Database Design

The database has 2 entities: a Patient table, and a Request table.

The Patient table has the attributes: PatientID, FirstName, Surname, DOB, PhoneNumber, Notes, Symptoms.

The Request table has the attributes: RequestID, PatientID, UserID, Items, Date.

There is a one-to-many relationship from the patient table to the request table. A Patient can have a minimum of 0 requests, and a maximum of many requests. A Request must have a minimum of 1, and a maximum of 1 patient.

The PatientID in the Request table, is a foreign key from the patient table. The UserID is a foreign key from the User Authentication API. The RequestID, PatientID, and UserID are composite keys making up the primary key of the Request table. All 3 of the attributes must be present for a valid request.

Needs to be developed:

Adding an Item entity along with a RequestLine entity into the database would allow for single requests to contain multiple items. The Item entity contains detailed information like item ID, item name, dosage, quantity on hand, etc. The RequestLine entity is like an order line on a receipt, and would link the Request to a specific item, thus enabling the Request to contain multiple Items along with quantity or dosage information. The RequestLine entity is a composite entity whose primary key is comprised of the RequestID and ItemID.

Adding a Location entity, that would contain LocationID, Name, AmountOfPatients, AmountOfUsers, etc. The LocationID could be a foreign key in most other entities. This would add expandability to the project, allowing for different locations to join the project with minimal effort on their part, and likewise expand the scope of available resources, for both patients and users.

Inventory API

Implementing an inventory system (e.g. openMRS) so Supplier can enter their inventory (items, quantity, locations, etc.) into the database that would be accessible to Health Providers through the Web and phone UI, who would then be able to check the requested item's availability in real time, and respond to the patient as quickly as possible. This could be implemented into the UI by adding a detailed table of items that the Provider queries.

User Interface

There is a web portal to login to the application. There are 3 options available to users for login: Patient, Health Provider, and Admin. Patients can enter their information and their symptoms. Health Providers can login and see pending requests as well as communicate with each other. The pending requests have a email address that the health provider can message to inform the patient what location they need to go to in order to receive their medication. Once this message is sent out their request is removed from the request list.

Expandability

We assume there will be a central repository that would link smaller, distributed databases in varying locations. For this to happen, each piece of data (e.g. items, users, patients) would contain a LocationID from the Location entity that would link that piece of data to a certain location.

Authentication

An admin handles user management: adding users, disabling users, removing users. A user (health provider) is able to change and recover their password. A patient does not need to authenticate.

Needs to be developed:

The super admin manages the whole site, and creates admin accounts to manage each sub-site. This way multiple sites can use a single system.

Input API

The user inputs their name, location, e-mail address, and their supply request into the webpage. The webpage maintains a queue of these inputs in the form of data rows. Using this information, the nurse/admin will be able to check inventory for availability according to the received data.

Needs to be developed:

For the user end, a dropdown box that would allow the user to select an item instead of input their own, along with a separate quantity/amount box (if applicable to item).

Once the database design is improved on, users will be able to make multiple item requests instead of one request per item.

Output API

The nurse/admin can then read these data rows and see what supplies are needed by a person using a specific email address. They can then check for availability of the item and respond back to the requestor with a location where the item is ready for pick up.

What needs to be developed:

linking in SMS API

Currently MedLink uses email alerts that are sent out to the hospital administrator, but has no way of communicating back to the patient that their medicines have arrived at the hospital. Since the patients are giving us their phone numbers, we need a way to communicate with them using this phone. SMS is the most viable option, since they can be automated and sent out by the system. Here are the steps to integrate an SMS API into the app.

1) Identify an SMS API.

There are a few options online, but you will need to check whether they support the Gambian networks (you may end up having to use more than one API, to get support for all the networks (Africell, Gamcell, etc.). Another way to do this is, instead of an API, is using an email to SMS gateway (Africell used to have one, letting you send an email to 787992@africell.gm to have it forwarded as a text message). You may also need to use a mix of methods (e.g. the SMS to email gateway for Africell, and an SMS API for QCell and Gamcell. You'll need to do research to find out how to tackle each network.

2a) Create an SMSNotification class

You can save it under app/controllers/mailers. It should contain a generic "sendSMS" method, which takes in a phone number and a message and sends out the message as an SMS.

2b) The sendSMS method

As stated above this takes a phone number and a message as its arguments. It takes the first digit of the phone number and uses it to decide which network the phone number belongs to. Once it's decided what network it should be sent to, it then forwards it to one of the following methods (all defined in the SMSNotification class):

sendToGamcell - this sends out to Gamcell network
sendToAfricell
sendToQcell
sendToComium

Separating each network out into its own function allows us to use different methods for each network, and if one of them changes we only have to change the corresponding method. In the actual functions you can then decide to use an SMS API or email-to-SMS gateway or anything else that the network uses.

3) Adding “Notify User” to app

Eventually we want to have the notifications sent out automatically when the meds come in, but to start with you can make them manual (i.e. the nurse has to click a button to send them out to the patient). On the requests page (which you can get to using the Nurses link in the top nav bar) we have a list of requests. Add a button to this called “Notify Patient” (you can do this by editing the file “medlink/app/views/requests/index.html.erb”.

The “Notify Patient” button should be a Rails “button_to” which will call a function (sendNotification) in the requests controller to send out the actual SMS. The “button_to” should also include the selected request’s ID. This is for the frontend.

On the backend you will need a sendNotification method. This can be defined in app/controllers/requests_controller.rb, and will be a simple method that just uses the request ID to get the associated patient and their phone number. It then sends out this notification by using the SMSNotification class that we created earlier.