

SENG2011 19T3



Final Report

# medics

Abanob Tawfik	z5075490
Kevin Luxa	z5074984
Lucas Pok	z5122535
Michael Yoo	z5165635
Rason Chia	z5084566

# 1 Executive Summary

Project Vampire is an electronic records management system for the handling of medical blood inventories.

The project aims to replace paper-based records systems and provide typical benefits of digital records management – such as reducing costs, consistency and improving usability. However, concerns are often raised regarding the reliability of a digital system in managing a critical resource with human lives at stake. To provide assurance, our system uses formal verification techniques to ensure that it conforms to safety requirements.

Our system integrates with pre-existing business processes of blood inventory management. For example, there are built-in processes for the expiration of blood, distinction between blood types, requests and acceptance of donated blood, to list a few.

Our system comprises of a graphical interface that interacts with a verified datastore service. The implementation details will be detailed in their respective sections.

Our final project and report will describe a minimum viable product to demonstrate that safety and assurance requirements can be met with a digital records management system.

## 2 Requirements

### 2.1 Stakeholders

- **Vampire staff**
  - Handles the physical delivery of requested blood to hospitals
  - Physically disposes of expired blood by sending it to a blood disposal service
- **Hospital staff**
  - Can request blood to treat patients
  - Can query the level of different types of blood in Vampire's inventory
- **Batmobile admin** [Vampire employee]
  - Deposits blood into the system
- **Pathology staff**
  - Tests all blood within the system
- **Member of the public (potential donor)**
  - Finds out when, where they can donate blood, and if they can donate
- **Blood donor**
  - Donates blood to the Batmobile
  - Can see the outcome of their blood donation
- **Emergency donor**
  - Upon hitting a critically low supply of blood in the system, these people will donate blood via the Batmobile
  - Firefighters, police officers, volunteers, actively registered through the Vampire system
- **Vampire headquarters**
  - Business owners
  - Manage all critical and sensitive actions
- **Blood dump**
  - Will eliminate blood in a medically suitable fashion

### 2.2 Priority key

- **P1 (Priority 1, Must have)**
  - All these requirements form the minimum viable product, crucial to launch
- **P2 (Priority 2, Should have)**
  - All these requirements should be done in the time frame, but are not critical to launch, can be delayed for future release
- **P3 (Priority 3, Could have)**
  - All these requirements are features we would like, but in the time frame we may delay them to future releases to focus on priority 1 and priority 2
- **P4 (Priority 4, Would have)**
  - Features we do not expect to have but show the roadmap of where we see our product down the line

## 2.3 Requirements

### 1. Querying the system. [P1]

#### 1.1. Hospital staff can enquire about the levels of different types of blood in Vampire's inventory. [P1]

*Assumptions:*

- As in the real world, there are 8 blood types: A+, A-, B+, B-, O+, O-, AB+, and AB-.
- The disposal of expired blood occurs daily so queries will always return information about the amount of fresh blood.
- Whole blood can be divided into separate components (such as red blood cells, platelets, and plasma), and each component has a different lifespan. For simplicity, we assume that Vampire only needs to store whole blood.

##### 1.1.1. Hospital staff can see the total amount of each blood type in the inventory. [P1]

**1.1.1.1.** Requires an aggregation function that takes a collection of blood bags and produces a collection of results, where each result item consists of a blood type and an amount. [P1]

##### 1.1.2. Hospital staff can sort results by blood type to see which types are more abundant. [P2]

**1.1.2.1.** Requires a sorting function that takes a collection of result items (described in *requirement 1.1.1.1*) and sorts them by amount. [P2]

##### 1.1.3. Hospital staff can use a blood type filter to see information relating to particular blood types. [P2]

**1.1.3.1.** Requires a filtering function that takes a collection of blood bags and a set of blood types and produces a collection containing only those blood bags that contain one of the specified types. [P2]

##### 1.1.4. Hospital staff can view the results of the query in an easy-to-read format (such as a table). [P1]

#### 1.2. A member of the public (potential donors) may enquire about how, when and where they may give blood. [P4]

##### 1.2.1. Potential donors can find out what the health requirements are for giving blood. [P4]

**1.2.1.1.** Requires the Vampire site to have a page that lists the health requirements for giving blood. [P4]

##### 1.2.2. Potential donors can find out which locations the Batmobile will visit. [P4]

**1.2.2.1.** Requires the Vampire site to have a page that displays the Batmobile's schedule. [P4]

#### 1.3. A donor may enquire about the outcome of their blood donation. [P4]

*Assumptions:*

- When the hospital uses a donor's blood to treat a patient, the hospital will acknowledge the use of the blood.

##### 1.3.1. Donors can check the results of their blood test to see if their blood was clean or unclean, and if unclean, which tests came back positive for disease. [P4]

- 1.3.1.1. Requires the system to keep a record of all blood donations that have been made. [P4]
- 1.3.2. Donors can check if their blood was used to save another life. [P4]
  - 1.3.2.1. Requires the system to have a portal through which hospital staff can acknowledge the use of a donor's blood. [P4]

## 2. Hospital staff can request blood from Vampire. [P1]

*Assumptions:*

- Hospital staff can request multiple blood types at the same time. So a full request (called a **batch of requests**) will be in the form of a mapping from blood type to bags/volume requested.
  - Since Vampire staff dispose of expired blood daily, the inventory will contain only fresh blood at all times, so there is no need to filter out expired blood.
  - For simplicity, blood delivery and transportation occur instantaneously.
- 2.1. The system must be able to check if the batch of requests can be fulfilled. [P1]
    - 2.1.1. Requires a function that takes a collection of blood bag objects and a collection of requests and determines if the request can be fulfilled. [P1]
      - 2.1.1.1. See *requirement 1.1.1.1* [P1]
  - 2.2. The system must fulfill requests that can be fulfilled. [P1]
    - 2.2.1. Requires a function that fulfills a full batch of requests. [P1]
      - 2.2.1.1. Requires a function that fulfills a single request, which takes the blood inventory and split it into a collection that is used to fulfill the requests, and the remaining inventory. [P1]
    - 2.2.2. The system should additionally display a message to the hospital staff to confirm that the request has been fulfilled. [P2]
  - 2.3. The system must send an alert to Vampire headquarters if a request cannot be fulfilled (see *requirement 4*). [P1]
    - 2.3.1. The system should also inform the hospital staff of the current situation and assure them their request will be handled but may be delayed. [P2]
  - 2.4. The system should prioritise blood that is closer to expiring to minimise wastage. [P2]
    - 2.4.1. Requires a function that sorts a collection of blood bags by age. [P2]
  - 2.5. The system should check if the inventory has low blood supply after fulfilling a request (see *requirement 4.2*). [P1]

## 3. Vampire staff can dispose of all expired blood within the inventory. [P1]

*Assumptions:*

- Expired blood is disposed of daily.
  - All expired is sent to the blood dump so that the blood can be disposed of in a medically suitable fashion.
- 3.1. Vampire staff can make a request to the system to remove expired blood from the inventory. [P1]
    - 3.1.1. Requires a function to filter a collection of blood bags by age. [P1]
  - 3.2. The system should check if the inventory has low blood supply after disposing of expired blood (see *requirement 4.2*). [P1]

#### 4. Vampire staff will ensure there is enough blood to satisfy all requests. [P1]

*Assumptions:*

- The default 'low supply' is assumed to be 50 bags or fewer.
- Every blood bag contains the same volume of blood, 3 Litres.
- Vampire staff can configure the threshold for 'low supply' by number of bags.
- Once Vampire Headquarters receives a low blood supply alert, Vampire headquarters will do something (e.g., acquire blood from emergency donors) to push the blood supply up to non-critical levels. Vampire Headquarters must acknowledge the request.
- There are enough emergency donors at all times to push the blood supply up to a non-critical level.

4.1. The system must check if the inventory is low on supply for any blood types after performing any operation that removes blood. [P1]

4.1.1. Requires a function that takes a collection of blood bags and determines if there are any blood types that are in low supply. Blood types that are in low supply are added to an 'alert' collection. [P1]

4.1.1.1. See *requirement 1.1.1.1* [P1]

4.2. The system must automatically alert Vampire headquarters if there is a low supply of any blood type. [P1]

#### 5. Depositing tested blood. [P1]

5.1. Pathology staff can deposit clean, tested blood into the inventory. [P1]

*Assumptions:*

- The Batmobile never deposits blood directly into Vampire's inventory - it always passes the blood to Pathology for testing, which then tests the blood and then deposits the blood into Vampire's inventory if the blood was clean.
- Donors can also donate at a hospital. The process is still the same - the blood is passed to pathology for testing.
- Pathology staff sends unclean blood to the blood dump so that it can be disposed of in a medically suitable fashion.
- Pathology staff enter details of unclean blood test into Vampire's system so that the donor can be notified.
- Pathology staff properly tests the blood for infectious diseases.
- All blood that is stored within the inventory is clean blood that contains no positive results for diseases.
- Information about the origin of the blood (donor name, donor id, blood type, donation date, donation location) is available, and was taken during the donation.
- For simplicity, it is assumed that the transportation of blood is instantaneous.
- It's assumed that in *requirement 5.1.1*, Pathology staff entering the details of the tested blood is equivalent to the blood being deposited into the inventory

5.1.1. Pathology staff can enter the details of the tested clean blood into the system. [P1]

5.1.1.1. The details must include the donor's name, id, and blood type, and the date and location where the donation occurred. [P1]

5.1.1.2. Requires a function that takes a collection of clean blood and inserts it into the inventory. [P1]

5.1.1.3. Requires the system to store blood test results. [P3]

- 5.2. **Pathology staff may enter details of blood tests for unclean blood. [P3]**
  - 5.2.1. The details include the donor's name, id, and blood type, the date and location of the donation, and tests that came up positive. [P3]
  - 5.2.2. Requires the system to store blood test results (see *requirement 5.1.1.3*). [P3]
- 6. **All relevant users have an account on the system. [P4]**
  - 6.1. Requires a database to store account details, e.g., ids, email addresses, (encrypted) passwords, notifications. [P4]
  - 6.2. New users are able to create an account on the system. [P4]
    - 6.2.1. Requires a sign-up page. [P4]
    - 6.2.2. New donors can sign up with just their name and email address. [P4]
    - 6.2.3. Medical professionals (e.g., hospital staff and Pathology staff) need to provide additional identification such as proof of employment. [P4]
      - 6.2.3.1. May require the use of an external API service to prove identity and employment. [P4]
  - 6.3. Users can log in to their account with their id and password. [P4]
    - 6.3.1. Requires a login page. [P4]
    - 6.3.2. Requires a server and database to authenticate users (see *requirement 6.1*). [P4]
  - 6.4. Every user has a profile and can update their personal details, such as their phone number and password. [P4]
    - 6.4.1. Requires a profile page and options to update and save details. [P4]
    - 6.4.2. Requires a database to store and update users' details. [P4]
  - 6.5. When signed in, a user can perform the actions they are authorised to do. [P4]
    - 6.5.1. Requires a menu to enable users to select the action they wish to perform. [P4]
    - 6.5.2. Requires pages for each action (e.g., querying the inventory, submitting blood test details) and a server to handle actions. [P4]

## 3 Use-Cases

The following use cases cover all Priority 1 requirements in addition to some Priority 2 requirements.

### 3.1 Request blood

<b>Use Case</b>	Hospital staff request blood from Vampire
<b>Actor</b>	Hospital staff
<b>Stakeholders</b>	Hospital staff, Vampire staff, Vampire Headquarters
<b>Overview</b>	A hospital needs blood to treat patients. An employee at the hospital makes a request to the system specifying how much of each blood type they want. The system checks if there are sufficient supplies to carry out the request. If there is, the system delivers the requested amount. Otherwise, the system sends an alert to Vampire HQ.
<b>Category</b>	Priority 1
<b>Trigger</b>	A hospital is low on blood and needs more to treat patients.
<b>Precondition</b>	The requested blood types are valid, blood in Vampire's inventory is not expired
<b>Postcondition</b>	If the request can be fulfilled, then the blood used to fulfill the request is removed from the inventory, and the hospital receives the blood. Otherwise, Vampire headquarters is alerted of the blood shortage and no change is made to the inventory.

#### 3.1.1 Basic flow

Description		Requirement(s)
1	A hospital staff queries the system to see the levels of different blood types in Vampire's inventory.	1.1.1
2	The system determines the total amount of each blood type in the inventory by aggregating the blood based on type.	1.1.1.1
3	The system returns the result of the query and it is displayed to the user.	1.1.4
4	The user filters the query result by selected blood type.	1.1.3
5	The system responds by showing the volume of available blood of only those types, after filtering out other blood types.	1.1.3.1
6	The hospital staff then sends a request / multiple requests to the system for amounts of different blood types.	2
7	The system checks if the request(s) can be fulfilled and determines that it can be fulfilled.	2.1
8	For each blood type requested, the system fulfills the request by filtering the inventory to get blood of that type, and then removing the appropriate amount from the inventory.	2.2 2.2.1 2.2.2.1



9	To minimise wastage, the system may choose to use the oldest blood to fulfill the request, by first sorting the blood by age.	2.4
10	The system displays a confirmation message to the user.	2.2.2
11	The hospital receives the blood and uses it to treat patients.	Requirement 2 Assumption
12	The system checks if there is still a sufficient supply of each blood type and finds that there is.	2.5 / 4.1

### 3.1.2 Alternative flow 1

Description	Failure scenario – not enough blood in inventory to fulfill request.	Requirement(s)
1	A hospital staff queries the system to see the levels of different blood types in Vampire's inventory.	1.1.1
2	The system determines the total amount of each blood type in the inventory by aggregating the blood based on type.	1.1.1.1
3	The system returns the result of the query and it is displayed to the user.	1.1.4
4	The hospital staff then sends a request / multiple requests to the system for amounts of different blood types.	2
5	The system checks if the request(s) can be fulfilled and determines that it can't be fulfilled.	2.1
6	The system sends an alert to Vampire headquarters so they can do something about the shortage.	2.3
7	The system displays a message to the user to inform them of the situation and that their request will be delayed.	2.3.1

### 3.1.3 Alternative flow 2

Description	Partial failure scenario – low supply after fulfilling the request.	Requirement(s)
1	A hospital staff queries the system to see the levels of different blood types in Vampire's inventory.	1.1.1
2	The system determines the total amount of each blood type in the inventory by aggregating the blood based on type.	1.1.1.1
3	The system returns the result of the query and it is displayed to the user.	1.1.4
4	The hospital staff then sends a request / multiple requests to the system for amounts of different blood types.	2
5	The system checks if the request(s) can be fulfilled and determines that it can be fulfilled.	2.1
6	For each blood type requested, the system fulfills the request by filtering the inventory to get blood of that type, and then removing the appropriate amount from the inventory.	2.2 2.2.1 2.2.1.1

7	To minimise wastage, the system may choose to use the oldest blood to fulfil the request, by first sorting the blood by age.	2.4
8	The system displays a confirmation message to the user.	2.2.2
9	The hospital receives the blood and uses it to treat patients.	Requirement 2 Assumption
10	The system checks if there is still a sufficient supply of each blood type and finds that the inventory is low on a particular blood type.	2.5 / 4.1
11	The system sends an alert to Vampire headquarters so they can do something about the shortage.	4.2

## 3.2 Dispose of expired blood

<b>Use Case</b>	Vampire staff dispose of expired blood to ensure Vampire's blood inventory does not have any expired blood.
<b>Actor</b>	Vampire staff
<b>Stakeholder</b>	Vampire staff, Blood dump, Vampire Headquarters
<b>Overview</b>	Blood donations will eventually expire, so Vampire needs to frequently search the blood inventory for expired blood and dispose of it.
<b>Category</b>	Priority 1
<b>Trigger</b>	It is the start of the day.
<b>Precondition</b>	None.
<b>Postcondition</b>	All expired blood at the time of the request is removed from the store and sent to the blood dump. Additionally, if disposing expired blood causes Vampire to have low supply for any blood type, an alert will be sent to Vampire headquarters.

### 3.2.1 Basic flow

Description		Requirement(s)
1	At the beginning of the day, a Vampire staff makes a request to the system to remove all expired blood.	3 3.1
2	The system filters the blood by age to find all the expired blood and removes it from the inventory.	3.1.1
3	The expired blood is sent to the blood dump.	Requirement 3 Assumption
4	The system checks if there is still a sufficient supply of each blood type and finds that there is.	3.2 / 4.1

### 3.2.2 Alternative flow

Description		Requirement(s)
1	At the beginning of the day, a Vampire staff makes a request to the system to remove all expired blood.	3
2	The system filters the blood by age to find all the expired blood and removes it from the inventory.	3.1 3.1.1
3	The expired blood is sent to the blood dump.	Requirement 3 Assumption
4	The system checks if there is still a sufficient supply of each blood type and finds that there isn't.	3.2 / 4.1
5	The system sends an alert to Vampire headquarters so they can do something about the shortage.	4.2

### 3.3 Deposit tested blood

Use Case	Pathology staff deposit clean blood into Vampire's inventory after running various blood tests on it.
Actor	Pathology staff
Stakeholder	Pathology staff, Vampire staff
Overview	After testing donor blood, Pathology staff will enter the details of the clean blood into Vampire's system and deposit the clean blood into Vampire's inventory.
Category	Priority 1
Trigger	Pathology has finished testing a batch of donor blood.
Precondition	The blood to be inserted is clean
Postcondition	The blood is added to Vampire's inventory

#### 3.3.1 Basic flow

Description		Requirement(s)
1	Pathology staff enters details of the expired blood into the system, including the donor's name, id, and blood type, and the date and location of the donation	5.1.1 5.1.1.1
2	At the same time, the blood is deposited into Vampire's inventory	Requirement 5 Assumption 5.1.1.2

### 3.4 Maintain blood inventory

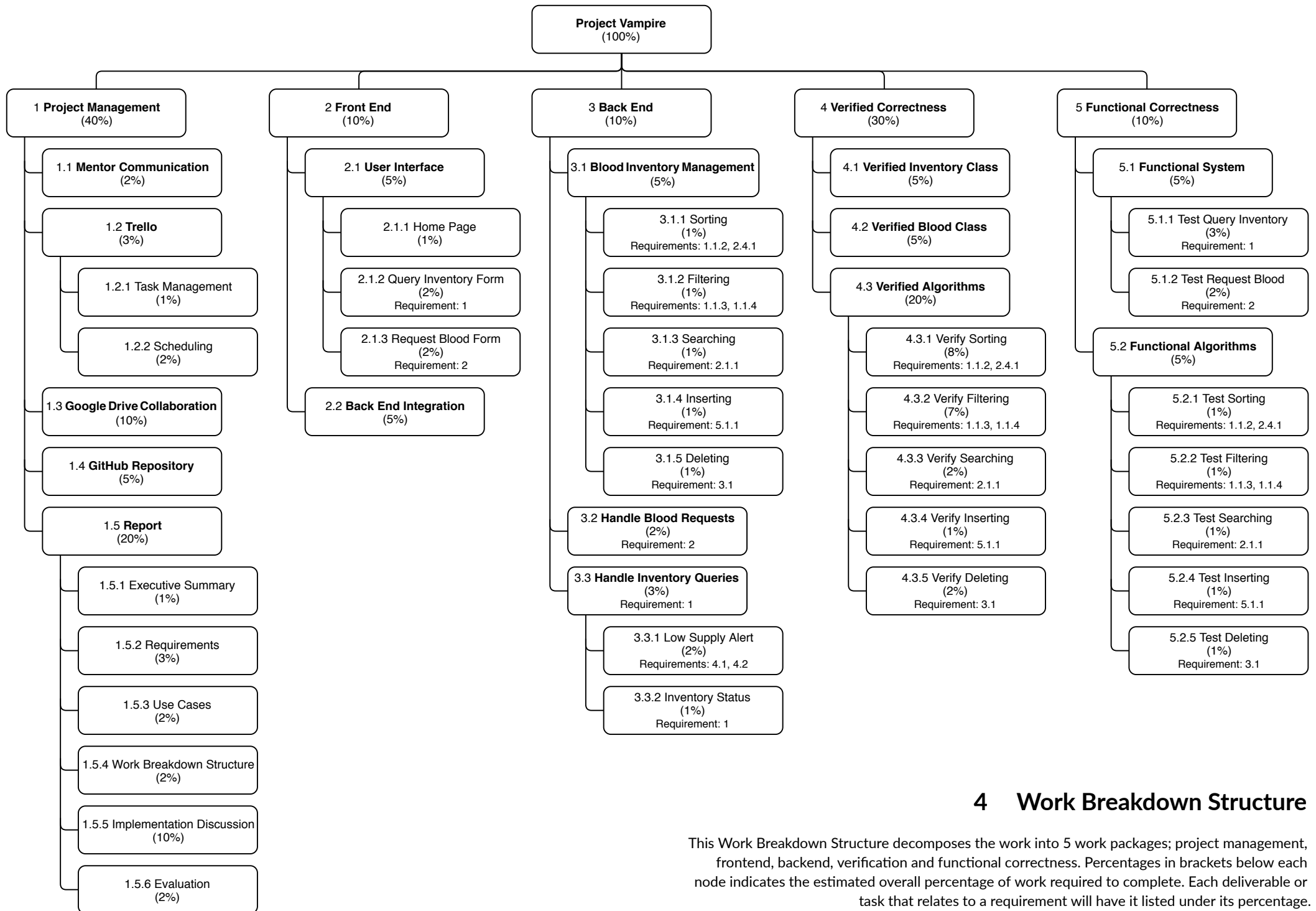
<b>Use Case</b>	Vampire staff ensure that Vampire Headquarters is always notified of a low supply of any blood type.
<b>Actor</b>	Vampire system
<b>Stakeholder</b>	Vampire staff, Vampire Headquarters
<b>Overview</b>	The system must notify Vampire headquarters when blood levels drop below 50 bags of blood for any type of blood.
<b>Category</b>	Priority 1
<b>Trigger</b>	An update is made to the inventory that removes blood.
<b>Precondition</b>	User is registered Vampire staff.
<b>Postcondition</b>	If Vampire's inventory has a low supply of any blood supply Vampire Blood inventory replenished.

#### 3.4.1 Success scenario 1

<b>Description</b>		<b>Requirements</b>
	Success scenario 1 - There is enough blood supply for each blood type so no action is taken	
1	After an update to the inventory that removes blood (e.g., fulfilled blood request, disposal of expired blood), the Vampire system automatically checks the levels of all blood types.	4.1
2	Any blood type below the 'low-blood' threshold will be recorded on an alert list as system cycles through each blood type.	4.1.1
3	There are no blood types that are in low supply so nothing is done	4.2

#### 3.4.2 Success scenario 2

<b>Description</b>		<b>Requirements</b>
	Success scenario 2 - Vampire HQ is notified about a low supply of some blood type(s) in Vampire's inventory and Vampire HQ restores the blood supply	
1	After an update to the inventory that removes blood (e.g., fulfilled blood request, disposal of expired blood), the Vampire system automatically checks the levels of all blood types.	4.1
2	Any blood type below the 'low-blood' threshold will be recorded on an alert list as system cycles through each blood type.	4.1.1
3	The system pushes an alert to Vampire Headquarters for all blood types on the alert list.	4.2
4	The system requires the acknowledge button to be clicked on by Vampire Headquarters for the alert to be dismissed.	Requirement 4 Assumption
5	Vampire Headquarters does something about the shortage and pushes the blood supply levels up to a non-critical level.	Requirement 4 Assumption



## 4 Work Breakdown Structure

This Work Breakdown Structure decomposes the work into 5 work packages; project management, frontend, backend, verification and functional correctness. Percentages in brackets below each node indicates the estimated overall percentage of work required to complete. Each deliverable or task that relates to a requirement will have it listed under its percentage.

## 5 Front-End Implementation

### 5.1 Technology

We opted to implement our front-end with a responsive web application, due to its superior flexibility and cross-platform nature. To minimise development time, we built the application with the Angular 7 framework using TypeScript, HTML templates and CSS. Additionally, existing JavaScript and CSS libraries such as *Bootstrap 4*, *ng-bootstrap* and *Chart.js* were utilised to create a consistent and intuitive user interface. These assets are then bundled with Webpack to form a single page application, and are served by a Node.js web server.

The responsive single page application is easily accessible using a modern web browser, and provides a seamless user experience by eliminating the need to reload the page. The page also updates dynamically upon changes to various HTML controls (input fields, buttons, dropdowns), achieved by binding controls to variables in TypeScript components. Hence, users are provided instantaneous feedback upon interacting with the application.

Communication to the model is achieved by sending HTTP requests to the various endpoints of the back-end server, which in turn returns appropriate responses. This strategy is employed to query and update the state of the system; for example, fetching the blood inventory to display in a table format. Currently, authentication is handled entirely in TypeScript through request interception, although we intend to transfer this to a proper authentication server for deployment.

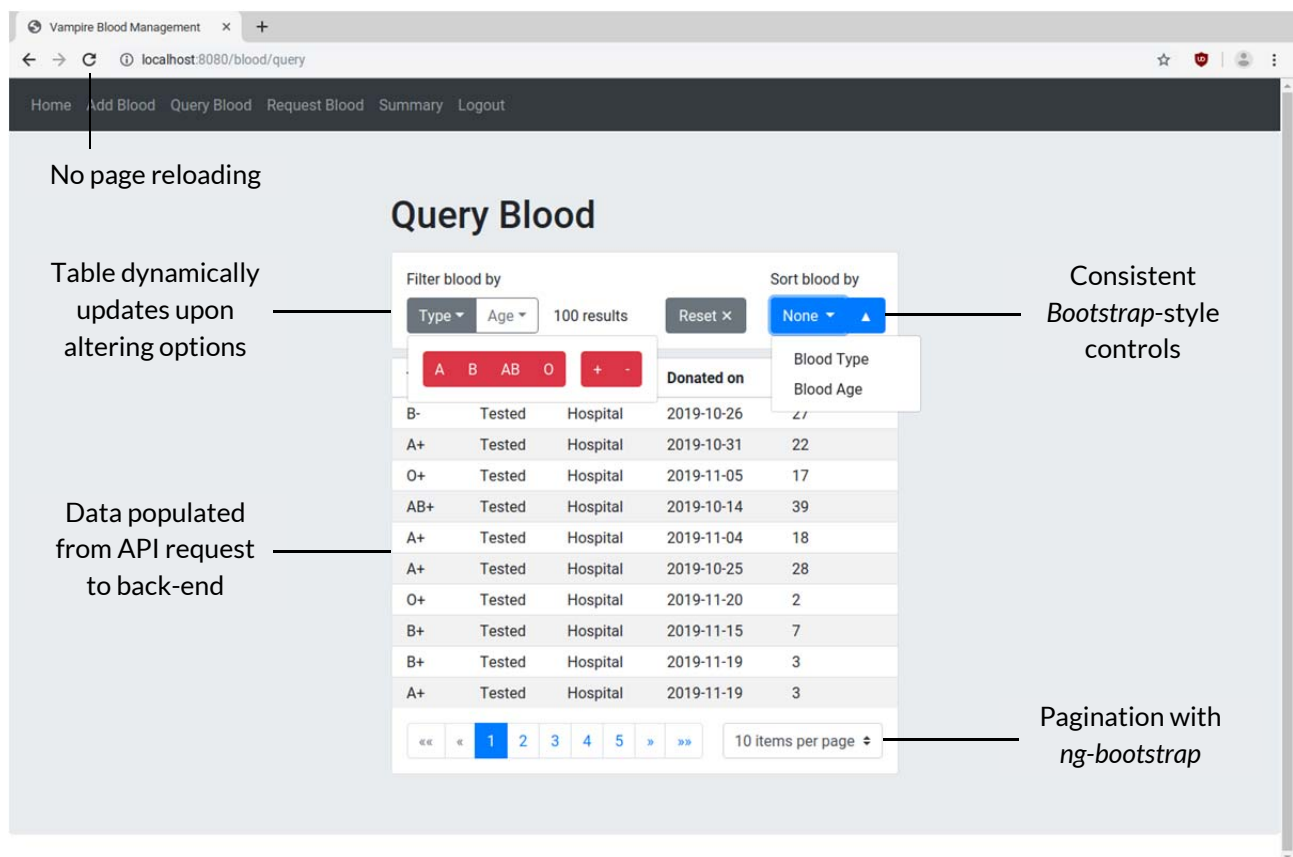


Figure: Front-end interface

Refer to Appendix Front-End for further interface screenshots.

## 5.2 Structure

### User management components

- Login component – authenticate an existing user on the system
- Register component – register a new user on the system

### Home component

- Displays expiring blood and a button to activate disposal of expired blood
- Displays blood types with inventory levels under *low blood threshold*
- (Admin function) Update the *low blood threshold*
- (Admin function) Delete an existing user
- (Admin function) Fix alerts – dismiss low blood warnings by simulating donations from emergency donors

### Add blood component

- Displays a form for entering blood into the inventory, with fields: Donor Name, Blood Type, Blood Status (Tested/Untested), Date Donated, and Location Acquired
- Upon submitting, alerts the user of the outcome of their add blood request

### Query blood component

- Displays a table of blood bags (by default with all blood in inventory), with fields: Blood Type, Blood Status, Date Donated, Age of Blood and Location Acquired
- Provides options to sort by Blood Type and Age of Blood
- Provides options to filter by Blood Type and Age of Blood (upper and/or lower bounded)
- Shows the total count of matching blood bag results

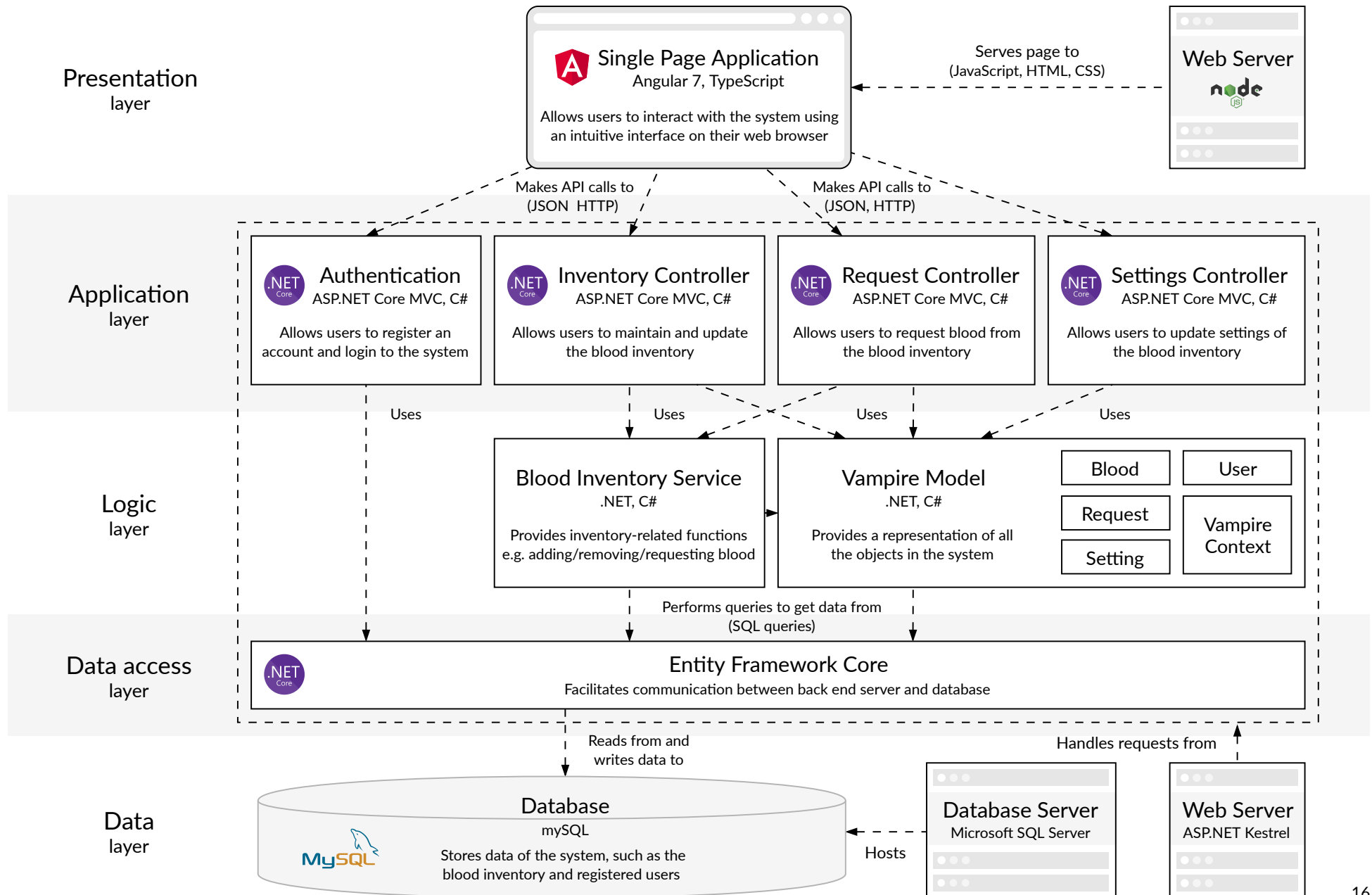
### Request blood component

- Displays a form for requesting blood from inventory, with fields: Blood Type Required, Number of Blood Bags Required, and Destination of Blood Request
- Users can add the entry to a blood “cart”, enabling batch requests of multiple blood types
- Upon submitting, a request is sent to the server which returns a result containing
  - Updated state of blood inventory
  - Blood required to fulfill the request
  - Any alert messages from HQ
- Any alert state fixed by HQ during a request will be displayed in an alert
- A response from the server will be alerted to user
  - Request successful - Blood from inventory assigned to request and the blood will be sent out to the hospital
  - Request unsuccessful - Insufficient blood in inventory to fulfil request and alert sent to Vampire HQ. Currently, the back-end automatically resolves this by adding blood objects into inventory from emergency donors

### Summary Page

- Provides a visual display of the bag count of each blood type in a bar chart
- Contrasts the current supply levels with the expected blood frequency to show the blood types in low supply at a glance
- Provides a visual display of the bag count of blood in a certain age range in a line graph

# Software Architecture





## 6 Back-End Implementation

### Dafny Verification

Our Dafny code is split into several files, each containing the implementation and verification of a core class or algorithm, or blackbox tests. The most notable of those are:

<code>Blood.dfy</code>	Blood class
<code>BloodInventory.dfy</code>	BloodInventory class
<code>TestBloodInventory.dfy</code>	Black-box tests for the BloodInventory class
<code>MergeSort.dfy</code>	Merge sort implementation
<code>Filter.dfy</code>	Filter implementation

We discuss the major components of our Dafny implementation below.

#### Blood Class

The blood class was implemented and verified in `Blood.dfy`. It contains fields for the blood type, the name of the donor, the date of the donation, the location where the blood was donated, and whether the blood has been tested, along with simple getters for reading these fields. For simplicity, the date is represented by an integer, however in the backend real `DateTimes` were used. The blood type is represented by an enumerated data type (defined in `BloodType.dfy`).

```
datatype BloodType = AP | BP | OP | ABP | AM | BM | OM | ABM
```

#### Blood Inventory Class

The blood inventory class was implemented and verified in `BloodInventory.dfy`. It contains two fields: one for storing blood, and the other for the low-supply threshold. According to our priority 1 requirements, the blood inventory needed to support these major operations:

- Querying the amount of blood of a given type
- Determining if a request can be fulfilled
- Fulfilling a request that can be fulfilled
- Removing expired blood
- Depositing tested blood

We initially represented the blood storage by a single array of blood objects.

```
var inv: array<Blood>;
```

Using this representation, we were able to successfully implement and verify some of the required operations as follows:

- **Querying the amount of blood of a given type** - implemented by looping through the array and counting the number of blood objects of the specified type
- **Removing expired blood** - implemented by placing all of the non-expired blood into a new array, and then replacing the inventory with that array
- **Depositing tested blood** - implemented by creating a new array that was identical to the original inventory, except with the new blood appended to the end, and then replacing the inventory with that array

One advantage of using a single array of blood objects was that the class invariant was very simple - it merely required all of the blood objects in the inventory to be non-null and valid. However, there were some disadvantages:

- Mixing together all blood types in a single array made it impossible to verify request fulfilment, as we were unable to keep track of the amount of a specific blood type while satisfying requests for that blood type. Furthermore, we were unable to specify exactly how the blood storage array changed after fulfilling a request, or exactly which blood objects were returned, which made black-box testing impossible.
- It caused some of our algorithms to be overly complex. For example, our request fulfilment algorithm was forced to remove blood objects from the inventory one at a time, as the blood was not organised.

Hence, we decided to change the representation of the blood storage to a map that maps blood types to arrays of blood objects of the same type. In this representation, blood of different types can be thought of as being stored in separate 'buckets'.

```
var inv: map<BloodType, array<Blood>>;
```

Using the map representation of blood storage increased the complexity of the class invariant, but greatly simplified the implementation of the core operations and enabled us to verify operations that we could not verify while using the single-array representation. Furthermore, we were able to fully specify the exact result of each operation (i.e., make our post-conditions as strong as possible), which made blackbox testing possible.

The major operations were now implemented as follows:

- **Querying the amount of blood of a given type** - implemented by returning the size of the bucket (i.e., length of the array) corresponding to the specified blood type
- **Determining if a request can be fulfilled** - implemented by checking if the size of the bucket corresponding to each of the requested blood types is greater than or equal to the amount requested
- **Fulfilling a request that can be fulfilled** - implemented by removing the first  $r_t$  blood objects from the corresponding bucket for each requested blood type  $t$ , where  $r_t$  is the amount requested for blood type  $t$
- **Removing expired blood** - implemented in a similar way as for the single-array representation, but repeating once for each bucket
- **Depositing tested blood** - implemented by adding the new blood object to the bucket corresponding to its blood type

We were also able to implement and verify other accessory operations, such as:

- Querying what blood there is of a given type
- Querying the amount of blood of all types
- Restoring blood supply levels for blood types with low supply
- Automatically restoring blood supply levels after a request, if the request left any blood types with low supply

## Merge Sort

Merge sort was our sorting algorithm of choice, and was used to sort query results in the front-end. We initially implemented and verified the algorithm for an array of integers, and then modified and re-verified the algorithm to make it sort blood objects.

## Filter

Filter was used for filtering query results in the front-end, as well as for removing expired blood in the back-end. Like merge sort, we first implemented and verified the algorithm for an array of integers. We then made the algorithm generic by allowing it to take in an array of any type along with a test predicate.

## Dafny Testing

Even though our algorithms and data structures were verified using Dafny, we still wanted to make sure that our specification of these methods make sense, and that the postcondition of these methods are strong enough for the Dafny verifier to build an abstract model and derive meaningful conclusions from their input.

```
method TestRemoveExpiredBlood1()
{
    var inv := new BloodInventory();
    var amt;

    var blood1 := new Blood(AP, "Amy", 1, "Hospital A", true);
    inv.AddBlood(blood1);
    var blood2 := new Blood(AP, "Bob", 2, "Hospital B", true);
    inv.AddBlood(blood2);

    amt := inv.GetBloodTypeCount(AP);
    assert amt == 2;

    inv.RemoveExpiredBlood(44);
    amt := inv.GetBloodTypeCount(AP);
    assert amt == 1;

    var blood := inv.GetBloodOfType(AP);
    assert blood[..] == [blood2];
}
```

Figure: An example of one of our testing methods.

The above figure shows a typical example of how we tested our Dafny code. We wrote tests to ensure that our postconditions make sense, and behave correctly with regards to blood-specific data.

## Sorting and Filtering to TypeScript

Our sorting and filtering algorithms were implemented and verified in Dafny, and then translated to TypeScript.

Our filtering method starts with a specification for Matches (see figure below). Matches is an inductive specification that takes in a test predicate, and recursively counts the number of elements that pass the given test predicate.

```
function Matches<T>(a: array<T>, end: nat, test: T -> bool): nat
  reads a;
  // ...more preconditions
{
  if end == 0 then
    0
  else if test(a[end - 1]) then
    1 + Matches(a, end - 1, test)
  else
    Matches(a, end - 1, test)
}
```

Figure: The Matches inductive specification. The end parameter is necessary for Matches to be used as a loop invariant.

Based on this inductive definition, we specified VerifyFilter, which returns a sequence of elements that pass the test predicate.

```
function VerifyFilter<T>(a: array<T>, end: nat, test: T -> bool): seq<T>
  reads a;
  // ...more preconditions
{
  if end == 0 then
    []
  else if test(a[end - 1]) then
    VerifyFilter(a, end - 1, test) + [a[end - 1]]
  else
    VerifyFilter(a, end - 1, test)
}
```

Figure: The VerifyFilter inductive specification.

After specifying VerifyFilter, specifying our Filter method was easily accomplished by ensuring equivalence between the two. The actual implementation of Filter uses a loop to determine the number of items that pass the test, and then initialises an output buffer of that size and copies those elements over.

While verifying Filter, we ran into a challenging problem where we had to verify that the variable tracking the current index in the output buffer was not out of bounds. As this index depends on Matches, we were required to make a proof by contradiction using the help of Dafny automatically verifying our inductive properties. Our filtering method was then translated to TypeScript.

```

// Prove that j == count never occurs here.
// General flow of proof:
// j == count
//   ==> j == Matches(a, a.Length, test)
//   ==> Matches(a, i, test) == Matches(a, a.Length, test)
// but i < a.Length
//   ==> Matches(a, i, test) < Matches(a, a.Length, test)
// therefore contradiction and false.
// The key lies in proving the theorem for the contradiction.

// Given from invariant
assert i < a.Length;

// Make the argument that an increase of an additive inductive function
// must have been caused by more induction.
assert forall x: nat, y: nat | x <= a.Length && y <= a.Length
  :: Matches(a, x, test) < Matches(a, y, test) ==> x < y;

// However, the converse cannot establish a strict inequality
// as the differences in elements may evaluate to zero.
assert forall x: nat, y: nat | x <= a.Length && y <= a.Length
  :: x < y ==> Matches(a, x, test) <= Matches(a, y, test);
// Example of what the above theorem implies
assert Matches(a, i, test) <= Matches(a, a.Length, test);

// In order to prove the strict inequality of the converse,
// we will make use of the inductive property of Matches,
// and use it to infer a result about a previous element.
//
// The statement below is a direct derivation of the theorem we've
// already proven above. As this results in a contradiction,
// the antecedent must be false.
assert Matches(a, a.Length, test) < Matches(a, i + 1, test)
  ==> a.Length < i + 1;
// Inverse of the above antecedent
assert Matches(a, i + 1, test) <= Matches(a, a.Length, test);

// Once we know that test(a[i]) is true, we have observed
// a witness that at least one forward element evaluates to 1.
//
// Using this difference, we can establish this strict inequality.
assert test(a[i]) ==> Matches(a, i, test) < Matches(a, i + 1, test);
// We can extend the RHS.
assert test(a[i]) ==> Matches(a, i, test) < Matches(a, a.Length, test);
// RTP.

if test(a[i])
{
  // N.B. Proof obligation P0-1
  assert Matches(a, i, test) < Matches(a, a.Length, test);
  assert j < Matches(a, a.Length, test);
  assert j < count;
  assert j < b.Length;
}

```

## C# Back-End Implementation

Our back-end is implemented in C# using the ASP.NET framework. As it is an MVC framework, our code is separated into; controllers which handle web requests, models which handle data structures and services which handle logic. In our implementation, the functionality is located inside the BloodInventoryService.cs file, most of which is a direct translation of our verified Dafny code. The front-end would send an http request for adding data, retrieving data or removing data. We used a persistent MYSQL express database for the sole purpose of data storage, no queries were performed besides retrieving tables in the form of a list. Whilst carrying out requests we would mirror the impact on the database similar to how the ghost variables mirror the effect on the real data for Dafny verification. This was checked by returning the change in state of the inventory inside the response for each request. We had observed that the database had all the same entries as the changed blood inventory. This allowed us to use a persistent data set for our blood inventory regardless if we restarted the application.

The C# code was almost identical to the dafny code, but slight changes were made due to the difference in syntax, and available ways of handling logic. One key difference is iterating through a map in C# in comparison to Dafny. Figures 1 and 2 show a side by side comparison of a difference in logic between Dafny and C#

```
var bloodTypes = new string[] { "A+", "A-", "B+", "B-", "AB+", "AB-", "O+", "O-" };
var oldBloodArray = await this._bloodInventory.bloodInventory.ToArrayAsync();
var bloodInventoryAsMap = convertArrayToDictionary(oldBloodArray);
foreach (var bloodType in bloodTypes)
{
    bloodInventoryAsMap[bloodType] = GetNonExpiredBlood(bloodInventoryAsMap[bloodType]);
}
```

Figure 1: C# implementation of iterating through a map

```
var types := set t | t in inv;
var typesLeft := types;

while typesLeft != {}
  decreases typesLeft;
  invariant Valid();
  invariant forall t | t in inv && t !in typesLeft :: inv[t][..] == getNonExpiredBloodSeq(old(inv[t]), currentDate);
  invariant forall t | t in inv && t in typesLeft :: inv[t] == old(inv[t]);
  invariant forall t | t in inv && t in typesLeft :: inv[t][..] == old(inv[t][..]);
{
  var t :| t in typesLeft;
  RemoveExpiredBloodForType(t, currentDate);
  typesLeft := typesLeft - {t};
}
```

Figure 2: Dafny implementation of iterating through a map

As can be seen the way of iterating through the blood types is slightly different, but achieve the same result. (note that RemoveExpiredBloodForType and GetNonExpiredBlood are equivalent methods with different naming)

Other differences included return types and data required in the functions. Some functions were required to return extra information, for example the change in state of the inventory, a list of strings containing the result of an alert and so on. These were necessary to return to the front end in order to display an outcome to the user, they were not required in terms of verification and logic, and had no impact on the logic in the backend. An example would be the request method. In C# it had returned information showing the update in state of the blood inventory, the result array of blood from the request, and a list of strings containing all alert messages from HQ. Figure 3 shows the C# function declaration for Request

```
public async Task<Tuple<UpdatedBloodInventoryReturn, Blood[], List<string>>> Request(Request[] batchRequest)...
```

Figure 3: C# Request method declaration

In contrast the Dafny return for Request was just the update in the state of the blood inventory shown in Figure 4.

```
method RequestManyTypesWithLowSupplyFix(req: array<Request>) returns (res: map<BloodType, array<Blood>>)
```

Figure 4: Dafny Request method declaration

In the Appendix Back-End section is an example method comparison between our c# code and Dafny verified code for adding blood into the blood inventory highlighting the similarities between C# logic and dafny logic. All other methods follow a similar style from this example.

Some extra setup was required to pull the state from the database, and mirror changes to the database, but the core logic used in the return is identical to the dafny code, and almost a one to one translation.

```
VampireBackend/  
  Models/  
    DTO's used for JSON serialisation  
  Controllers/  
    Web request parsing etc.  
    Uses BloodInventoryService  
  Services/  
    Most of our actual model code  
    Methods using logic from Dafny
```

Figure: Description of our C# project hierarchy

## Blood Inventory to C#

In Dafny, the blood inventory is represented by a map, where the keys are the different blood types, and the values are arrays of blood objects for the corresponding type. Our database had a table containing blood objects. Using the LINQ library and Entity Framework Core, we were able to pull the table into an array of blood in C#. This required methods to convert from an array of blood objects to a map of blood objects, and vice versa. In C# and our front-end, the blood inventory was simply seen as an array of iterable blood, however when performing our logic and translating our verified code, we were required to convert the array to an equivalent map.

In each method, since we didn't have access to the inventory as a class field, we were always required to pull the data from the database and perform overhead extra setup, creating some minor differences in code between Dafny and C#.

## Where to Find Verified Algorithms

1. SENG2011-Project/VampireBackEnd/VampireBackEnd/Services/BloodInventoryService.cs
2. SENG2011-Project/VampireUi/src/app/Dafny.ts

## 7 Evaluation

### 7.1 Requirements Status Table

Note: For any requirements where we haven't implemented it or its sub-requirements, we have excluded the sub-requirements from the table.

Req No.	Requirement	Priority	Implemented	Verified
<b>1</b>	<b>Querying the system</b>	P1	See sub-requirement(s)	See sub-requirement(s)
1.1	Hospital staff can enquire about the levels of different types of blood in Vampire's inventory.	P1	Yes	See sub-requirement(s)
1.1.1	Hospital staff can see the total amount of each blood type in the inventory	P1	Yes	See sub-requirement(s)
1.1.1.1	Requires an aggregation function that takes a collection of blood bags and produces a collection of results, where each result item consists of a blood type and an amount.	P1	Yes	Yes, but instead of a collection of results, a map of blood type to amount was returned.
1.1.2	Hospital staff can sort results by blood type to see which types are more abundant.	P2	Yes	See sub-requirement(s)
1.1.2.1	Requires a sorting function that sorts blood objects by type.	P2	Yes	Yes
1.1.3	Hospital staff can use a blood type filter to see information relating to particular blood types.	P2	Yes	See sub-requirement(s)



1.1.3.1	Requires a filtering function that takes a collection of blood bags and a set of blood types and produces a collection containing only those blood bags that contain one of the specified types.	P2	Yes	Yes
1.1.4	Hospital staff can view the results of the query in an easy-to-read format (such as a table).	P1	Yes	N/A
1.2	A member of the public (potential donors) may enquire about how, when and where they may give blood.	P4	No	N/A
1.3	A donor may enquire about the outcome of their blood donation.	P4	No	N/A
<b>2</b>	<b>Hospital staff can request blood from Vampire</b>	<b>P1</b>	Yes	See sub-requirement(s)
2.1	The system must be able to check if the batch of requests can be fulfilled.	P1	Yes	See sub-requirement(s)
2.1.1	Requires a function that takes a collection of blood bag objects and a collection of requests and determines if the request can be fulfilled.	P1	Yes	Yes
2.1.1.1	See <i>requirement 1.1.1.1</i>	P1	Yes	Yes
2.2	The system must fulfill requests that can be fulfilled.	P1	Yes	See sub-requirement(s)
2.2.1	Requires a function that fulfills a full batch of requests.	P1	Yes	Yes

2.2.1.1	Requires a function that fulfills a single request, which takes the blood inventory and split it into a collection that is used to fulfill the requests, and the remaining inventory.	P1	Yes	Yes
2.2.2	The system should additionally display a message to the hospital staff to confirm that the request has been fulfilled.	P2	Yes	N/A
2.3	The system must send an alert to Vampire headquarters if a request cannot be fulfilled (see <i>requirement 4</i> ).	P1	Yes	N/A
2.3.1	The system should also inform the hospital staff of the current situation and assure them their request will be handled but may be delayed.	P2	Yes	N/A
2.4	The system should prioritise blood that is closer to expiring to minimise wastage.	P2	No	No
2.4.1	Requires a function that sorts a collection of blood bags by age.	P2	Yes	Yes
2.5	The system should check if the inventory has low blood supply after fulfilling a request (see <i>requirement 4.2</i> ).	P1	Yes	Yes, we wrote a method that automatically restocks any blood type that is low on supply after fulfilling a request
<b>3</b>	<b>Vampire staff can dispose of all expired blood within the inventory.</b>	P1	See sub-requirement(s)	See sub-requirement(s)
3.1	Vampire staff can make a request to the system to remove expired blood from the inventory.	P1	Yes	See sub-requirement(s)

3.1.1	Requires a function to filter a collection of blood bags by age.	P1	Yes	Yes
3.2	The system should check if the inventory has low blood supply after disposing of expired blood (see requirement 4.2).	P1	Yes	Yes, via a method that restocks any blood type that is low on supply after removing expired blood
<b>4</b>	<b>Vampire staff will ensure there is enough blood to satisfy all requests</b>	P1	See sub-requirement(s)	See sub-requirement(s)
4.1	The system must check if the inventory is low on supply for any blood types after performing any operation that removes blood.	P1	Yes	Yes, via a method that restocks any blood type that is low on supply after fulfilling a request
4.1.1	Requires a function that takes a collection of blood bags and determines if there are any blood types that are in low supply. Blood types that are in low supply are added to an 'alert' collection.	P1	No - the logic for this was incorporated into the methods that perform removal operations	No
4.2	The system must automatically alert Vampire HQ if there is a low supply of any blood type.	P1	Yes	Yes
<b>5</b>	<b>Depositing tested blood</b>	P1	See sub-requirement(s)	See sub-requirement(s)
5.1	Pathology staff can deposit clean, tested blood into the inventory	P1	See sub-requirement(s)	See sub-requirement(s)
5.1.1	Pathology staff can enter the details of the tested clean blood into the system	P1	See sub-requirement(s)	See sub-requirement(s)

5.1.1.1	The details must include the donor's name, id, and blood type, and the date and location where the donation occurred	P1	Yes	Yes, via the Blood class
5.1.1.2	Requires a function that takes a blood object and inserts it into the inventory	P1	Yes	Yes
5.1.1.3	Requires the system to store blood test results	P3	No	No
5.2	Pathology staff may enter details of blood tests for unclean blood.	P3	No	No
5.2.1	The details include the donor's name, id, and blood type, the date and location of the donation, and tests that came up positive.	P3	No	No
5.2.2	Requires the system to store blood test results ( <i>see requirement 5.1.1.3</i> ).	P3	No	No
<b>6</b>	<b>All relevant users have an account on the system.</b>	P4	See sub-requirement(s)	N/A (verification is not relevant here)
6.1	Requires a database to store account details, e.g., ids, email addresses, (encrypted) passwords, notifications.	P4	Yes	N/A
6.2	New users are able to create an account on the system	P4	See sub-requirement(s)	N/A
6.2.1	Requires a sign up page.	P4	Yes	N/A
6.2.2	New donors can sign up with just their name and email address.	P4	Yes	N/A

6.2.3	Medical professionals (e.g., hospital staff and Pathology staff) need to provide additional identification such as proof of employment.	P4	No	N/A
6.3	Users can log in to their account with their id and password.	P4	Yes	N/A
6.3.1	Requires a login page.	P4	Yes	N/A
6.3.2	Requires a server to authenticate users and a database (see <i>requirement 6.1</i> ).	P4	Yes	N/A
6.4	Every user has a profile and can update their personal details, such as their phone number and password.	P4	See sub-requirement(s)	N/A
6.4.1	Requires a profile page and options to update and save details.	P4	No	N/A
6.4.2	Requires a database to store and update users' details	P4	Yes	N/A
6.5	When signed in, a user can perform the actions they are authorised to do.	P4	Yes	N/A
6.5.1	Requires a menu to enable users to select what action they want to perform.	P4	Yes	N/A
6.5.2	Requires pages for each action (e.g., querying the inventory, submitting blood test details) and a server to handle actions.	P4	Yes	N/A

## 7.2 Teamwork

- **Abanob Tawfik**, z5075490: Requirements, WBS, C# Backend code, linking frontend to backend, Dafny verification on blood inventory
- **Kevin Luxa**, z5074984: Requirements, Use Cases, C# Backend code, Dafny merge sort and blood inventory
- **Lucas Pok**, z5122535: WBS, Dafny filter and query, front-end query and summary
- **Rason Chia**, z5084566: Front-end & front-end linking, Use Cases
- **Michael Yoo**, z5165635: Dafny sort integration, elaboration and explanation of Dafny algorithms, testing of Dafny code, report writing

## 7.3 Review

Working together as a team to produce our final take on the Vampire system has taught us many lessons. There are many tasks we would have done differently, tasks we might have liked to accomplished and approaches we would have liked to have taken.

Initially we began verification naively on an array of blood and simply just verified code till it worked. This had proven to be problematic when we attempted to create black box tests for our methods. We then decided to use a map over an array allowing for more complicated verification. Ideally we would have liked to have planned our approach more carefully and ask more questions along the way. As we had decided to perform this change after we had implemented our backend in C#, this led to a major refactor requiring a full rewrite of the C#. In future we will make sure to carefully plan and design our approach and look at the end picture, rather than hacking at the problem till its solved.

As a group we all learned how to better communicate between each other, and delegate tasks. Before we began working on implementation and verification we would only discuss in group meetings what we will do and complete tasks based off memory and perception of what was discussed. Later on in the project we setup a Trello board containing tasks organised into their categories, and assigned tasks to group members. This had greatly helped us keep focus on what our main goals were and what we had to do. It had also stopped us from spending too much time on a feature that wasn't contributing to that goal.

Whilst working on a big project, it is quite easy to lose track of what was initially set out in the requirements. One big learning point for all of us was looking back at the requirements. When we first began verifying our methods we forgot what requirements we set out, and went off what we thought the system had to do, often leaving out key requirements. As a group we had learnt to constantly re-read our requirements and fully understand what we are building. Once we had an idea of what had to be done, we set out tasks to accomplish those requirements as quickly (and correctly) as possible. Something we could have done which would have been a great help, is conducting our mentor meetings based on requirements. We often went in with what we thought the system should do, and would have benefited greatly from re-reading our requirements before our meeting to know exactly what to discuss.

We were able to complete many of our requirements, and produce a system which encapsulated all of our priority 1 requirements. With more time we would have liked to include more stakeholders, particularly the pathology user to test the blood. Our assumption was that all blood deposited to the inventory was tested, but we would have liked to have verified algorithms for testing blood. Given more time we would have also liked to have improved the user interface, for example using ng toaster over JavaScript alerts. Overall however, we are happy with the final outcome of our Vampire system.

## 8 Appendices

### 8.1 Appendix Back-End

C#:

```
public async Task<UpdatedBloodInventoryReturn> AddBlood(Blood blood)
{
    if (_bloodInventory != null)
    {
        // dafny logic
        var oldBloodArray = await this._bloodInventory.bloodInventory.ToArrayAsync();
        var bloodInventoryAsMap = convertArrayToDictionary(oldBloodArray);
        var bloodType = blood.bloodType;
        var newBucket = new Blood[bloodInventoryAsMap[bloodType].Length + 1];
        for (var i = 0; i < bloodInventoryAsMap[bloodType].Length; i++)
        {
            newBucket[i] = bloodInventoryAsMap[bloodType][i];
        }
        newBucket[bloodInventoryAsMap[bloodType].Length] = blood;
        bloodInventoryAsMap[bloodType] = newBucket;
        var addedToInventory = ConvertDictionaryToArray(bloodInventoryAsMap);
        await this._bloodInventory.bloodInventory.AddAsync(blood);
        await this._bloodInventory.SaveChangesAsync();
        return new UpdatedBloodInventoryReturn()
        {
            oldBloodInventory = oldBloodArray,
            newBloodInventory = addedToInventory
        };
    }
    return null;
}
```

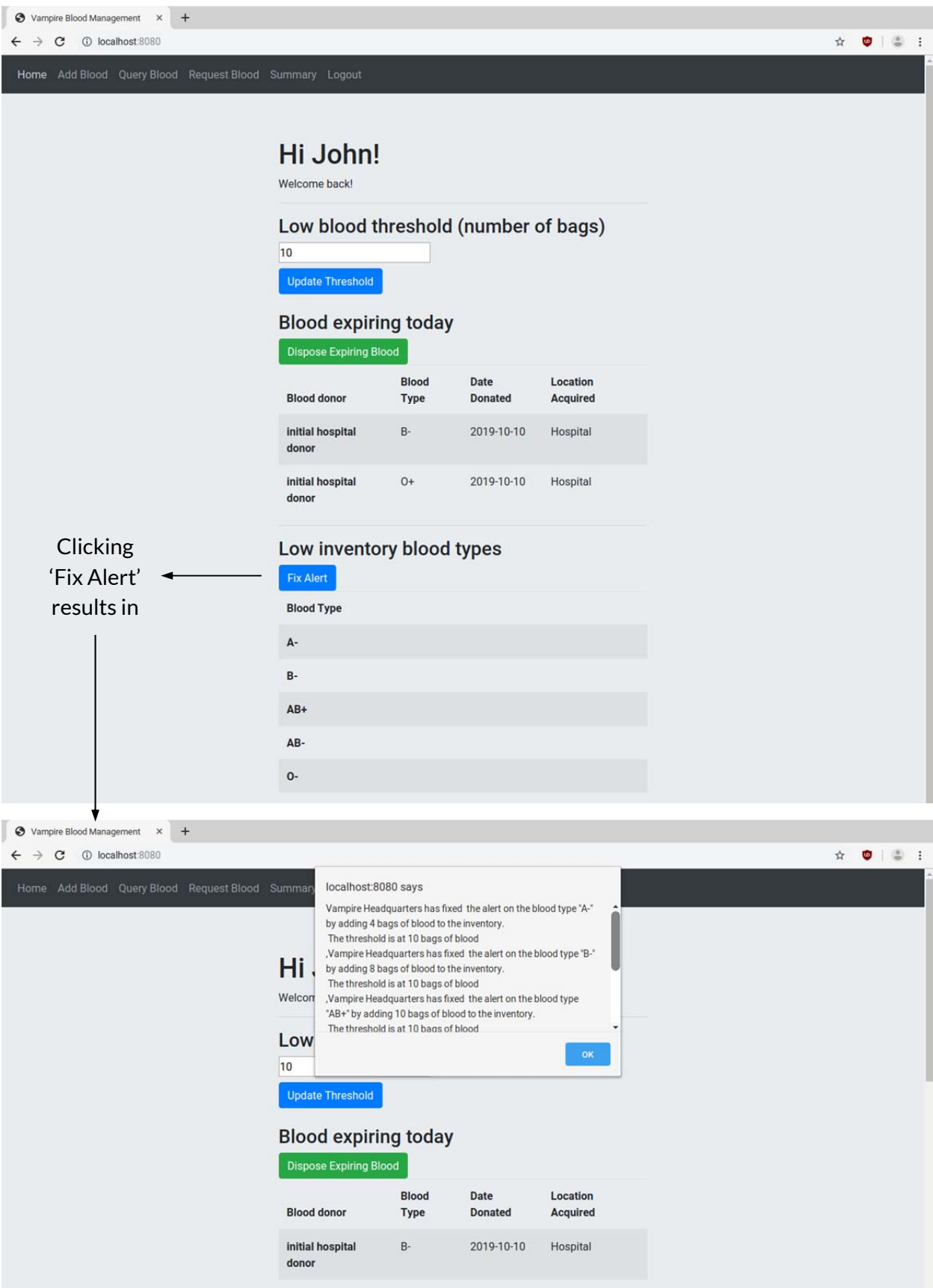
Dafny:

```
method AddBlood(blood: Blood)
  modifies this`inv;
  requires Valid();
  requires blood != null;
  requires blood.Valid();
  ensures Valid();
  ensures inv[blood.GetBloodType()][..] == old(inv[blood.GetBloodType()][..]) + [blood];
  ensures fresh(inv[blood.GetBloodType()]);
  // ensure other blood buckets are unchanged
  ensures forall t | t in inv && t != blood.GetBloodType() : inv[t] == old(inv[t]);
  ensures forall t | t in inv && t != blood.GetBloodType() : inv[t].Length == old(inv[t].Length);
  ensures forall t | t in inv && t != blood.GetBloodType() : inv[t][..] == old(inv[t][..]);
{
    var t := blood.GetBloodType();

    var newBucket := new Blood[inv[t].Length + 1];
    forall i | 0 <= i < inv[t].Length
    {
        newBucket[i] := inv[t][i];
    }
    newBucket[inv[t].Length] := blood;
    inv := inv[t := newBucket];
}
```

# 8.2 Appendix Front-End

## Home Page and Fix Alert response





## Add blood page

[Home](#) [Add Blood](#) [Query Blood](#) [Request Blood](#) [Summary](#) [Logout](#)

### Add Blood

Donor Name

Select Blood Type

Blood Status

Date Donated

Location

## Adding blood to inventory response

[Home](#) [Add Blood](#) [Query Blood](#) [Request Blood](#) [Summary](#)

localhost:8080 says  
Added Blood to Inventory.

### Add Blood


Donor Name

Select Blood Type

Blood Status

Date Donated

Location



## Query blood page

[Home](#) [Add Blood](#) [Query Blood](#) [Request Blood](#) [Summary](#) [Logout](#)

### Query Blood

Filter blood by

Type ▼ Age ▼ 100 results

Reset ×

Sort blood by

None ▼ ▲

Type	Status	From	Donated on	Age (days)
O+	Tested	Hospital	2019-11-16	6
A+	Tested	Hospital	2019-10-31	22
O+	Tested	Hospital	2019-10-24	29
A+	Tested	Hospital	2019-11-12	10
B+	Tested	Hospital	2019-10-28	25
O+	Tested	Hospital	2019-10-16	37
O-	Tested	Hospital	2019-11-01	21
A+	Tested	Hospital	2019-11-05	17
O+	Tested	Hospital	2019-10-18	35
O+	Tested	Hospital	2019-11-10	12

« « 1 2 3 4 5 » »

10 items per page ▾

## Filtering blood by A+ type

[Home](#) [Add Blood](#) [Query Blood](#) [Request Blood](#) [Summary](#) [Logout](#)

### Query Blood

Filter blood by

Type ▼ Age ▼ 30 results

Reset ×

Sort blood by

None ▼ ▲

A B AB O + -

Donated on

Age (days)

A+	Tested	Hospital	2019-10-31	22
A+	Tested	Hospital	2019-11-12	10
A+	Tested	Hospital	2019-11-05	17
A+	Tested	Hospital	2019-11-06	16
A+	Tested	Hospital	2019-11-05	17
A+	Tested	Hospital	2019-10-20	33
A+	Tested	Hospital	2019-10-19	34
A+	Tested	Hospital	2019-10-17	36
A+	Tested	Hospital	2019-10-24	29
A+	Tested	Hospital	2019-11-01	21

« « 1 2 3 » »

10 items per page ▾

## Filtering blood by A+ blood type, sorted by increasing age

[Home](#) [Add Blood](#) [Query Blood](#) [Request Blood](#) [Summary](#) [Logout](#)

### Query Blood

Filter blood by

Type ▾ Age ▾ 30 results Reset ×

Sort blood by

Age ▾ ▲

Type	Status	From	Donated on	Blood Age
A+	Tested	Hospital	2019-11-15	/
A+	Tested	Hospital	2019-11-14	8
A+	Tested	Hospital	2019-11-12	10
A+	Tested	Hospital	2019-11-12	10
A+	Tested	Hospital	2019-11-09	13
A+	Tested	Hospital	2019-11-06	16
A+	Tested	Hospital	2019-11-05	17
A+	Tested	Hospital	2019-11-05	17
A+	Tested	Hospital	2019-11-05	17
A+	Tested	Hospital	2019-11-04	18

« « 1 2 3 » »

10 items per page ▾

## Filtering blood by A+ blood type and age range, sorted by increasing age

[Home](#) [Add Blood](#) [Query Blood](#) [Request Blood](#) [Summary](#) [Logout](#)

### Query Blood

Filter blood by

Type ▾ Age ▾ 4 results Reset ×

Sort blood by

Age ▾ ▲

Type	Age	From	Donated on	Age (days)
A+	Tested	Hospital	2019-11-15	7
A+	Tested	Hospital	2019-11-14	8
A+	Tested	Hospital	2019-11-12	10
A+	Tested	Hospital	2019-11-12	10

« « 1 » »

10 items per page ▾

## Request blood page

[Home](#) [Add Blood](#) [Query Blood](#) [Request Blood](#) [Summary](#) [Logout](#)

### Request Blood Page

To be Delivered To:

St George Hospital

Number of Bags Required (1 Bag = 285 ml)

2

Select Blood Type

A+

Add Blood Entry

Requests:

Blood Destination	Blood Type	Quantity Required
Submit Blood Requests		

## Standard request response

[Home](#) [Add Blood](#) [Query Blood](#) [Request Blood](#) [Summary](#)

localhost:8080 says  
Requests Submitted.  
OK

### Request Blood Page

To be Delivered To:

Number of Bags Required (1 Bag = 285 ml)

Bags Required

Select Blood Type

Add Blood Entry

Requests:

Blood Destination	Blood Type	Quantity Required
St George Hospital	A+	2
Alice Springs Hospital	O+	7

Submit Blood Requests

## Fixing alert in request response

localhost:8080 says

Vampire Headquarters has fixed the alert on the blood type "A-" by adding 5 bags of blood to the inventory.  
The threshold is at 10 bags of blood

Vampire Headquarters has fixed the alert on the blood type "B-" by adding 5 bags of blood to the inventory.  
The threshold is at 10 bags of blood

OK

Number of Bags Required (1 Bag = 285 ml)

Bags Required

Select Blood Type

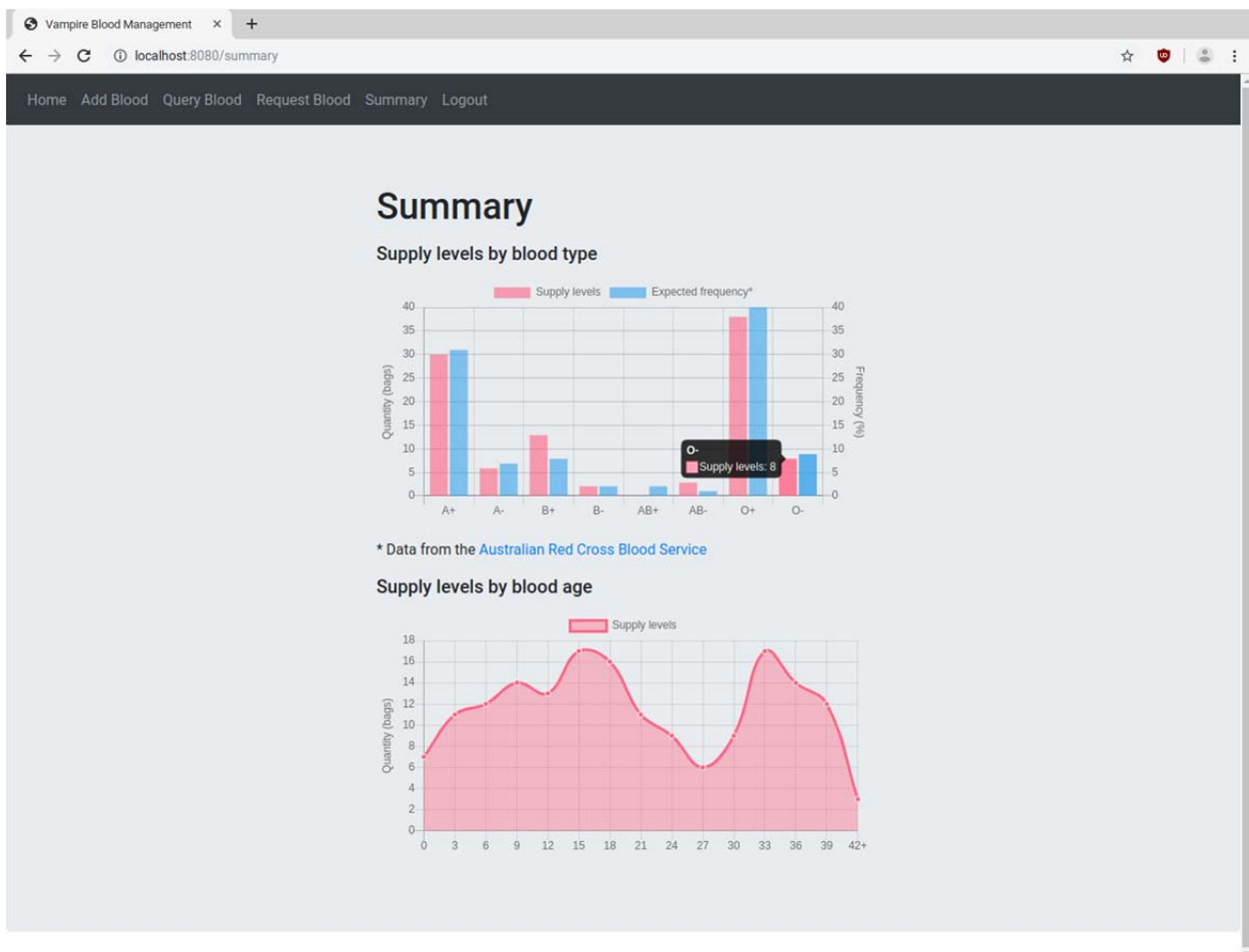
Add Blood Entry

Requests:

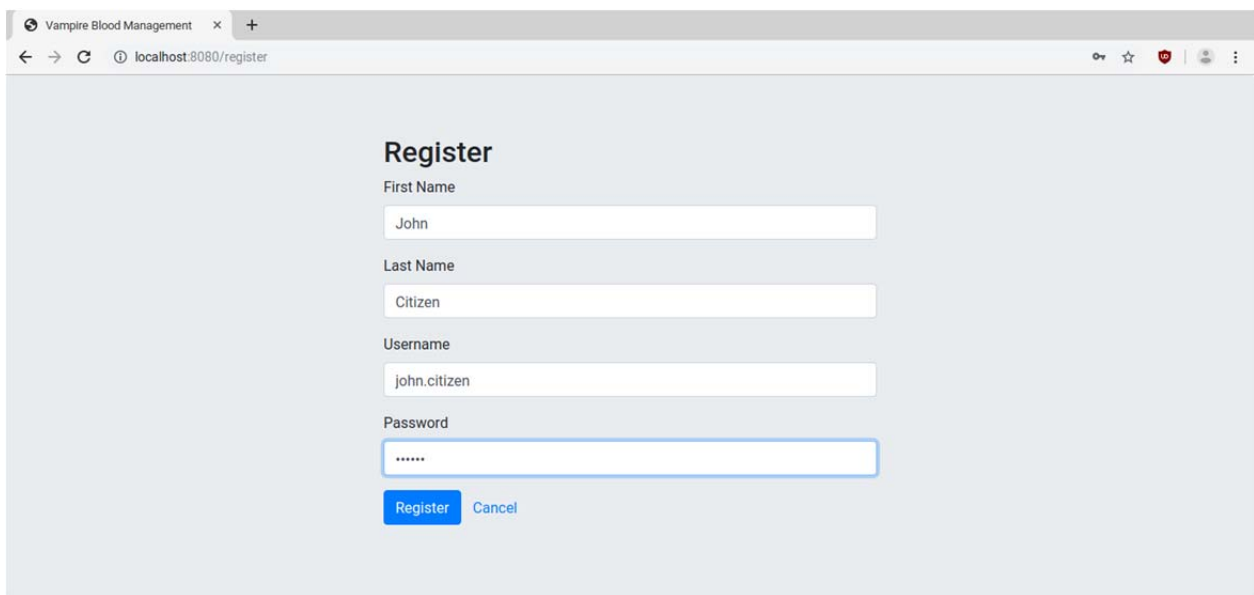
Blood Destination	Blood Type	Quantity Required
ABC Hospital	A-	5
Westmead Hospital	B-	5

Submit Blood Requests

## Summary page



## Register page



A screenshot of a web browser window showing the 'Register' page of a 'Vampire Blood Management' application. The browser's address bar shows 'localhost:8080/register'. The page has a light blue background. The 'Register' title is centered. Below it are four input fields: 'First Name' with the value 'John', 'Last Name' with the value 'Citizen', 'Username' with the value 'john.citizen', and 'Password' with masked characters '\*\*\*\*\*'. At the bottom are two buttons: a blue 'Register' button and a blue 'Cancel' button.

Vampire Blood Management x +

localhost:8080/register

### Register

First Name

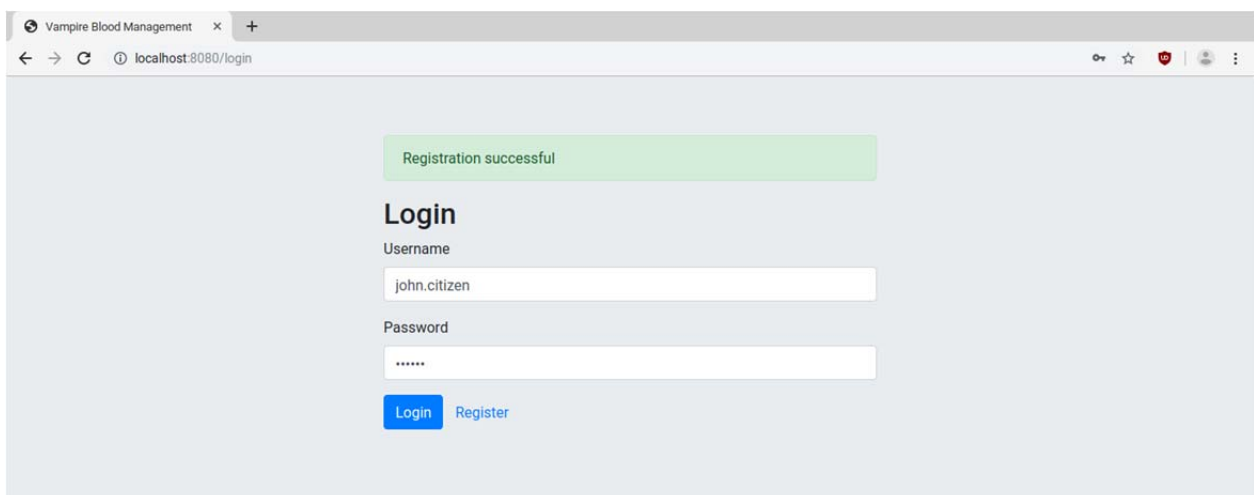
Last Name

Username

Password

Register Cancel

## Login page



A screenshot of a web browser window showing the 'Login' page of a 'Vampire Blood Management' application. The browser's address bar shows 'localhost:8080/login'. At the top, a green message box says 'Registration successful'. Below it is the 'Login' title. There are two input fields: 'Username' with the value 'john.citizen' and 'Password' with masked characters '\*\*\*\*\*'. At the bottom are two buttons: a blue 'Login' button and a blue 'Register' button.

Vampire Blood Management x +

localhost:8080/login

Registration successful

### Login

Username

Password

Login Register