail	Outcome	Alterations
1	22 July	Pass	A stop gap was placed in the code to allow debugging by	N/A

	2023		returning the name of the user that the database had recalled based on the HTML form input. This confirmed that the system was selecting the correct patient to delete.	
2	22 July 2023	**Pass**	With the stop gap removed, when the form was submitted the system correctly removed the patient's cases and all associated notes from the database.	N/A

Utilities



Server information

Justification

The server information window contains key information about the PIMS instance. This information serves little functional purpose, but is included when exporting case files to PDF. The screen is a holdover from PIMS's Flask based precursor, where the screen would typically display the TCP server telemetry, such as the number of requests that had been made. Server information is stored in a `.ini` config file.
The server information panel is accessible to all authenticated users, as the contained information may be critical for operation. Editing the data, however, is restricted to admin users only; edit mode can be toggled by using the button at the bottom of the form which will link to a Javascript function.

Design & code

Fig 1: A PIMS window, opened to the 'server information' screen.

```
@main.route('/utilities/serverinfo', methods=['GET', 'POST'])
@login_required
def serverInfo():
    if request.method == "GET":
        return render_template('serverinfo.html',
orgName=pimsconf['server']['orgname'],
location=pimsconf['server']['location'], cmo=pimsconf['server']['cmo'])
    elif request.method == "POST":
        if current_user.privs < 10:
            return render_template('error.html', errormessage='403
Forbidden', errortext='Server info can only be editted by admin
accounts.', endpoint=url_for('main.serverInfo'))
        orgName = request.form.get('orgName') or None
        location = request.form.get('location') or None
        cmo = request.form.get('cmo') or None
        if None in [orgName, location, cmo]:
            flash('A value is required for all three variables')
            return redirect(url_for('main.serverInfo'))
        pimsconf['server'] = {'orgname': orgName,
                              'location': location,
                              'cmo': cmo}
        with open('/var/www/PIMSFE-DEMO/instance/pimsServer.ini', 'w')
as conffile:
            pimsconf.write(conffile)
        flash('Server information has been updated successfully')
        return redirect(url_for('main.serverInfo'))
```

Variables

1. **Organisation name**

   The name given to the first aid organisation, used to identify the PIMS instance.

2. **Location**

   The location where the first aid is being performed.

3. **Chief Medical Officer**

   The lead first aider assigned to the first aid organisation

Test plan

Test	Method	Expected output
The page renders correctly with the correct data.	The server information page will be opened.	The page should be rendered as shown in fig. 1.
Editing server information is restricted to admins only.	With the server information page open to the edit page, an attempt to save edits will be made by both an admin and non-admin user.	The edits should only be accepted when the admin account is used, otherwise they should be refused and an error message displayed.
Server information can be edited and saved.	With the server information page open to the edit page, an attempt to save edits will be made by an admin user.	The edits should be accepted and the server config file updated. These edits should then be displayed when the window is reloaded.

Test log

Test no.	Date	Pass / Fail	Outcome	Alterations
1	23 July 2023	**Pass**	When loaded, the page renders correctly, following the same common form style used on other pages across the system.	N/A
2	23 July 2023	**Pass**	While viewing the server information is not restricted, the POST route to edit the information is contained to admin users only, preventing tampering.	N/A
3	23 July 2023	**Pass**	Edits made to the server information are successfully stored within the .ini config file. This can be checked by reopening the server information page, showing the updated information.	N/A

Project review:

In PIMS Tk, the server information window serves an additional purpose of displaying server telemetry data, such as uptime, request count, and boot time. This was useful for debugging and helped to provide information on the health of the

system. While PIMS FE is more stable than Tk, these telemetry features could still be helpful for debugging - and could be expanded to show additional information on the user settings (i.e. cookie data).

Manage triage

Justification

When a patient is placed into triage, their name will appear on a categorised window until they are discharged. However, sometimes it may be necessary for a patient to be removed from triage before they are discharged, in which case the manage triage window can be utilised.

The manage triage window will only open when there is a patient on the triage board, otherwise a message will be displayed stating that the board is blank. When opened, the page is designed to be uncomplicated, with a simple dropdown box being used to select the user the patient wishes to remove from triage. This was chosen over a checkbox style list to reduce the risk of the accidental removal of a patient and to discourage use of the feature (removal through discharge is preferred.

Design & code

Fig 1: A PIMS window, opened to the 'manage triage' screen.

```
@main.route('/utilities/managetriage', methods=['POST', 'GET'])
@login_required
def manageTriage():
    if request.method == "GET":
        patients = Triage.query.all()
        if (patients is None) or (len(patients) == 0):
            flash("There are no patients on the triage board")
            return redirect(url_for('main.utilities'))
        triagedPatients = []
        for p in patients:
            triagedPatients.append(Patients.query.get(p.patientno))
        return render_template('managetriage.html',
endpoint="/utilities/managetriage", patients=triagedPatients)
    if request.method == "POST":
        triageCase = request.form.get('patient')
        triageCase = Triage.query.get(triageCase) or None
        if triageCase is None:
            flash('Triaged patient could not be found or was not
selected')
            return redirect(url_for('main.manageTriage'))
        triagedPatient = Patients.query.get(triageCase.patientno)
        db.session.delete(triageCase)
        db.session.commit()
        flash(f'Patient {triagedPatient.name} was removed from triage')
        return redirect(url_for('main.index'))
```

Variables

1. **Patient**

   The triaged patient to be removed from the triage board

Test plan

Test	Method	Expected output
The correct patient is removed from triage.	The triage patient window is opened, a user selected, and the form submitted.	The selected user is removed from the triage board.

Test log

Test no.	Date	Pass / Fail	Outcome	Alterations
1	25 July 2023	Pass	A stop gap was placed in the code to allow debugging by returning the name of the user that the database had recalled based on the HTML form input. This confirmed that the system was selecting the correct patient to remove from triage.	N/A
2	25 July	Pass	The triage record for the selected patient is deleted	N/A

	2023		from the database.	

Quick Access Code

Justification

Quick Access Codes (QACs) are short numerical identifiers given to PIMS cases that allows for their quick recall. QACs combine several primary keys used across the database into a form that can be easily remembered or jotted down.

QACs have to levels of 'resolution': case level and note level. Case level QACs are provided in the format PIMS-*patient ID-case ID*, when entered PIMS will recall and load the patient's data and the data on their specified case - this is achieved through setting the patient and case ID cookies to the specified values. The user is then redirected back to the homepage, which will then recall the data for the case. Note level QACs append the note ID to the end of the string, the user is instead redirected to the view note screen with the mentioned note displayed. The QAC entry field allows the entry of QACs both with and without the PIMS prefix.

Design & code

Fig 1: A PIMS window, opened to the 'open QAC' screen.

```
        @main.route('/openQAC', methods=['POST', 'GET'])
@login_required
def openQAC():
    if request.method == 'GET':
        return render_template('openqac.html', endpoint='/openQAC')
    if request.method =="POST":
        qac = request.form.get('qac') or None
        if qac is None:
            return render_template("error.html",
errormessage="Forbidden", errortext='The passed patient ID does not
match patient ID in cookies.',redirectURL='/openQAC')
        qac = qac.split('-')
        if qac[0].upper() == "PIMS":
            try:
                pid = int(qac[1])
                cid = int(qac[2])
                if len(qac) == 4:
                    nid = int(qac[3])
            except ValueError:
                return render_template("error.html", errormessage="400
Bad request", errortext='The QAC entered is not valid, please check
QAC. \
                                        QACs are two or three numbers
separated with a hyphen.',redirectURL='/openQAC')
            qac.pop(0)
        elif (len(qac) < 2) or (len(qac) > 3):
            return render_template("error.html", errormessage="400 Bad
request", errortext='The QAC entered is not valid, please check QAC. \
```

```
                                    QACs are two numbers separated with
a hyphen.',redirectURL='/openQAC')
        patient = Patients.query.get(int(qac[0])) or None
        if patient is None:
            return render_template("error.html", errormessage="404
Patient not found", errortext='The patient from the QAC entered could
not be found in the database, please check
QAC.',redirectURL='/openQAC')
        case = Cases.query.get(int(qac[1])) or None
        if case is None:
            return render_template("error.html", errormessage="404 Case
not found", errortext='The case from the QAC entered could not be found
in the database, please check QAC.',redirectURL='/openQAC')
        if case.patientno != patient.patientid:
            return render_template("error.html",
errormessage="Forbidden", errortext='The QAC entered is not valid,
please check QAC.',redirectURL='/openQAC')
        if len(qac) == 3:
            if (Notes.query.get(qac[2]) or None) is not None:
                if (Notes.query.get(qac[2]).patientno == int(qac[0]))
and (Notes.query.get(qac[2]).caseno == int(qac[1])):

resp=make_response(render_template('opennotelink.html', noteid=qac[2]))
                else:
                    resp =
make_response(redirect(url_for('main.index')))
            else:
                resp = make_response(redirect(url_for('main.index')))
        else:
            resp = make_response(redirect(url_for('main.index')))
        resp.set_cookie('patientID', qac[0])
        resp.set_cookie('caseID', qac[1])
        return resp
```

Variables

1. **Patient ID component**

   The first numerical component of a QAC, serves as the ID of the patient.

2. **Case ID component**
   The second numerical component of a QAC, serves as the ID of the case.
3. **Note ID component**
   The optional third numerical component of a QAC, serves as the ID of the note.

Testing

Test	Method	Expected output
The correct patient, case, and note is loaded.	A valid QAC is entered into the form and submitted.	The patient ID, case ID, and note ID cookies are changed to match the inputted values.
Invalid QACs are rejected	The open QAC window is opened and invalid QACs are entered. Invalid QACs include: • Malformed QACs, and • QACs with invalid ID keys, and • Mismatched patients, cases, and notes - they must all be linked.	The system rejects the QACs, prompting the user to the reason why they were rejected.
The user is redirected to the correct page after entry.	A valid QAC is entered into the form and submitted.	Case resolution QACs should redirect the user to the home page, where case information can be viewed. Note resolution QACs should redirect the user to the view note page, displaying the

		applicable note's data.

Test log

Test no.	Date	Pass / Fail	Outcome	Alterations
1	26 July 2023	Fail	QACs with a PIMS- prefix are considered invalid and will not load.	Alterations were made to the QAC decompilation process to recognise if the supplied QAC has the PIMS prefix. If so, it is removed before the process continues.
2	26 July 2023	Pass	QACs will open the correct patient, case, and note (if specified).	N/A
3	26 July 2023	Pass	Invalid or malformed QACs are rejected and not opened. This includes where the case does not belong to the specified patient, the note does not belong to the specified case, and visa versa.	N/A