## Deployment

Summary:

In order to host our project's services, we decided to use Carolina Cloud Apps.

Problems:

We need somewhere to host the backend API, the frontend, and the persistent database. We need something that is easy to deploy too and that requires minimal management from us. Ideally, we should be able to push a git commit and have the deployment automatically update.

Constraints:

We don't want to be running our own servers because that requires specialized knowledge and we assume the client won't be able to handle doing it once we finish this class. We want something that is reasonably cheap, and we assume the client would prefer the cheapest option possible.

Options:

- AWS:
  - Pros:
    - Team member experience
    - Stable
    - Well documented
    - Continuous Deployment
  - Cons:
    - 3rd party
    - Costs money
- Carolina Cloud Apps:
  - Pros:
    - Free
    - UNC runs it which is a plus since our project is in house
    - Continuous Deployment
  - Cons:
    - Poor documentation
    - New to all of us
    - Experiences occasional outages

Rationale:

We chose Carolina Cloud Apps to host our system deployment. We did this because it is free and in house at UNC. We think that the hospital will have better luck working with UNC IT than having to deal with AWS if something goes wrong. It being free also means that we don't have to deal with getting money from the client which seems like it could be a headache.

## Backend API

Summary:

In order to create a backend API, we decided to use the Python framework Flask.

Problem:

We need a backend service that accepts API calls and interacts with our data storage. It also needs to support sending emails and generating pdfs.

Constraints:

Our backend needs to support several data storage options concurrently. It also needs to easily support a restful API.

Options:

- Python Django:
  - Pros:
    - Carlos and Alice both know Python
    - Alice prefers Python to javascript
    - Alice has experience with Django
  - Cons:
    - Lots of boilerplate code
    - Overkill for a restful API
- Python Flask:
  - Pros:
    - Carlos and Alice both know Python
    - Alice prefers Python to javascript
    - Simple
    - Well suited for restful API
  - Cons:
    - No one on the team has worked with it before
- Node JS:
  - Pros:
    - One language for whole project
    - Well supported
    - Carlos is familiar
  - Cons:
    - Alice doesn't like JavaScript

Rationale:

We chose Python as our backend programming language because Alice is the one primarily writing the backend code. We went with Flask over Django because it seems simpler and better suited for building a restful API.

## Frontend

Summary:

In order to easily and effectively create a responsive user-friendly front-end, we decided to use React.

Problem:

We need a dynamic front-end that makes calls to our backend API. The webpage itself should handle as little as possible and offload the work to the backend.

Constraints:

The front-end needs to be responsive because we assume the clients will use it on their phones occasionally, but our primary target is desktop. The webpage needs to be simple and easy to use so that people can learn how to use it quickly.

Options:

- Pure Javascript
  - Pros:
    - Lightweight
  - Cons:
    - Everything has to be written from scratch.
    - Makes front-end development much harder.
- React:
  - Pros:
    - Easy to use
    - Our team, especially tom, is familiar with it.
    - Simplifies the development process
  - Cons:
    - Adds a 3rd party dependency.

Rationale:

React is the obvious choice for the front-end because it simplifies the process of development substantially compared to pure javascript. It also is the web framework that our team is most familiar with by a long shot.

**Data Storage**

Summary:

In order to store persistent user data, we decided to use MongoDB.

Problem:

We need to persistently store form data as well as user accounts. People need to be able to log in to the site and have forms associated with their accounts. They also need to be able to access previously filled out forms and fill out new forms. They also may need to create new forms. This means that we need to store data persistently.

Constraints:

Data needs to be persistent and must be backed up.

Options:

- SQL-based DB:
  - Pros:
    - Powerful
  - Cons:
    - Lots of complexity that our project doesn't need.
- MongoDB:
  - Pros:
    - Easy integration with Python
    - Easy to use Schemas
    - No SQL
  - Cons:
    - Less powerful then SQL

Rationale:

We decided to use MongoDB for storing persistent information because of its ease of use and ease of integration with python. We don't need the relational tools provided by SQL and as such we don't think the added complexity is worth dealing with.

**Data Backup**

Summary:

In order to ensure access to form data at all times, we (are currently still deciding on an archival method to use).

Problem:

The form data needs to be accessible at all times. Loss of data can not be allowed. This means that the database needs to be backed up across multiple providers in multiple physical locations.

Constraints:

Data must be accessible at the time of inspection, we assume this means within a few minutes of the inspector asking to see it.

Options:

- Amazon Glaciar:
  - Pros:
    - Stable
    - Cheap (to store)
  - Cons:
    - Expensive (to retrieve)
    - 3rd party dependency
- Backup mongoDB cluster:
  - Pros:
    - Easy to use if main DB goes down
  - Cons:
    - Needs to be hosted somewhere other than Carolina Cloud Apps
    - Requires Management
    - Overkill
- Paper:
  - Pros:
    - The only things that will make the data inaccessible are likely to make the whole lab inaccessible
    - Mimics current data storage practices of the lab
    - Easy to implement
  - Cons:
    - Requires someone to print things
    - Takes up lots of space
    - Hard to manage
    - Hard to recover computer database from paper

Rationale:

We are still investigating our options for archiving the data in a fault-tolerant manner, but we are thinking of using something like Amazon Glacier or another cloud storage provider. We

are also likely going to suggest that the lab prints out paper backups regularly in order to provide an absolute emergency form of access.

## SMTP

We don't have a write-up for this because we haven't made a decision yet, we also don't have enough information to even list our options. We need to talk to the hospital IT department before we can figure out how to proceed. All we know is that we need an SMTP server to send emails for notifications. Thus we included it in our Architecture Diagram.