PyMedServer: A Server Framework for Mobile Data Collection and Machine Learning

by

John Mofor Nkaze

S.B., C.S. M.I.T., 2015

Submitted to the
Department of Electrical Engineering and Computer Science
in Partial Fulfillment of the Requirements for the Degree of Master of Engineering in Electrical Engineering and Computer Science

at the

Massachusetts Institute of Technology

September 2019

Author: Signature redacted

Department of Electrical Engineering and Computer Science
June 25, 2019

Certified by: Signature redacted

Richard R. Fletcher, Research Scientist - *Thesis Supervisor*

Accepted by: Signature redacted

Katrina LaCurts, Chair, Master of Engineering Thesis Committee

PyMedServer: A Server Framework for Mobile Data Collection and Machine Learning

by

John Mofor Nkaze

Submitted to the Department of Electrical Engineering and Computer Science on June 28th, 2019, in Partial Fulfillment of the Requirements for the Degree of Master of Engineering in Electrical Engineering and Computer Science

Abstract

Over the past decade, the widespread adoption of smart phones has enabled their use as a primary data collection platform for a wide variety of research studies, including areas such as global health field research. In addition, smart phones also provide a portable means of collecting labelled data for use in Machine Learning and Artificial Intelligence. Despite these important global trends, existing scalable mobile data collection platforms are not available for use with machine learning data collection. In order to address this need, I have created PyMedServer, which is an easy-to-use server framework designed for large scale medical research spanning multiple distinct institutions. The framework provides built-in abstractions for clinicians, patients, medical measurements, clinician diagnoses, and machine learning analyses. To further facilitate adoption, PyMedServer provides client libraries for Android and Web. These libraries are compatible with any server built using the PyMedServer framework and provide features such as Local Storage, API integration, User Authentication and Authorization, Multi-Group Support and Measurement Labelling, to name a few. PyMedServer is written in Python and is designed to permit and facilitate the addition of plugins. Developers are granted access to labeled data and can contribute with Feature Extraction and Machine Learning plugins, while the framework takes care of concerns such as of Group Isolation, Security, Scalability, and Deployability.

Thesis Supervisor: Richard R. Fletcher

Title: Research Scientist

ACKNOWLEDGEMENTS

I thank my research advisor, Rich Fletcher, for his continuous support and guidance throughout these last five years. For this year in particular, he went to extra-ordinary lengths to ensure that I was all set for my next step in life after graduation.

I would also like to thank the MIT TATA Center for its generosity sponsoring me while I pursued this degree. I will also like to thank TATA for sponsoring my two research field trips to India. Special thanks to Angeliki Diane for making my experience as a TATA fellow a treat. Special thanks to Dr. Robert James Stoner, Dr. Chintan Vaishnav and Dr. Jason Prapas for the fantastic Seminar courses. Finally, special thanks to the MIT TATA staff who help run the program.

I want extend my gratitude to the Department of Electrical Engineering and Computer Science here at MIT. Special thanks to my instructors and the administrators.

Lastly, I will like to thank my friends and family for their continued support throughout these years. Special thanks to my parents back home in Cameroon - Mr. Mofor Edward and Dr. Mofor Clautilde for their relentless guidance and support throughout these years. I pray God grants you both many more years, and that your children keep making you proud!

TABLE OF CONTENTS

TABLE OF FIGURES

1 INTRODUCTION

The world has seen an increase in the adoption of cloud technologies for medical analyses, as well as a widespread growth in the smartphone space. The cloud allows for more effective medical information collection, processing, and analyses, while smartphones make these capabilities ever more accessible to the end users - be it medical personnel, patients, or any other intermediaries. This document describes the MIT Mobile Technology Lab's answer to best leverage these ever-increasing trends.

This chapter describes the background for this project and motivates why PyMedServer was built. Some existing solutions will be presented, as well as the gaps in those solutions which led to the creation of PyMedServer. In the next chapter, PyMedServer's architecture is described in depth - with special emphasis on how it more closely fits our group's needs.

1.1 BACKGROUND AND MOTIVATION

I am a Research Assistant at MIT's Mobile Technology Lab. Our group specializes in the medical space and strives to incorporate cutting edge Artificial Intelligence and Machine Learning techniques into our projects to aid with a variety of applications, mainly medical screening, and personal health. In this section, we enumerate some of the core requirements which motivated us to build the PyMedServer Framework.

1.1.1 Data Diversity

The Mobile Technology Lab is in a unique position where it runs several medical research projects concurrently and needs to be able to spin up new projects within relatively short timelines - typically one to two months. At the point of writing this document, our group is actively running over seven different research projects across the world. Across our projects arose a common need - the need to build a scalable, secure, and effective data collection, labeling and

analysis pipeline. For example, the analysis generated from our various algorithms is used to assist medical staff in their disease screening efforts. At the base of this pipeline, needed to rest a robust data ingestion phase.

1.1.2 Mobile Client Support

When it came to the choice of devices to support our data ingestion, multiple factors had to be considered:

- How much does the equipment cost?
- Who are our users and what devices would they be able to operate?
- How available is the equipment in the target areas?

Smartphones stood out as the best option for multiple reasons. They are relatively cheaper than laptops, come in with built-in input methods such as cameras and are powerful enough to meet the needs of our platform for data gathering and pre-processing purposes. In addition, in our target regions, which tend to be underdeveloped, smartphones are more available and omnipresent than laptops or desktop computers. Hence smartphones were chosen as our primary means of data collection and pre-validation.

For most of our projects, trained medical professionals and community health workers use smartphones to record patient measurements. These measurements could be images - such as retina images captured with the smartphone's camera, sound recordings - such as lung sounds from patients, survey answers - such as questionnaires taken by patients, and the list goes on. As a result, we needed a platform that could handle a vast array of data types.

1.1.3 Support for offline Use

Because our target population is predominantly afflicted groups in remote areas, it is important that our solution works regardless of the state of internet connectivity on the field. In most cases, the medical staff have access to the internet at their hospitals and clinics but have little to no internet connectivity on

the field when they head out to visit the patients. While the patient visits are taking place, it is crucial that data integrity is upheld - that is, patient data should be correctly associated with the right patient, and be stored locally such that they can be synchronized with a server once internet connectivity is re-established.

1.1.4 Support for Machine Learning and Data Labelling

Labeling is an important cornerstone for machine learning groups like ours because the quality and size of the training data available to our algorithms determine their classification performance on unseen measurements. In our case, in particular, we prefer not to utilize large scale labeling services like Amazon's Mturk mainly due to the complexity and specializations required to properly label medical data, as well as the various constraints that need to be taken into consideration when processing and storing sensitive medical data. In addition, we needed to support use cases wherein the labeling could be carried out offline, and later synchronized with a secure centralized repository in the cloud. Many of the existing solutions we found did not offer labeling capabilities, let alone offline labeling support.

1.1.5 Support for Multi-Group Research

We needed a platform that could correctly isolate separate studies (study realms) while at the same time sharing software tools and resources. We define a study realm as a set of clinicians, administrators, and patients belonging to a study at a particular period in time. All measurements need to be tagged with a study identifier, such that, although a given patient could be part of multiple studies, we always retain the ability to differentiate between measurements taken as part of separate studies.

Group management, as a result, was an important requirement for us. Clinicians should be able to enroll new patients into any of their groups, as they see fit. In addition, this enrollment process must be available offline, such that

clinicians can register new patients while they are on the field when there is limited or no internet access.

1.2 EXISTING SOLUTIONS

At the start of my graduate research, I first reviewed existing Electronic Medical Record (EMR) systems which support online-offline synchronization of all medical artifacts, including but not limited to, patient metadata such as group membership or profile information, patient measurements, clinician diagnoses, and machine analyses. It also included popular platforms that are currently used by top academic research groups that use mobile phone tools for data collection. In this section, we list some of our findings, and for each, we accentuate on the gaps which steered us towards our custom solution.

1.2.1 Open Data Kit (ODK)

Open Data Kit [1] is a set of tools that allows people around the world to collect, manage, and use their data. ODK is very popular amongst the medical community thanks to its ease of use and deployment.

Pros:

- Simple to use - it is very easy to get started with ODK as it does not require any programming experience at all.
- Simple to deploy - In just a couple of clicks, it is possible to deploy a fully operational ODK set.
- Large community - ODK has a large and vibrant community of users and developers. This makes their proposition quite attractive for long term projects like ours.

Cons:

- Complexity
    - ODK is very end-user centric. Everything from deployment to development is centered on giving a great user experience to the end users, at the expense of the complexity of the internal systems.

- Not Plugin Friendly
    - We needed an easy to way in which one could add database entries, APIs, and Machine Learning jobs without needing being concerned about the rest of the system. ODK did not provide an easy way for us to achieve this.

Market Research Conclusion:

Mainly due to the complexity of ODK for backend engineering, we choose to not pursue this avenue further.

1.2.2 Commcare

Commcare [2] is a powerful data collection platform, by Dimagi.

Pros:

- Easy to Set Up - Similar to ODK, Commcare is extremely easy to setup and deploy.
- Affordable - Of all commercials solutions we explored, Commcare emerged as one of the most affordable solutions.
- Scalable - They provide excellent scalability guarantees for small to medium scale projects like ours.
- Offline support - We were pleasantly surprised that they included an app which could work offline.

Cons:

- Lack of native support for custom Machine Learning Algorithms.
    - We need a system which enables us to easily develop plugins. These plugins require custom DB measurement types, and of course custom Algorithms.

Market Research Conclusion:

As it would have required a lot of engineering on our end to provide this Plugin-like interface we are seeking to establish, we decided to explore other solutions, including building ours.

1.2.3 Amazon EMR

This is an EMR System offered by Amazon [3] on AWS.

Pros:

- Security
    - Data is safeguarded by Amazon
- Scalability
    - Backed by Amazon's Elastic Storage, which will permit us to grow
- Relatively cheap to operate
    - Users pay by the hour, and Amazon offers a very generous trial package.
    - The code lives on an Amazon-hosted EMR Notebook and is backed up in S3. This further makes the whole package cheap to operate.

Cons:

- Lack of an easy User Group Management support
    - Unfortunately, Amazon EMR is very research data focused and seems to be best suited for users who only upload measurements and analyze them on the cloud.
    - We would need to invest a lot of time and effort in building an extra cloud system to provide just this missing part, but that increases the complexity of our system and increases the overhead during onboarding and maintenance.
- Unicorn Environment
    - Because this is purely a cloud service set up on Amazon and hidden from end users, it makes bugs more difficult to track down. Students and collaborators will not have a way to reproduce Amazon's configuration on their local machines, nor will they ever

have a one-to-one representation of the cloud deployment they are supposed to be working with. This black-box nature, despite the attractiveness of the features discussed above, made Amazon EMR an unpleasant prospect for our group for some use cases.

- Ties us to Amazon
    - We needed to make sure that our platform is not overly tailored to and dependent on Amazon's services. This would make it harder to switch Cloud Providers in the future without sacrificing a considerable amount of development time as Amazon EMR can quickly become a significant financial burden down the road as our system is more widely used.
- Closed Source
    - Amazon does not open source the code which powers their EMR product. As a result, we will have to trust them with medical data which our research partners have entrusted to our group. The usual way to provide data to untrusted third parties is to encrypt or obfuscate the data before handing it over to the third party. This meant that significant effort would have been required to make Amazon EMR work for us.

Market Research Conclusion:

Despite the Pros being quite attractive, the Cons outweighed them far too much for our group to embark on this journey with Amazon. We would have needed to invest quite a bit of engineering efforts to make this system work for us and we would have ended up with a less flexible system that is more dependent on Amazon as a Cloud Provider that we would like. Hence we rejected the idea of using Amazon EMR.

1.2.4 Open EMR

We started looking into open source solutions, which we could spin up and manage ourselves at low cost on any cloud provider. We found such an EMR solution in Open EMR [4].

Pros:

- Open Source
    - This means we had access to the entirety of the code which touched the medical data we collected, so no need to spend time and as much effort on obfuscation techniques to anonymize our data further.
- Common Execution Environment
    - This is quite an attractive feature for us, as it means we can share setup instructions for other students or researchers to use so as to reproduce bugs or other anomalies.
    - To make this process even smoother, we can share a VM image which contains a barebone installation of Open EMR, which can be used as a baseline for every party involved and as a packaging system for our Cloud Provider.

Cons:

- Complexity
    - Open EMR is an extremely generic system which would have required a substantial engineering undertaking on our end to fully understand and simplify for other students and research partners.
- Deployment
    - Due to its complexity, it adds an overhead which would make it quite challenging to tweak and reshape for new projects that might spring up. This makes Open EMR not flexible enough for our needs.
- Project Size

- As mentioned above, OpenEMR is bulky and complex. This is ideal for resourceful organizations that need those extra features such as billing and scheduling, but this tends to be quite a burden for smaller more dynamic teams like ours that may not have all the resources to fully vet such a solution, understand it through and through, and customize it to support the extra features we needed from it.

- Lack of Python Support - OpenEMR is mainly PHP based.
  - Python has rapidly become the most popular programming language known to Computer Science students here at MIT. As a result, basing our technology on PHP would have substantially burdened the group's prospective students during the onboarding experience into our group. Undergraduate Research positions usually only last a single semester and the Machine learning tasks completed by our students are usually quite technically involved by themselves already.
  - Some plugins will require some Database modification as well as some API linking, and hence will require some PHP code to glue together with almost every Machine Learning script contributed by a student. We worry that figuring out this process will further decrease the net time which our students will be able to spend actually working on their project.

Market Research Conclusion:

Due to the cons listed above, we decided not to move forward with Open EMR.

1.2.5 OpenMRS

OpenMRS [5] is another software platform we came across while looking for an open source option. It is being built by a community of volunteers

worldwide and allows building customized medical records systems with limited programming experience.

Pros:

- Provides well-built interfaces for quickly building studies even with no programming experience.
- The platform already has features in place for login, security, data entry and export and localization.
- Open sourced
    - As with OpenEMR, having access to the entirety of the code base negates the need to obfuscate data.
- Android support:
    - Open MRS already boasts an android client which is attractive given our chosen medium of data collection.

Cons:

- Java vs Python:
    - Open MRS is built mostly using java, which is becoming less frequently used by rising students at MIT during their courses. Hence, similar to Open EMR, the language barrier again acts as an avoidable hurdle in the way of students making progress on the business logic of their contributions to our lab's ongoing projects.
- Code Size
    - Again with Open MRS, the complexity and bulkiness of the code base makes it less tractable for UROPs to be able to overcome the learning curve and start contributing to their projects.
- No offline synchronization support
    - One important functionality we need in order to be able to operate in remote regions with limited connectivity is offline support. Unfortunately, Open MRS does not offer offline support and the

code base being large and not specialized to meet our needs, makes it harder to tweak.

Market Research Conclusion:

After evaluating the options above, we decided against a complex Java Solution, and instead opted to focus on finding a Python solution which we could onboard a new undergraduate student into, in under 2 weeks.

1.3 The Case for the PyMedServer Framework

During our market research, we realized that multiple platforms offered a subset of the features we sought: scalability, security and data security among others. However, a gap was apparent in a key aspect: the applications built using these platforms provided little to no offline support which important for operation in low-resource areas that do not have access to the internet. In addition, the coding language barriers and learning curve for the complex code bases we encountered left us wanting for more streamlined solution. Upon further search, we were unable to find a suitable Python EMR framework that could fit our needs. Hence we decided to build a custom and flexible platform, using a language that facilitates on-boarding and maintenance.

Special precautions were taken to ensure that the security and scalability provided by the servers built using the PyMedServer framework were of high quality. Most importantly, from its conception, PyMedServer was designed with simplicity in mind. Our goal was clear, it should be easy for an undergraduate student to set up a fully working PyMedServer is one sitting, and to be in a position to start contributing to his or her project in at most 3 weeks, or 18 hours in total.

The next chapter describes in detail how PyMedServer was designed, and the hows and whys of its design. We will also expand on how PyMedServer makes up for the cons of alternatives like OpenEMR or AmazonEMR and how PyMedServer's design allows for flexibility and also catering to our future needs.

2 TECHNICAL DESIGN

This chapter describes in-depth the technical design of the PyMedServer framework. At this point, it is important that the design goals of the framework be formalized.

2.1 DESIGN GOALS

Below are some of the core design goals behind PyMedServer in somewhat rough order of importance.

2.1.1 Maintainability

Maintainability was a top focus because of the expected lifetime of this framework. Some of our studies run for years, and we might export this technology to research partners who could use this framework for years beyond this thesis. If the core framework code is poorly written, it will be extremely difficult to find maintainers for this project over time - in other words, the framework could decay rapidly. This design goal trumps all others mainly because a maintainable system with a good design foundation is much easier to evolve on - security can be added, plugins could be added, more unit tests could be added, etc. So it was crucial that this framework be well written, thoroughly documented, and well tested.

2.1.2 Security

We need our servers to be robust to attacks. We understand it is impossible to guard against all attack vectors, but our goal here is to make the system difficult to penetrate. Common attack patterns such as dictionary attacks, replay attacks, or outright sniffing should be mitigated as much as possible.

Because this is a framework, it was all the more important that PyMedServer's security modules be easy to patch in. This is a subtle but important requirement, as it means the framework should transparently allow for security

patches to be added in without the project developers needing to do much. This design goal, if achieved, would further simplify the life of project engineers, who ideally should remain plugin experts - that is, they should be subject matters on their plugins, and not have to worry about the overall security of the system.

2.1.3 Malleability

Due to the variety of use cases which we may encounter in the field, it is important that this framework be malleable enough to cope with the myriad of data types any one of our servers may need to support. At the minimum, we will need a database layer which helps us support files, in addition to the vast array of types supported by SQL.

Contrary to how OpenEMR operates, it is important that the addition of these new plugins be simple to do and not require in-depth knowledge of SQL commands to perform database migrations. Similar to the Security design goal, this design goal also aims at keeping project engineers plugin experts and ensure they spend the vast majority of their efforts on building their plugins without worrying about such framework changes.

2.1.4 Plugin Friendliness

It is important to formalize what a plugin is. We can define a plugin as a set of related database, API, and background job code changes which act together to enable a new feature. For example, a fever-analysis plugin could include some temperature tracking fields added to the database, along with a background job to perform the analysis, and a set of APIs to expose both the temperature and the analysis results to clients.

The easier this process was, the quicker our students could focus on the core logic behind their plugins and not spend time figuring out the other requirements for running a production server on the cloud serving internet traffic - such as deployment, security, scalability, form validation, IP banning, etc.

2.1.5 Built-in Features

Because this was an in-house solution, we could pull a lot of common features into the framework. For instance, Authentication is a vital feature, but one which is virtually the same across all our projected EMR installations. So, it is a prime candidate for some "out of the box" features the framework should already provide. Standardizing authentication will again, further alleviate the cognitive load on our students and research partners. Below are some such core features which the framework was designed to provide.

2.1.5.1 Authentication and Authorization

Authentication and Authorization are common requirements across all our PyMedServers, and as a result should be readily provided by the framework. The framework however, must provide these as opt-in features, so as to permit customization options. This customization mantra applies not only to this built-in feature, but all other built-in features as well - that is, it should be relatively easy for project engineers to customize or outright skip any built-in feature.

2.1.5.2 Group Isolation

Formally, Group Isolation refers to the ability of the system to differentiate between objects belonging to distinct Groups. A Group could be defined as a set of clinicians, administrations, and patients sharing a professional relationship. Group Isolation is extremely important for all our PyMedServers, as it allows Clinicians and Administrators to create a logical arrangement of medical data from patients. Due to the foundational nature of this requirement, it is important that this should be guaranteed by the framework. In fact, this is one of those design points which could be very challenging to add after-the-fact as that Group information needs to be tracked from the very creation of every object. One could imagine how difficult it is, to retroactively tag thousands of measurements with group membership information.

2.2 ARCHITECTURE OVERVIEW

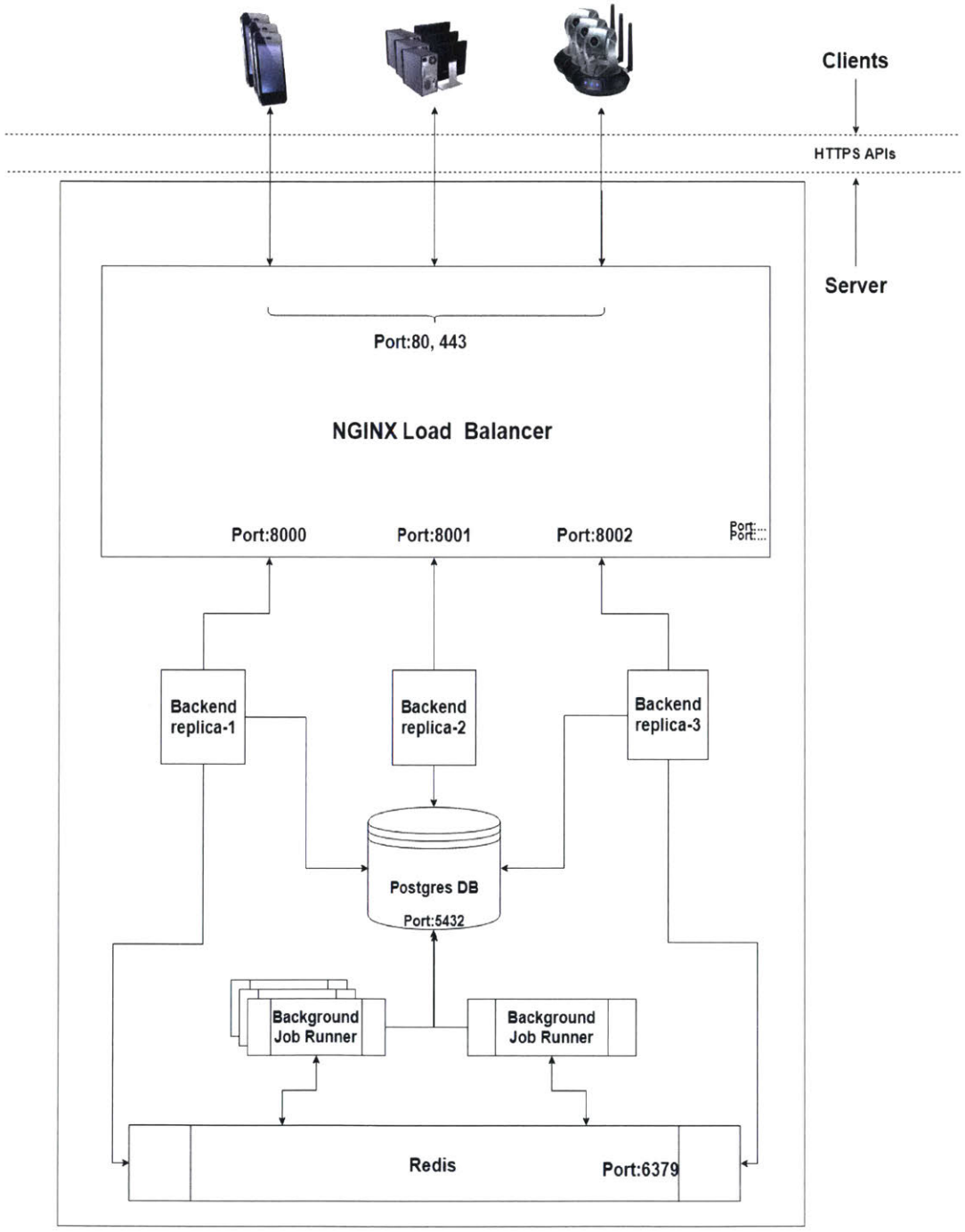

Figure 1 - PyMedServer Architecture

The default PyMedServer configuration provides a consumer grade Load Balancer, NGINX [6], which is configured as a reverse-proxy to three identical Backend replicas acting as Web Servers. The backend replicas are responsible for serving API requests, while NGINX is responsible for serving static content. The Backend Replicas are themselves configured with two endpoints each, a PostgreSQL [7] Database endpoint, as well as a Redis [8] Server endpoint. The backend replicas use their Redis Server endpoint to kick-off background jobs on a Background Job Runner. Lastly, Background Job Runners are configured with the same PostgreSQL Database endpoint as the Backend replicas, and both these components use their PostgreSQL Database endpoint to communicate with the database. In subsequent sections, we will unpack each component and further explain their individual roles. To close this section off, we will provide a 'Life-of-an-HTTP-Request' path to explain how all these components tie together.

2.2.1 The NGINX Load Balancer

PyMedServer uses NGINX as a free, open-source, high-performance load balancer. At a high-level, PyMedServer's use of NGINX is twofold:

1. As a Reverse-Proxy to three Backend Replicas.
    - A reverse-proxy [9] could be understood as a server which listens on a port, and simply forwards all requests to backends that actually handle the request.
2. As a Static File Server for all uploads and media files required by the server.
    - All static files are served from NGINX as it is more optimized to serve static files than Backends [10].
    - In particular, every request served from a Backend would incur extra overhead, as the request would first be received by NGINX, then copied over to a backend, then the response from the backend would be copied back to NGINX and then finally sent out

to the client. By making NGINX serve those static files directly, we are avoiding the additional overhead of hitting the backend.

All HTTP and HTTPS requests from the internet for a PyMedServer will hit the NGINX server first, as it is the sole process which listens on Ports 80 (HTTP) and 443 (HTTPS).

2.2.1.1 Configuration

For security reasons, this section will not delve into overly specific details but will touch on some of the various features PyMedServer configures on NGINX.

- SSL Termination [11]
    - NGINX is configured with the ssl certificates for the domain belonging to this PyMedServer. This is because in some cases, NGINX will need to be able to read the SSL encrypted request, and hence needs the right certificates to be able to decrypt the request payload.
- The NGINX server is configured to use a custom SSL cipher suite:
    - This is to ensure the PyMedServers are not susceptible to attacks which might work on widely public/default cipher suites.
- Multiple backends
    - By default, PyMedServer configures three backends. The backends are all identical, but listen on different ports. A Backend replica endpoint configured on NGINX is comprised of a `Hostname:Port-Number` combination. As such, Backends can be located on completely separate physical machines for scalability purposes.
- Request sizes are limited:

- This is to mitigate a class of DoS attacks where the victim server is bombarded with extremely large requests - forcing the victim machine to spend considerable computational resources on parsing.

- Sessions are short-lived:
  - Both the Backends and Nginx are configured in such a way that sessions are forgotten frequently. This is to mitigate a class of replay attacks.

- Backend configurations:
  - Backends are configured **only** on HTTPS.

- HTTPS redirects
  - All unsecured HTTP requests are rejected with an HTTPS redirect by NGINX. This is to ensure that communication between the backends and the client are SSL encrypted. Here is a snippet responsible for such a redirect.

```
server {
    listen 80;
    Server_name myserver.com;

    location / {
        return 301 https://$host$request_uri;
    }
}
```

  - In the above snippet, the NGINX configuration instructs the server to listen on port 80 for **http://myserver.com** and return an HTTP redirect error to the client, prompting them to retry **https://myserver.com** instead. This is to guarantee that all

communications between the client and our server is always encrypted. It is important to note that, even static file HTTP requests are redirected to HTTPS.

- Alias Hostname Support
    - PyMedServer configures NGINX with alias hostnames. This allows the backends to receive requests from each of these aliases, e.g. `www.myserver.com` and `myserver.com`.

2.2.2 Backend Replicas

This is the central Web Server responsible for serving all APIs. For flexibility, JSON HTTP APIs were chosen as main interface of communication between the server and its clients. This is mainly because JSON is a widely popular framework, and hence facilitates compatibility with multiple client types. At its core, this web server is built using the Python Web Server framework - Django [12]. The next section describes why Django was chosen as our Web Server framework.

2.2.2.1 Why Django

Django was chosen due to its simplicity of setup, modularity, abundance of security features, and excellent community support. Django is maintained by thousands of developers and security experts, and is used within many companies around the world [13], including Instagram [13], YouTube [13], Everbrite [13], and many more [13]. This active community and support makes it ideal for long-lasting server frameworks like PyMedServer.

An important design requirement for PyMedServer is the ability to support security patches. By delegating its security foundations to Django, PyMedServer inherits this feature for free [14]. That said, quite some work was needed to ensure that PyMedServer is secure and scalable by default.

Out of the box, however, Django does not configure a secure, scalable server. This is because the Django framework aims to be a low level library which

simply recommends production configurations instead of imposing possibly incompatible configurations onto some of their users. PyMedServer on the other hand, strives to be at the opposite end of that default-setup-simplicity spectrum. It aims to be very strongly opinionated, so as to streamline the experience among Plugin developers. As a result, PyMedServer heavily modified its setup from Django, so as to achieve the best of both worlds, i.e. a scalable, streamlined system, that is optimized for plugin development and while also harboring all the underlying benefits of Django, including security and database integration features.

2.2.2.2 Configuration

By default, Django reads all its configurations from a file called `settings.py` [15]. This file is the main source of all of Django's complexity and flexibility. PyMedServer completely hides this configuration file away from plugin developers and exports a separate, much more restrictive configuration file called `config.yaml`. This config file is used to provide a PostgreSQL Database and Redis Server endpoint which the PyMedServer uses. PyMedServer then provides a robust `settings.py` file, ensuring that all security parameters are properly configured, and that the resulting backend configuration is in sync with NGINX. For instance, PyMedServer configures aliases on NGINX, and hence needs to ensure that the generated `settings.py` file for Django, contains these aliases as well.

2.2.2.3 Database Integration

By default, Django communicates with a database using the Django Object Relational Mapping (ORM) framework [16]. In that framework, Models are Python classes that represent database tables. Models permit easy composition of complex SQL queries using rather basic Python Object Oriented Program paradigms. This is a great plus for Python developers, as it means they will not need to learn otherwise complex, insecure, and error prone SQL. Finally, Django

also provides a database migration program, which automatically manages database migrations whenever any Models are added, edited, or removed.

PyMedServer does not configure any of these Database features differently from the Django's defaults. PyMedServer, however, needed to provide built-in support for Authentication and Authorization. As a result, it provides a set of Opt-in Models, which when enabled, automatically activates the default Authentication and Authorization modules available within the framework.

2.2.2.4 APIs

Out of the box, Django does not offer first-class support for JSON APIs. As a result, all our PyMedServers come pre-packaged with an extra Django library called Django Rest Framework [17] (DRF). DRF provides `Serializers` [18], which trivialize the serialization and deserialization between native Django Models (i.e. "DB table rows") and JSON.

PyMedServer provides a series of optional APIs. These optional APIs work hand-in-hand with the optional Authentication constructs provided by PyMedServer. The optional nature of these modules are to make sure it is easy for plugin developers to opt-in or out, and extend the built-in modules.

2.2.2.4.1 How APIs work

Once a request is received by the webserver, there is a set of operations performed before the API code is executed, and a response is returned to the client via NGINX. Below is a diagram illustrating how all HTTP Requests are processed on Django.

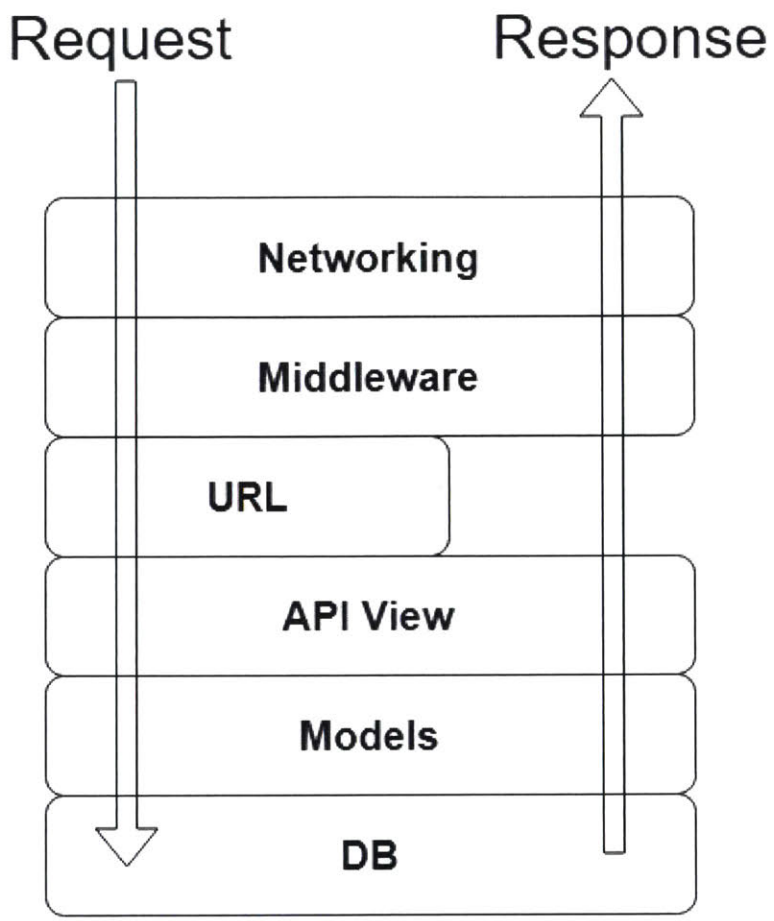

Figure 2 - Django Request Response Flows

Description

- Networking:
    - This refers to the networking framework within which Django runs. Out of the box, Django does not provide a production ready solution to listen to ports and write to networking sockets. PyMedServer configures Django to run within **GUnicorn** [19].
    - GUnicorn was chosen as it is a lightweight, robust, production-ready system, that is also recommended by Django.
    - More specifically, PyMedServer configures Django to listen to a port and write to that port's network socket.
- Middleware:

- This layer comprises of a series of libraries which validate and/or process the request towards, or the response from the API function.
- Django performs its security checks in this layer.
- It is in this layer the bulk of Authentication and Authorization happens.
- PyMedServer also utilizes this layer for Authentication, Authorization, Logging and Input and Output sanitization.
    - PyMedServer uses Token Authentication. In a nutshell, a cryptographically secure character sequence is handed out to users, so clients can use this sequence in-lieu of password transfers.
    - The Authentication Section in the Webserver Features [3.1.2] of PyMedServer will go into further details into how Authentication and Authorization is achieved.
- The figure below further breaks down the middleware layer and illustrates a sample middleware stack. The HTTP Request is passed through the SecurityMiddleware first, and ends at the XFrameOptionsMiddleware. The response on the other hand, is first received by the XFrameOptionsMiddleware, and eventually makes it to the SecurityMiddleware stage.

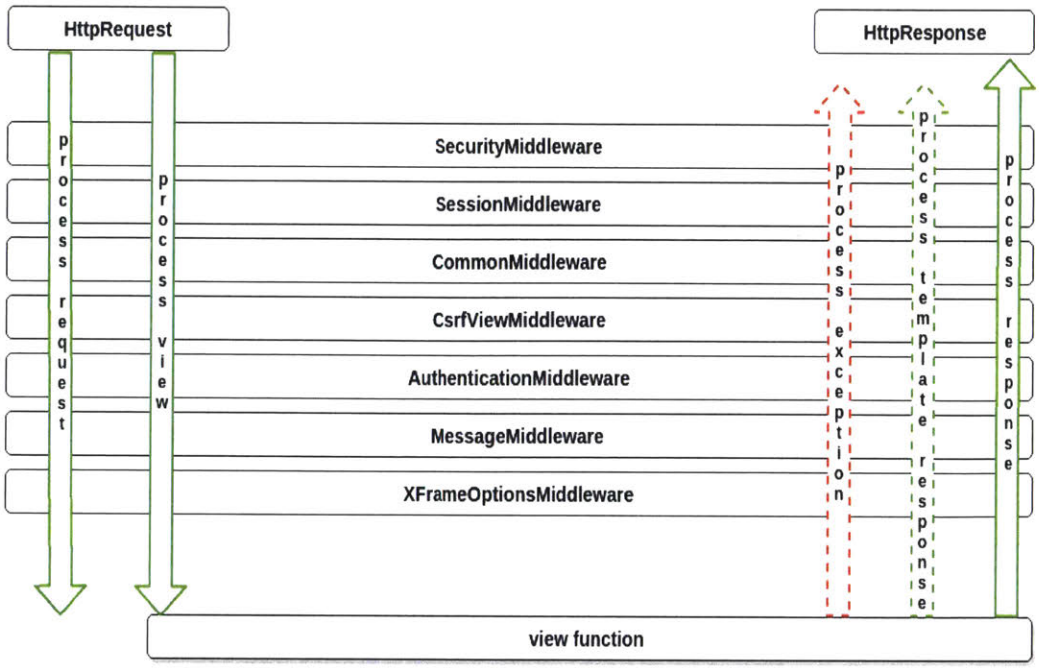

Figure 3 - Django Middleware Execution Stack

- URLs
    - In this layer, Django performs a reverse look-up, to match a URL to a unique API function.
    - For maintainability, PyMedServer enforces that API paths match folder structures. This is great for maintainability, as URLs self document the location of their handlers. For instance, the URL `https://myPyMedServer.com/apis/diagnostics/view_measurements/` will be located in the folder `project_root/apis/diagnostics/view_measurement/...`
- API View:
    - A View in Django terms, is a function which receives an HTTP request and returns an HTTP response.
    - In the case of PyMedServer API Views, the HTTP request and response bodies are in JSON format.

- API Functions utilize the database layer to complete their request. This Database access happens through Django's `Models` framework.
- Optionally, the API function can provide custom permission policies. PyMedServer provides a comprehensive set of base policies, which developers can compose into more complex permissions.
  - For instance, the policy `IsPatient`, will reject the request if the user making the request is not a patient. The `IsGroupMember` policy will reject the request if the user making the request is not part of the Group this request is about to modify. Both these permissions can be combined by a developer, using an AND operation, to create

```
IsPatientOfGroup = And(
    IsPatient, IsGroupMember
).
```

- Models and DB:
  - Commonly API functions use Django's Models framework to access the database. PyMedServer does not change this dynamic between the Plugin API Functions and the database.

2.2.2.4.2 API Data Types and File Uploads

Being powered by JSON, all PyMedServer APIs support all JSON data types. So, Text, Floating point values, Numbers, Booleans, and Null (Empty) Types are supported out of the box.

All Dates from the server, are transmitted as their Epoch Timestamps. The input parser however, supports interpreting dates from their ISO 8601 format.

Our APIs needed to be able to handle File uploads from clients. A lot of the infrastructure around PyMedServer, such as permissions and loggers, are

optimized to deal with JSON strings efficiently. Switching to Multi-Part parsing so files could be uploaded as separate blobs alongside the rest of the data, was not supported by our clients. As a result, files uploads are required to be serialized to Base64 Strings so they could be re-constructed by the server.

2.2.3 Background Job Runners

Background jobs are a critical component in PyMedServer framework. PyMedServer institutes the policy that all Machine Learning Algorithms be ran in the background - regardless of their expected duration. This is because, most of our machine learning tasks actually run longer than the default HTTP Read timeout configured on NGINX. So this decision was taken to ensure that all plugins are written the same way and interact in a rather predictable way vis-a-vis the framework. Django out of the box does not support background jobs, so we relied on another python library called Celery [20] to breach the gap. In the next sections, we explain how background jobs are kicked-off.

2.2.3.1 How Background Jobs are kicked off

One of the main reasons why we use background jobs is to ensure that a single request does not block one of our valuable backend replicas for a relatively long time. Ideally, all backend replicas execute their requests at quick rates, delegating all the heavy-lifting to background jobs.

By default, PyMedServer configures two Background Job masters (Celery Workers). These masters do not themselves run the background jobs, but rather, they serve as coordinators, spinning up slave processes (Background Job Runners) who actually execute the jobs. These Redis queues are First-In-First-Out (FIFO) multi-producer/multi-consumer queues [21]. Backend replicas register as producers, while Background Job masters register as consumers.

Background job runners have access to the same database as Backend replicas. As a result, background job request parameters include only primary key identifiers for tables, so the jobs fetch fresh values from the DB every time they

need to run. This is because, jobs are not guaranteed to run as soon as they are enqueued. There are several reasons why that guarantee is loose, for instance the queue might be full, or there might not be any available workers and the masters need to wait for workers to be available.

2.2.4 Redis Server

As mentioned in the Background Job Runners section, Redis is extremely important to PyMedServer as it is used as a queue to kickoff background jobs. Another vital use of Redis is as a Caching Layer for the Database. Though the background jobs pipeline simply uses Redis as a message broker, the Backend replicas also use Redis as a DB cache. In particular, all DB read requests are first forwarded to Redis, and only on failure do they pass-through to PostgreSQL. Though PyMedServer takes care of the configuration, the caching and retrieving are done with the help of a django add-on called `Django-Cachalot [22]`.

2.2.4.1 Configuration

By default, Redis is a *persistent* in-memory database. Periodically, Redis saves its state to disk in a way that guarantees consistency across reboots. **PyMedServer disables this persistence feature, and reconfigures Redis as a purely in-memory data store for its caching purposes**. This allows Redis to not have any disk footprint, and hence perform more efficiently for the two use cases which our PyMedServer will subject it to.

2.2.5 PostgreSQL Database

PostgreSQL was picked as our database of choice mainly because it is the RDBMS system recommended for Django. The next Sub Chapter [Scalability of PyMedServer], goes into details as to how PostgreSQL scales.

2.2.5.1 Configuration

PyMedServer takes care of installing and configuring the PostgreSQL server. PyMedServer reads its configuration file (`config.yaml`), and creates the required database. The config file will contain the database name, and other credentials which are required to ensure all internal systems can access this project's database.

2.2.6 Supervisor

Supervisor [23] is a process control system, which PyMedServer uses to ensure that the Backend Replicas, as well as the Background Job Runners, are restarted in case the operating system kills them inadvertently. In addition to keeping those processes running, supervisor is also configured to rotate the logs generated by those processes, so they do not grow indefinitely.

2.2.7 Life of an HTTP Request (Recap)

The following diagram illustrates the path taken by an HTTPS request towards a server powered by PyMedServer.

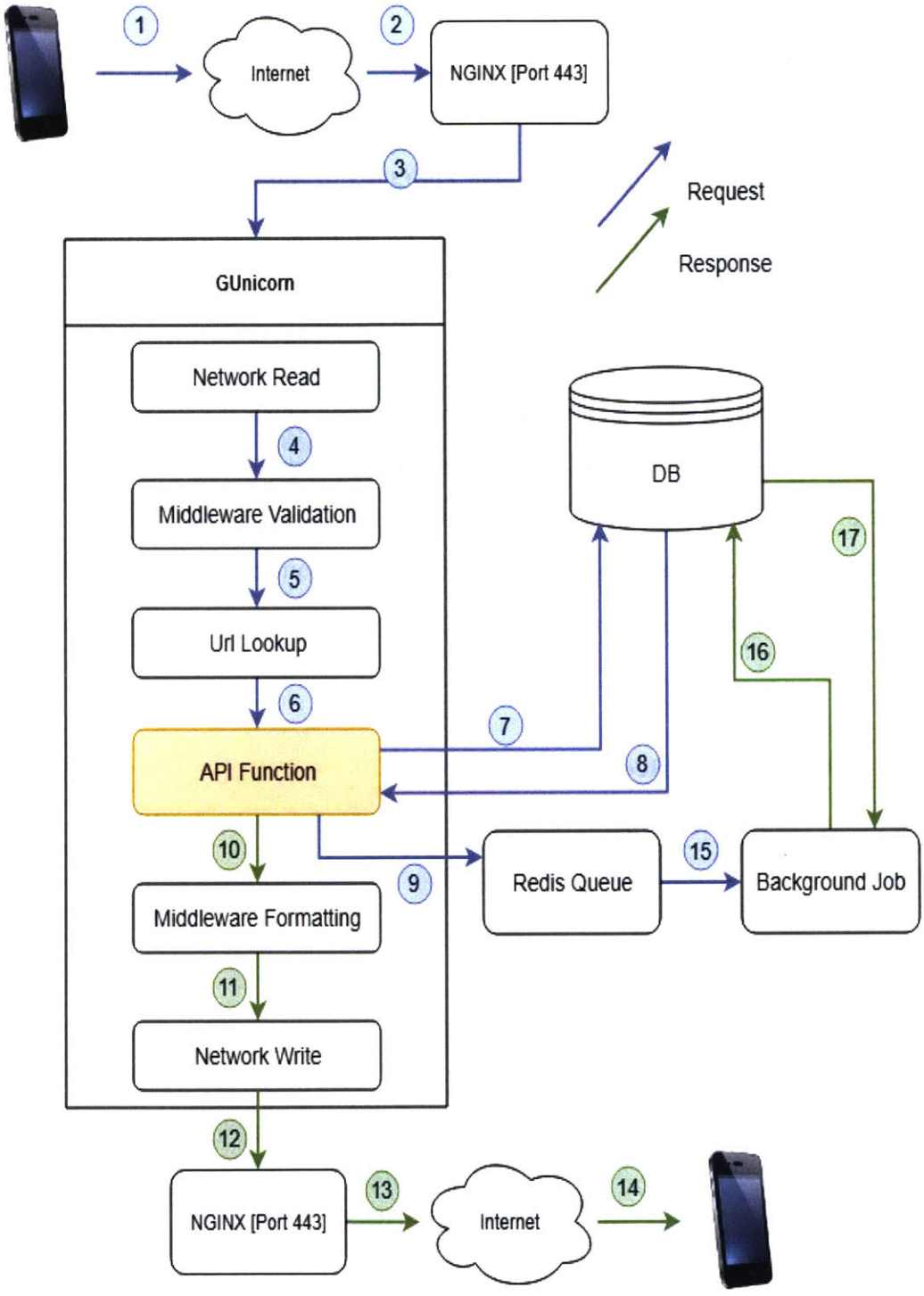

Figure 4 - Life of an HTTP request

2.2.7.1 Legend

1. The client sends out an HTTPS API request towards a PyMedServer. At this point, the request is routed across the internet till it reaches the machine hosting our PyMedServer.
2. NGINX listens on port 443, so receives the request. If this were a request for a static file, NGINX could serve such a request directly. In this case however, this is an API, so NGINX will need to forward this request to one of the backends it is configured with.
3. NGINX picks a Backend Replica available, and forwards this API request towards the port the backend is configured to listen on.
4. GUnicorn listens on the port, and successfully reads the request which NGINX forwarded (wrote) to this replica's port.
    a. Recall the GUnicorn runs Django, so there are more steps required before our API function receives this request and converts it into an appropriate response.
5. The request traverses Django's middleware stack. This is where PyMedServer Authenticates this user, and pulls out this user's information from the DB. The communication with the DB is omitted from the diagram.
    a. It is also at this layer where Django runs its suite of security checks, and potentially rejects the request.
6. Once the Django security suite as well as the PyMedServer checks have passed, a reverse lookup is performed to find the API function that is responsible for serving this request.
7. The API function has access to the database and makes a set of DB queries.
    a. In reality, only write requests are sent directly to the DB. Read requests are first forwarded to the Redis Cache. This detail is omitted from the diagram.
8. The PostgreSQL DB responds with the query "values" requested.

    a. It's important to not do so for most db queries, only DB Cursors are returned, and not raw values. This is a performance optimization.
9. For this API, we assume that a background job (machine learning task) needs to be executed. Due to this, the API function will enqueue a Background Job request into a well-known Redis Queue.
    a. As mentioned in the Redis section, there is no guarantee that the background job will start as soon as this request is enqueued. Because of this, Event 10 is actually the API Function returning the response back to Django.
10. Here, the API function has performed all the operations it needed to, and hands it's reply back to Django's middleware stack.
11. The response from the API function traverses Django's middleware stack in reverse order (last middleware first). Assuming everything goes well here, Django hands the finalized response back to the GUnicorn.
12. GUnicorn writes the finalized response back to NGINX.
13. NGINX encrypts the response using the HTTPS protocol, and forwards it towards the client.
14. The Client receives the response from the API function.

This concludes the architecture overview of PyMedServer. In the next section, we describe the scalability qualities of this architecture, and conclude this chapter.

2.3 Scalability of PyMedServer

To understand how PyMedServer scales, we need to analyze the various types of scaling axes our server could be subjected to.

2.3.1 Increase in rate of API calls

If there is a sustained increase in the number of Queries Per Second, the number of Backends can be scaled up to meet demand. NGINX will then have an

extra set of Django backend replicas on which it can spread the load. NGINX simply needs a hostname:IP address to know where to forward requests. If we really need to, we can split out the Django backends into clusters of machines, and simply configure NGINX to splay the traffic to all those backends.

2.3.2 Increase in processing time per API.

As the system scales, there might be times when some plugins will require more processing than others. There is a default timeout NGINX imposes on every API before giving up on waiting for a response from the backend. This default timeout is short, so as to encourage plugin developers to gravitate towards setting up background jobs instead. The default number of background job runners is two, but this can be scaled up to meet demand as well.

2.3.3 Increase in number of Queries to the DB

A PostgreSQL Database Server can easily handle over tens of thousands of queries per second. We anticipate this layer to be the last affected layer when scaling becomes an issue.

That said, the PostgreSQL process shares [limited] physical resources with all the other components on a single machine by default. If the DB processing times become high due to resource contention, PostgreSQL could be moved to a dedicated machine. It is important to note that no PyMedServer component depends on the database being local - all components refer to the database via its public address and port, and some internal credentials. By default, the address is just `localhost`. Once on a dedicated machine, the RAM and disk space available could be increased to fit the needs of the server.

If scaling becomes an issue even beyond the capabilities of one dedicated server, the database could be replicated, into a Master and Slave configuration. Django (and hence PyMedServer) supports this by default.

Beyond this point, the database will need to be sharded appropriately. Fortunately, Django (and hence PyMedServer) supports this.

Overall, sharding and replication can help any PyMedServer support up to millions of requests per second, and thousands of concurrent connections.

2.3.4 Increase in Redis Usage

As we discussed above in the Redis section, Redis is already configured to be an in-memory datastore only. If, however, memory resource constraints become a problem, Redis could be moved to a dedicated machine. Similarly to the database, the Redis endpoint is a public identifier, `Hostname:Port`. It just so happens that on the default PyMedServer configuration, the hostname is localhost, and the port is 6379, hence the identifier `localhost:6370`.

Beyond a single machine, Redis could be configured as a cluster - a Redis Cluster. For our use cases however, we do not anticipate getting to this point.

This concludes the Design overview of PyMedServer. In the next chapter, we discuss in greater detail the built-in features offered in PyMedServer.

3 WEBSERVER FEATURES OF PYMEDSERVER

One of the major design goals for PyMedServer was for it to provide a rich set of features at the framework level, so as to simply the task of developers. In this section, we go over some of those features.

3.1 AUTHENTICATION AND AUTHORIZATION

As mentioned previously, Authentication and Authorization are common needs across all our server projects. These were in some regards, perfect targets for the framework to provide. In this section, we will go over how PyMedServer provides Authentication and Authorization primitives to developers.

3.1.1 Abstract Database Authentication Tables

The diagram below illustrates a rough sketch of the way in which PyMedServer provides database features to developers. The solid rectangles represent concrete database tables, while the dotted rectangles represent abstract database tables provided by PyMedServer.

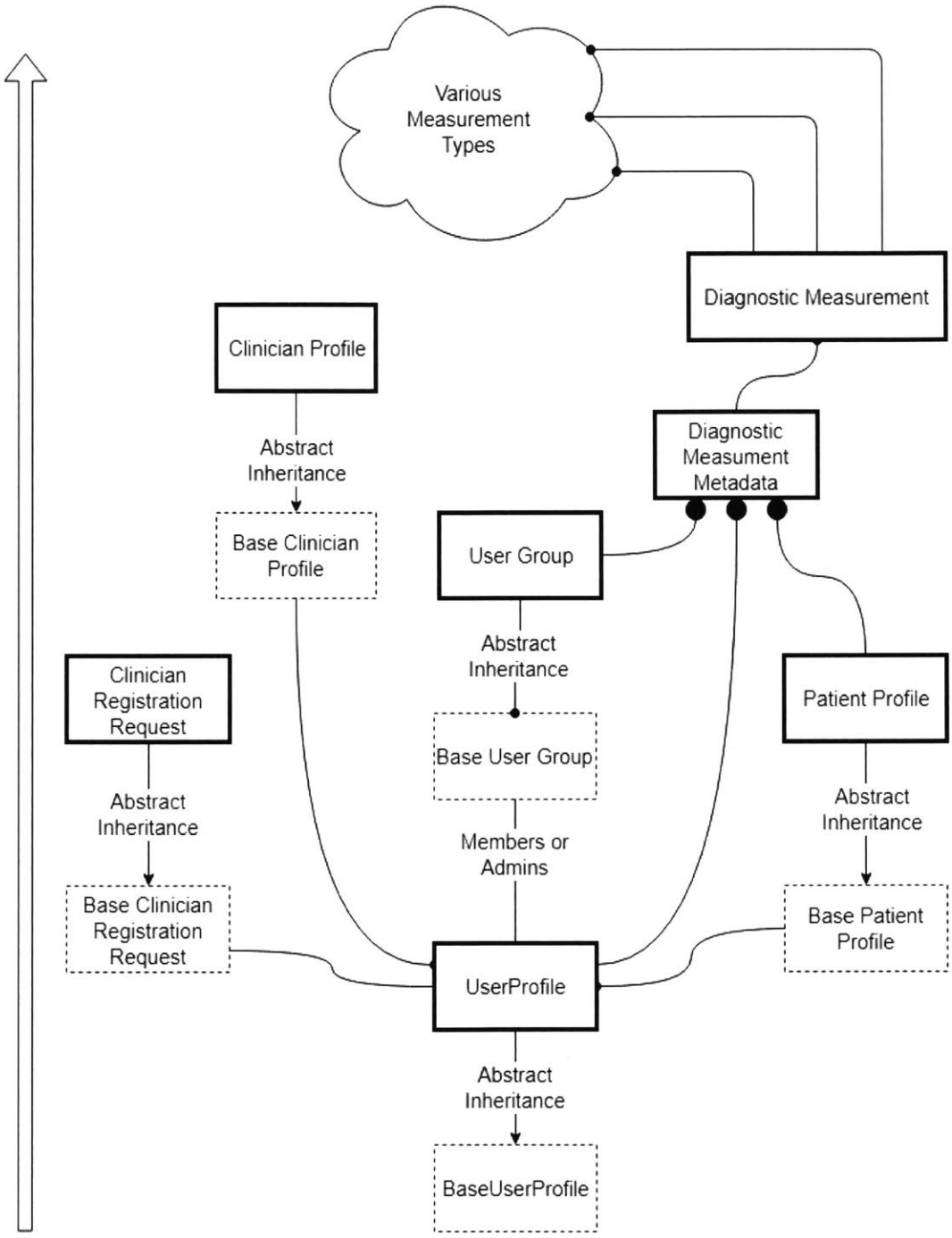

Figure 5 - PyMedServer Authentication Suite

PyMedServer provides a full Authentication suite, but also provides developers an easy way to extend those tables. For instance, the developer only

needs to declare an empty UserProfile Python subclass of BaseUserProfile, and the user would have enabled this feature. For example

```
...
class UserProfile(BaseUserProfile):
    pass

class ClinicianProfile(BaseClinicianProfile):
    Pass

...
```

Abstract Inheritance was chosen instead of composition because composition would have resulted in more complex and slower SQL queries. This is because, with composition, we need to perform more queries to resolve to these Base tables, while in inheritance, the user controls the Base tables directly.

3.1.2 Authentication

PyMedServer provides a suite of Authentication APIs. Some of these APIs include:

3.1.2.1 Clinician Registration Request

Using this API, Clinicians submit their registration requests to the server for Admin review. Administrators will be notified, and are given a chance to review all registration requests.

3.1.2.2 Patient Registration

This API allows Clinicians to enroll Patients into the Platform.

3.1.2.3 Clinician Registration Review

Through this API, Administrations can edit, approve or reject Clinician registration requests.

3.1.2.4 User Login

Using this API, clients convert user password credentials into Tokens. Tokens are cryptographically safe strings, which uniquely identify a User. These tokens expire on the server side, and are not guaranteed to be stable across logins. As a result, our clients are advised to store these tokens but not rely on them to be accepted always.

The pattern this encourages is for clients to issue a GET request towards this endpoint with the previous token they stored. This GET request if successful will indicate to the client that the server still recognizes the token. If the GET fails however, the clients should display a login screen.

3.1.2.5 User Logout

This API invalidates this user's session on the server. This is mainly used by clients which support sessions, like browsers.

3.1.3 Authorization

Authorization could be defined as the process of ensuring that users operate within some defined set of permission constraints. PyMedServer provides a suite of permission policies to developers. Each policy is by itself quite simple, but can be combined with other policies to construct more complex composite policies.

Permission policies have access to the database, and are executed after Authentication takes place, so have knowledge of the user as well as the API function invoked. In fact, permission policies could be thought of as wrapper functions, which wrap API Functions, ensuring that the API Functions they wrap are invoked only after their policy requirements are met.

It is important to note that these Permission policies offered by PyMedServer are optional. This is because they are based on the assumption that the developer is indeed making use of the framework's Authorization suite. In

subsequent sections, we will showcase a few of the Base Permission Policies offered by PyMedServer.

3.1.3.1 IsClinician

This policy ensures that only registered Clinicians are permitted to invoke an API Function. This policy is achieved by ensuring that the user issuing the request possesses a Clinician Profile.

3.1.3.2 IsPatient

Similar to IsClinician, this policy ensures that only registered Patients are allowed through. This is done by ensuring that the user issuing the request has an active Patient Profile.

3.1.3.3 IsReadOnlyMethod

Under this policy, only HTTP read-only methods (GET, HEAD, TRACE, and OPTIONS) requests are permitted. All other HTTP Method types will be rejected with a METHOD_NOT_ALLOWED error.

By itself, this policy may not be very useful. This is because, the developer could have achieved the same effect by just not providing a GET request handler for a given API. This policy becomes incredibly useful however, when used in composition with other policies. For example, we may have an API function whose GET Method could be called by both Clinicians and Patients, but whose POST method can be called only by Clinicians. Such a Permission policy could be constructed using the following snippet.

```
ClinicianWriteableButPatientReadOnly = Or(
    IsClinician,
    And(IsPatient, IsReadOnlyMethod)
)
```

3.1.3.4 IsUserGroupMember

This policy ensures that a user is a member of the Group this API needs to access. This is done by checking whether the user is a member of the user group in the DB.

This Base Policy is very powerful when used in conjunction with other Policies such as IsClinician or IsPatient. For example, to allow only clinicians of a given Group instead of all clinicians, one could use

```
IsClinicianMemberOfGroup = And(
    IsClinician, IsUserGroupMember
)
```

3.1.3.5 ClinicianCanSeePatient

This policy guarantees that the clinician and the patient being referred to in the request are both registered, and both belong to **any** Group in common. This is one of the more complex base policies, and an ideal candidate for abstraction by PyMedServer.

Should a developer want to check that the clinician issuing the request and patient both belong to a specific Group, one could use

```
ClinicianAndPatientBelongToGroup = And(
    IsClinician,
    IsUserGroupMember,
    PatientIsGroupMember,
    ClinicianCanSeePatient,
)
```

3.2 Group Isolation

PyMedServer provides support for Group isolation at multiple layers of the framework - these are, the database, API and Permission Policies layers.

3.2.1 Database Support

In the Authentication suite provided by PyMedServer, one of the Base database tables offered is the Base User Group table. As with all other abstract base tables, this Base User Group table contains the minimum set of fields required for a functional User Group construct, a group name, and ManyToMany relations to hold members and admins.

3.2.2 API Support

PyMedServer provides APIs to help with Group management. Below is a brief description of some of those APIs.

3.2.2.1 Patient Registration

Clinicians use this API to register a new Patient with the server. This API requires that the clinician specifies the Group which this patient will be created in. It is important to note that this patient could subsequently be enrolled into more groups, as needed. Lastly, the permissions on this API check to make sure that the user issuing the request is a registered Clinician, and a member of the Group specified.

3.2.2.2 Add Patient To Group

This API is invoked by an Administrator or a Clinician, and is used to add a patient into a group. This is useful for authorized users who need to add a user into a group they do not belong to yet. This API is a no-op if the patient is already a member of the group in question.

3.2.2.3 Leave Group

For compliance reasons, it was important that all users be able to opt out of a group. This API removes a user from the group specified. They need to be

registered with us and be a member or an administrator of the group in question. This API is a no-op if the user is not already a part of the group in question.

3.2.3 Permission Layer Support

PyMedServer treats groups as first class components of the Authorization layer. This is done by providing a couple of base permission policies which allow developers to securely use groups within their APIs. Some of these base policies include:

- `IsUserGroupMember` - ensures the user is a member of the group specified in the request.
- `IsUserGroupAdmin` - ensures the user is an admin of the group specified in the request.
- `PatientIsUserGroupMember` - ensures that the patient specified in the request belongs to a given group.
- `MeasurementBelongsToUserGroup` - ensures that a diagnostic measurement was taken as part of a given group.

Like other permission policies, these policies are designed to be composable, so as to generate more comprehensive permissions. Developers are encouraged to contribute directly to PyMedServer if they need a custom permission which does not involve their particular plugins.

3.3 Ease of use

The main way in which developers interact with PyMedServer is via a single command line tool called Python Fabric [24]. This single entry point drastically simplifies the management overhead of working with PyMedServer. In subsequent sections, we will describe in some detail, some typical workflows developers enjoy while working on PyMedServer.

3.3.1 New Project Setup

From a fresh installation of Ubuntu 16.04+, setting up a new PyMedServer project requires only 2 commands: a first command to clone a dummy PyMedServer project, and a final command to install every system required to run the project.

```
$ git clone https://<location>/generic_pymed_server.git my_project
$ cd my_project && ./install.sh
```

The provided `install.sh` script will

- Install the python package manager
- Use python's package manager to install a virtual environment
    - It is important to note that all executables are hosted within a virtual environment so as to provide project isolation from other projects and from the rest of the system.
- Then finally, install every python dependency into the newly created virtual environment.
- Once all the dependencies are installed, the script invokes
    - `fab install_services`
        - This installs, configures, and starts the NGINX, Redis and PostgreSQL Servers.
        - It also generates a new `config.yaml` file with reasonable default parameter.
    - `fab create_postgres_db`
        - This configures a new PostgreSQL database.

- `fab migrate_db`
  - The new database which was created is empty, and has no schema for our dummy project yet. This ensures that the database is migrated, and ready to receive connections from our backend replicas, as well as our background job runners.

Once `install.sh` completes, the new project will be fully functional and ready for tests and plugin addition.

3.3.2 Running the server

To start the server, developers run `` `fab start` ``, and to disable and stop the server, developers run `` `fab stop` ``. Starting an already started server will restart the server, while stopping an already stopped server will essentially be a no-op. Start and Stop were both to perform all operations required to bring the server to the desired end state.

3.3.2.1 fab start

This command is designed to be easy to remember and invoke. Under the hood however, this command is very complex. Below are a couple of operations which this command performs to ensure the PyMedServer is started:

- It generates the proper NGINX configs, such that NGINX is aware of this project's domain and backend point.
- It enables this project's domain name within NGINX.
- It generates the proper Supervisor configs, to ensure that required jobs are declared.
- It kicks of Supervisor - which in turn starts and monitors Gunicorn and the Backend Job Masters.
- It prepares NGINX's log files for the current project.
- It reboots NGINX so it picks up the new configuration changes.

- It ensures that all static files which NGINX will have to serve are accessible by NGINX.
- ...

A lot of thought went into this command to ensure that it starts the server in as many complex situations as possible, all in the aim of keeping its external interface simple.

3.3.2.2 fab stop

This command is used to stop a PyMedServer. Like is the case with `fab start`, `fab stop` is also designed to be simple to remember and to invoke, but is also very complex under the hood. Besides turning down Supervisor, it performs a couple of cleanup operations to ensure that other projects are not impacted by this current project stopping.

3.3.3 Database Management

Developers need to run `fab migrate_db` every time they make changes to the Models layer of the database. This command will use Django's built-in migration framework to ensure that db migration files are created. These migration files are required to keep the db at a consistent state with the Models layer of Django.

In addition to bringing the db up to date with the Models layer, this command also generates a visualization of the database objects. Below is an example of the type of visualization generated by this command. This is done using a django library called `django-extensions` [25].

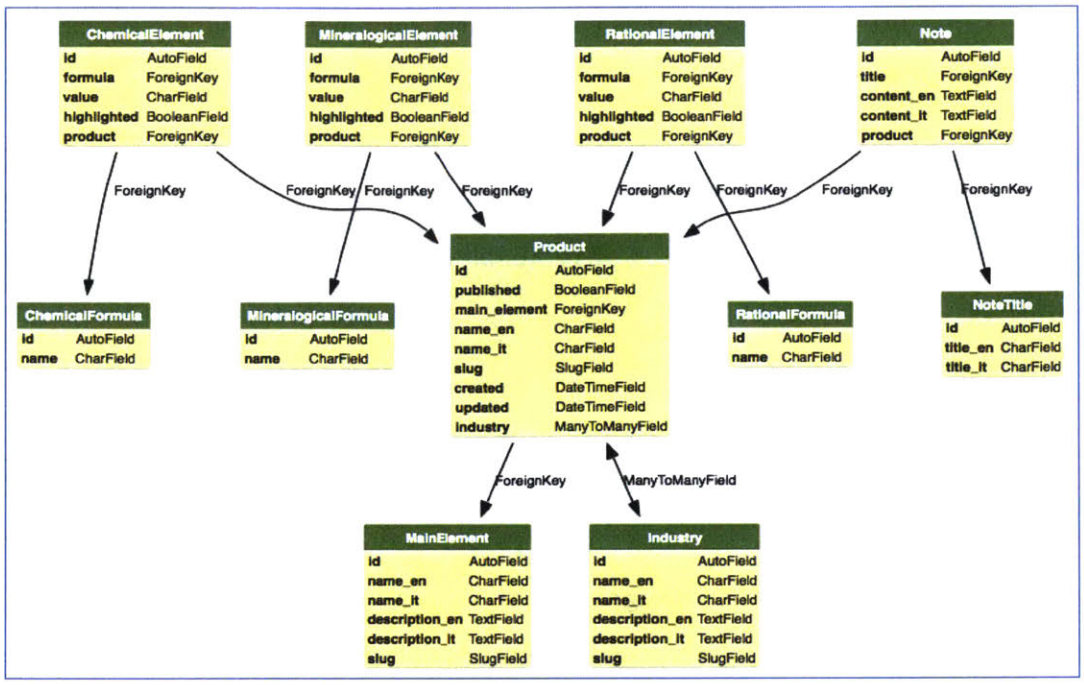

Figure 6 - Sample DB visualization image generated by fab migrate_db.

These diagrams are compact representations of the database, and easy to share amongst students, their supervisors, and other interested parties.

3.4 MAINTENANCE

The default project created by the setup phase is very well written and thoroughly document, with step by step instructions on how one a developer could get started implementing a new feature. The first step of this process is to delete the dummy plugin which comes packaged in the default project, and replace it with a production plugin. From the moment developers start contributing to the project, PyMedServer provides tools to help keep their project's code style and unit test coverage standards high.

3.4.1 Unit Tests

The `fab test` command scans the entire project and runs all test files it finds. By convention, tests file names all start with `test_`, and the files are

usually located within a module named `tests`. `fab test` can be provided a module path filter, which it will use to filter the tests it will discover and run.

PyMedServer also provides a Base Test Class, which contains a lot of test utilities to make sure that unit tests are properly isolated from each other. The Django test framework is used, in addition to the popular python testing framework `mock` and `unittest`, which PyMedServer packages with the default project.

Finally, `fab test` dumps its output into a file called `test_output.txt` located at the project root. It is expected that this file is checked-in into git along with other project files. This is to ensure that a record of all executed tests are available in the git history for debugging.

3.4.2 Code Quality

PyMedServer provides the `fab lint` command, which formats all Python files within the project, to ensure that all files follow a common style. The formatting is done in accordance with the Google Style Guide for Python [26], using a custom tool, which uses Yapf [27], Docformatter [28] and Isort [29]. Isort is used to sort import statements, Docformatter is used to format all python documentations in a file, and finally, Yapf is used for formatting the rest of the file.

3.5 Built-in Front-end

PyMedServer provides a front-end for common efforts which will otherwise be tedious to perform on phone screens. This built-in front-end, like most other built-in features offered by PyMedServer, is an opt-in feature and is enabled if the developer leveraged the built-in Authentication suite provided by the framework.

Through this portal, Clinicians and Administration can export measurements into CSV files, which they can use for plots, or other as input to other applications.

This concludes the Technical Design chapter of PyMedServer. In the next chapter, we will explore a few projects in which PyMedServer is used in production.

4 PyMedServer Implementation Examples

The Mobile Technology Group, under the direction of Dr. Rich Fletcher, has currently adopted the PyMedServer platform for several of its ongoing research studies. In this chapter, four different projects where PyMedServer is used will be described.

4.1 Cardio-Vascular Study Server

This server assists the Cardio team in collecting diagnostic measurements through various medical devices in order to assist in screening for cardiovascular diseases. Similar to the Diabetes Study server, the main clients to this server are Android applications. This server also has a Raspberry PI as client for some of the non-invasive measurement collections which are performed as part of this study.

4.1.1 Server Measurement Types

Below is a table describing some of the measurement types stored on this PyMedServer.

DB Measurement Type	Source	Description
`Framingham Questionnaire`	Android App	Stores answers provided by patients to a Framingham Questionnaire. Mainly numbers, text, floats, and booleans.
`PPG-Camera`	Android App	The Android application performs some analysis on using the built-in camera, and stores those measurements locally as CSV files. These files are uploaded to the server on user request.

4.1.2 Background Jobs Configured

Currently, this PyMedServer is configured with two background jobs. These background jobs are triggered when a client calls the `run_analysis` API on the server. This API takes as parameters, a set of Measurement IDs, which the background job runner will fetch from the db and use as bases for the analyses. Once the analysis is completed, the analysis results are stored to the database. The Clients repeated poll the server, using the `view_analysis` api, till the Server response with valid analysis results or an error message.

4.2 Diabetes Study Server

This server assists the Diabetes team with the collection and analysis of diagnostic measurements from patients, to help with the screening of Diabetes. The main clients used here are Android Applications. Each application was designed to collect a specific type of measurement, and synchronize them with the server once access to internet is restored.

4.2.1 Server Measurement Types

Below is a table describing some of the Measurement types configured on the server, as well as the Apps responsible for syncing them to the server.

DB Measurement Type	Source	Description
`Diabetes Questionnaire`	Android - Diabetes Questionnaire [30]	Stores answers provided by patients to a standard diabetes questionnaire. Mainly numbers, text, floats, and booleans.
`Perseverative Thinking Questionnaire`	Android - Perseverative Thinking Questionnaire	Stores answers provided by patients to a standard PTQ questionnaire. Mainly numbers, text, floats, and booleans.
`Blood Test Measurement`	Android - Blood Test App	Patient uses a blood glucose measurement tool, which measures their blood glucose levels. The readings are entered into the app and uploaded to the server. On the database, these readings are mainly stored as integer values.
`Thermal Scanner`	Android - Diabetes Thermal Screener	Patients are scanned using a thermal camera attached to the clinician's smartphone. The resulting image capture is uploaded to the server.

4.2.2 Background Jobs Configured

Currently, there are three different Machine Learning background tasks available on this server. These are triggered when a clinician calls the `run_analysis` API. Clinicians access this feature via an `Analyze` button

provided in the Diabetes Screener Android application. Similar to the Cardio Study Server, the clients retrieve these analysis results via Polling.

4.3 Mental Health Study Server

This server assists the Mental Health team by providing a platform for collecting various mood and habits related information from participating users. The main data source to this server is an Android App. This app records some signals from participant's android phones, stores them locally, and periodically uploads them to the server.

4.3.1 Server Measurement Types

These are some of the measurement types configured on the server for this project.

- Gyroscope and Accelerometer readings:
    - These are stored as time series on the Server - that is, a table containing, the timestamp, the x-axis, the y-axis, and the z-axis reading per sample.
    - The sampling rate on the phones we tested is about 400Hz, as a result, the recording session is limited to just one minute. During that minute, all the recordings are stored in-memory, to minimize latency. After the recording minute is over, the app then persists the recordings into a session entry on the phone's local storage.
    - Periodically, these readings are uploaded to the server.
- Frequency of unlocks:
    - Every time the phone is unlocked, the app records the event in the phone's local storage.

- Periodically, along with all other readings, these unlock events are exported to the server.

This concludes our chapter on the various PyMedServer installations deployed to date. In the next chapter, we will discuss the design of PyMedServer's Android Client.

5 THE PYMEDSERVER-ANDROID LIBRARY

In this chapter, I will describe the design of PyMedServer-Android, an Android library which I created to streamline and simplify the interface between Android and our PyMedServers.

5.1 MOTIVATION

One of the main advantages which PyMedServer provides, is a streamlined API surface across all our various cloud projects. PyMedServer is predominantly a JSON server, so we could invest in building clients with this fact in mind. Moreover, because PyMedServer does Authentication the same way across all cloud projects, we had the opportunity to librarize away the bulk of the work required to support Authentication on our various client classes (android, web, and python client classes).

PyMedServer-Android does exactly this – it provides a generic API interface pattern towards a PyMedServer, and comes pre-packaged with Authentication primitives that are compatible with those on a PyMedServer.

In addition to Authentication, PyMedServer-Android was also designed to support online-offline synchronization, and full offline support. By full offline support, we mean that the library is perfectly capable of working in offline conditions, and eventually syncing with a cloud server when internet becomes available.

PyMedServer-Android is divided into two sub libraries, APILib, and StorageLib. APILib hosts all API constructs required to safely issue APIs towards a given PyMedServer. StorageLib hosts all DB constructs to store resource details offline, while they are pending synchronization with the cloud.

5.2 APILIB

APILib is an Android library which contains utilities required to issue HTTPS API requests towards a PyMedServer. To best understand how these utilities fit together, we describe in the next section, how APIs are made.

5.2.1 How APIs are made – API Dispatcher

APIs on PyMedServer are HTTPS JSON APIs. In order words, the client requests need to be HTTPS, with JSON request bodies, and responses from the server will possess JSON bodies as well.

APILib provides an API Dispatcher, which serializes a Request Object (a Java Object) to its JSON form, then invokes an HTTPS request towards the desired server API URL, then parses the resulting response from the server from JSON into a Response Object (also a Java Object).

To serialize Java objects to JSON strings and deserialize JSON strings back to Java objects, the Dispatcher uses the GSON library [31]. To perform HTTPS requests, the OkHttp library [32] is used.

The Dispatcher is templatized, and requires developers to specify two classes, a `RequestType` class, and a `ResponseType` class. These types are used by the GSON library for serialization and deserialization.

The fact that all requests and responses are represented by Java classes is a very important concept. This is because it helps with code readability, input sanitization, and expandability. Expandability here refers to the ability of other frameworks to use these classes for their own purposes. In fact, StorageLib, which will be discussed shortly, relies heavily on these class definitions.

5.2.2 How File Uploads Are Handled

Because APILib is all JSON based, it may not be immediately clear how file uploads are handled. PyMedServer expects files to be sent as Base64 encoded strings. As a result, files become strings, which is one of the JSON primitives.

APILib provides a utility function which takes a file and converts it to Base64, respecting the format the server requires.

5.2.3 Built-in Authentication

Because APILib was designed for PyMedServer, and because all PyMedServers perform Authentication in almost identical ways, APILib provides all Authentication APIs our Android clients require.

One of the implications of the previous statement is that all Authentication types required for those APIs are provided by the library as Java classes. This is a fantastic bonus, as it means even more libraries, like StorageLib which we will see soon, can be built against on these classes – essentially providing very strong defaults, in lieu of more templating.

Lastly, APILib is fully aware of PyMedServer's Authentication schemes. For instance, Login and Logout are special APIs, as they permit a user to retrieve or expire an access token. As a result, when a developer uses APILib's login and logout APIs, the library takes care of resetting or invalidating the stored user token automatically.

5.3 StorageLib

StorageLib is an android library which hosts storage utilities and base activities for our Android applications. One can think of this as the Database and the UI layer of PyMedServer-Android. In subsequent sections, we will describe some of the utilities which this library provides.

5.3.1 Databases and Query Builders

We want to maintain a high degree of flexibility on the servers, without having to worry about migrating these Android databases every time the servers are updated. As a result, the decision was taken to store all server objects in the android databases as JSON serialized objects. Recall that APILib already imposes the rule that all sever responses need to be described by Java classes. StorageLib

builds on top of this rule, and hence can store all server resources into the DB as JSON strings directly.

The database system we use in StorageLib is Android Room [33], which is powered by SQLite. Most tables are configured with just 5 columns:

- LocalID – This is the primary key in the local database, and is an autoincrementing integer.
- ServerID – This is the server's ID for this resource. This is a 64-bit integer.
- LocalResource – This is a JSON object representing the local version of this resource.
- ServerResource – This is a JSON object representing the server version of this resource.
- MetaData – This is a JSON object representing meta information about this resource. In the case of the Measurement tables for example, some fields which this object stores are: the type of measurement this row contains, whether the machine learning labels for the measurement were updated locally or not, etc.

5.3.1.1 Query Builder

Because of our decision to store resources in the database as JSON objects, we could not find any open source libraries to help us query JSON strings in SQLITE. As a result, we built a 600 lines-of-code custom query builder package for SQLITE syntax with JSON support. Equipped with this package, we could construct very complex custom queries on the fly, which was ideal for some of our validators and finders. For instance, this piece of code generates a WHERE Sql query which could be used to find a clinician in the database by various available identifiers.:

```
private WhereClause
findMatchingClinician(ClinicianProfileDbRowInfo cInfo) {
  List<Clause> clauses = new ArrayList<>();
  // Find clinician from server id
  if (cInfo.getServerId() != null) {
    clauses.add(
      Eq(SERVER_ID_COL, cInfo.getServerId())
    );
  }
  // Find clinician by user name.
  if (cInfo.getUsername() != null) {
    clauses.add(
      JsonLike(LOCAL_DATA_COL, "username", cInfo.getUsername())
    );
    clauses.add(
      JsonLike(SERVER_DATA_COL, "username", cInfo.getUsername())
    );
  }
  // Find clinician by email
  if (cInfo.getEmail() != null) {
    clauses.add(
      JsonLike(LOCAL_DATA_COL, "email", cInfo.getEmail())
    );
    clauses.add(
      JsonLike(SERVER_DATA_COL, "email", cInfo.getEmail())
    );
  }
  // Find clinician by phone number.
  if (cInfo.getPhoneNumber() != null) {
    clauses.add(
      JsonLike(LOCAL_DATA_COL, "phone_number", cInfo.getPhoneNumber())
    );
    clauses.add(
      JsonLike(SERVER_DATA_COL, "phone_number", cInfo.getPhoneNumber())
    );
  }
  // Join all with an OR, and exit.
  return Where(Or(clauses));
}
```

Figure 7 - Sample Query with JSON support

In the figure above, the package provides mainly provides the **JsonLike** Query matcher, which allows us to search within Json strings.

5.3.2 Built-in UIs

One of the fundamental design goals for StorageLib, was to be as simple to use as possible. Returning to our plugin/feature model, this meant that libraries should do as much as possible to ensure that feature developers worry solely about the specific feature they need to build. More concretely, this meant that the libraries should take care of Authentication, not only at the API level, but also at the UI level.

StorageLib achieves this through Abstract Inheritance. For all UI pages where no input is required from the developer, StorageLib provides Activities which developers can invoke directly. However, some UI pages require app-specific features. For example, the Landing Page of an Application requires a custom logo, and a custom display title, which the library has no way of knowing ahead of time. As a result, the library provides the bulk of the UI, but uses Abstract Methods so the developer could provide those custom pieces. In the Landing page example, the developer subclasses a Base Landing Activity, and provides a GetLogo method, which the library calls to display the appropriate logo on the page. Below is a figure illustrating in greater details how this works.

Activity Hierarchy

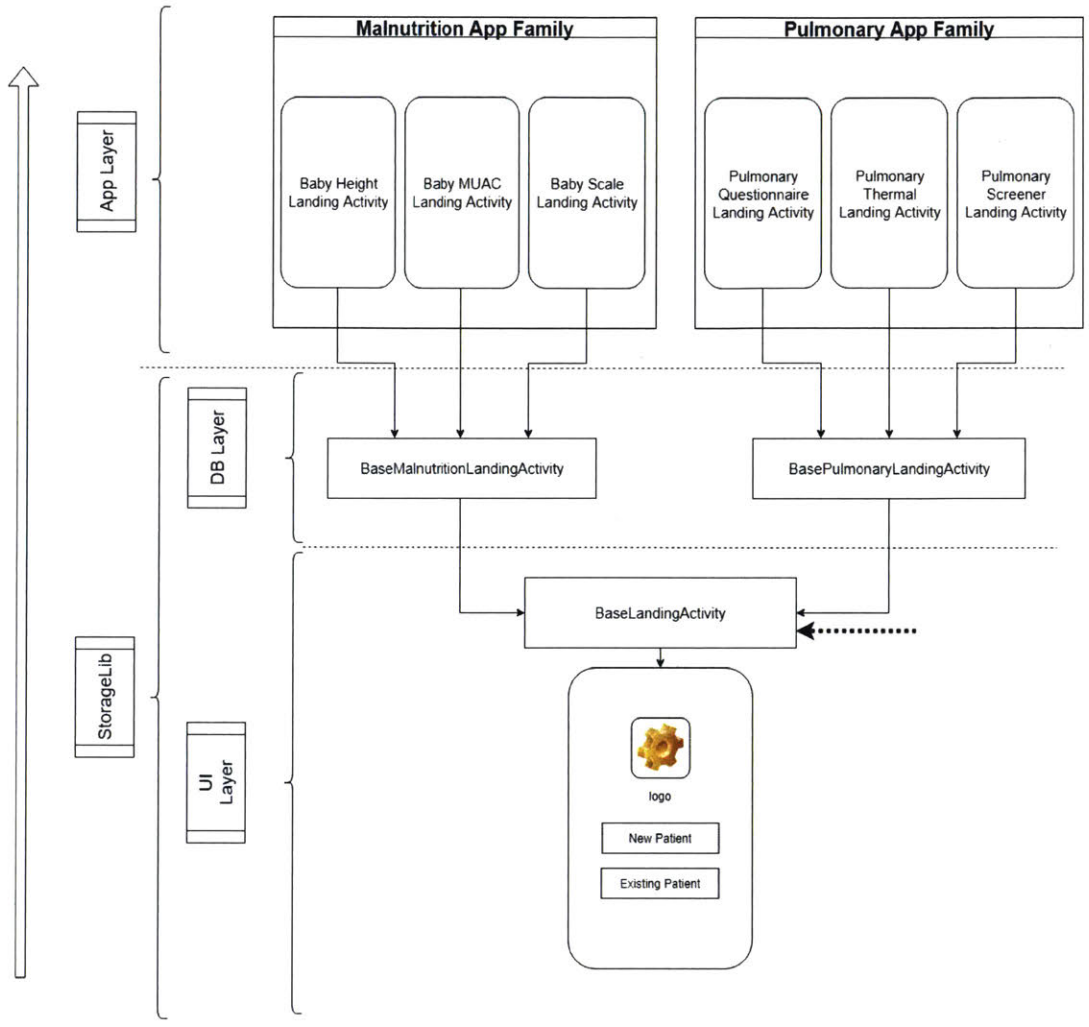

Figure 8 - Activity Hierarchy

5.3.2.1 UI Layer

This is the layer in which StorageLib takes care of configuring and displaying an Android Activity. This layer is not aware of Application specific details such as, which the database needs to be used, or which app provided resources needs to be displayed etc. This layer therefore provides abstract methods which application developers implement.

5.3.2.2 Db Layer

This layer provides database information to the UI layer below it. Because various projects use different databases, there are typically several specializations of a given UI layer base activity, on this layer. For instance, if the Malnutrition, Pulmonary, and Diabetes projects need Landing Pages, there will be three different specializations of the Base Landing Activity provided by the UI layer. Each specialization will provide a different DB to the UI layer. For example, the BaseMalnutritionLandingActivity will provide a Malnutrition Database to the UI layer, the BasePulmonaryLandingActivity will provide a Pulmonary Database to the UI layer, and so on.

It is important to note here that this layer is not usually the final. In particular, this layer itself still resides within StorageLib, so cannot know ahead of time, application specific details which the UI layer requires. For instance, in the case of the Landing activity, the BasePulmonaryLandingActivity will be able to provide the DB information to the UI layer, but will not be able to provide the Logo information. As a result, Activities in this layer are also mostly abstract as well.

5.3.2.3 Application Layer

This is the top most layer, and lives in the application space, outside of StorageLib. Activities in this layer are no longer abstract. At this layer, ever remaining missing piece which could not be filled by the DB layer will need to be provided by the developer. In the case of the Landing Activity for example, this will be the Logo information as well as more requirements not shown here for brevity.

Below are images of some of the landing page of some of our applications. Notice how the structure is similar in all of them.

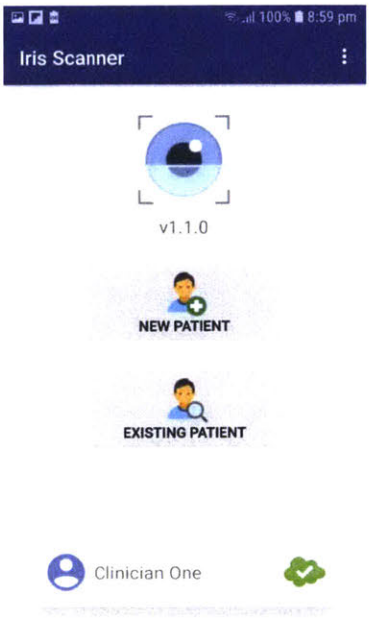

Figure 9 - Iris Scanner Landing Page

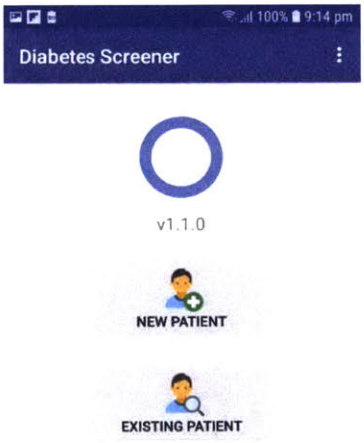

Figure 10 - Diabetes Screener Landing Page

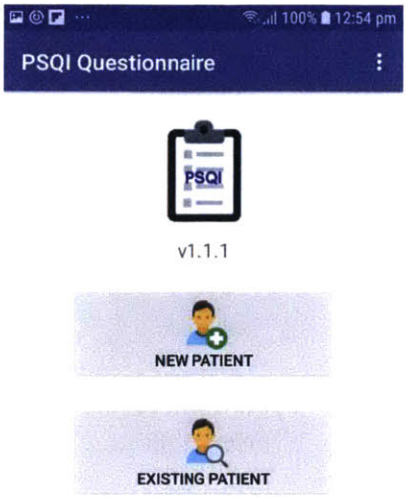

Figure 11 - Sleep Questionnaire Landing Page

5.3.2.4 Other Built-in Activities

Besides Landing Pages, StorageLib also offers a variety of other activities.

5.3.2.4.1 Patient Select

This activity allows clinicians to select the patient they are currently examining. From this activity, the clinician could also register new patients, or synchronize patient profiles with the cloud.

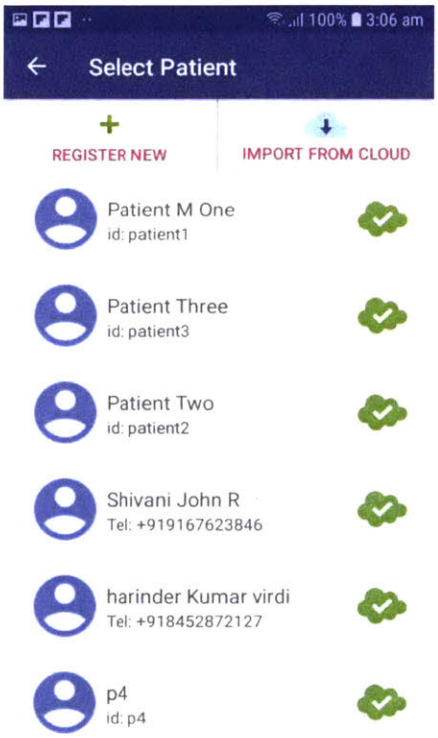

Figure 12 - Patient Select Example

5.3.2.4.2 Patient Home

After a patient has been selected, the clinician is presented with this activity. From this Patient Home Activity, the clinician can kick-off an exam, view the history of this patient's measurements, diagnosis, or analyses.

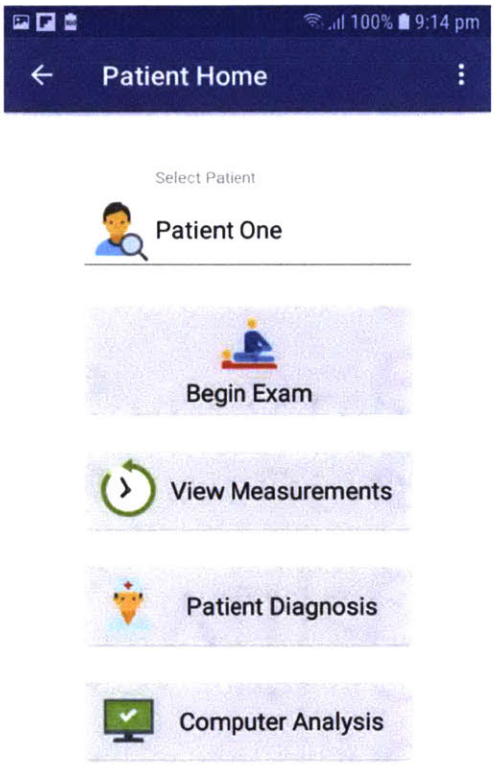

Figure 13 - Patient Home

5.3.2.4.3 View Measurements

Through this activity, the clinician will be able to inspect the history of measurements collected for this patient. The clinician will also have the ability to synchronize all the patient's measurements with the PyMedServer.

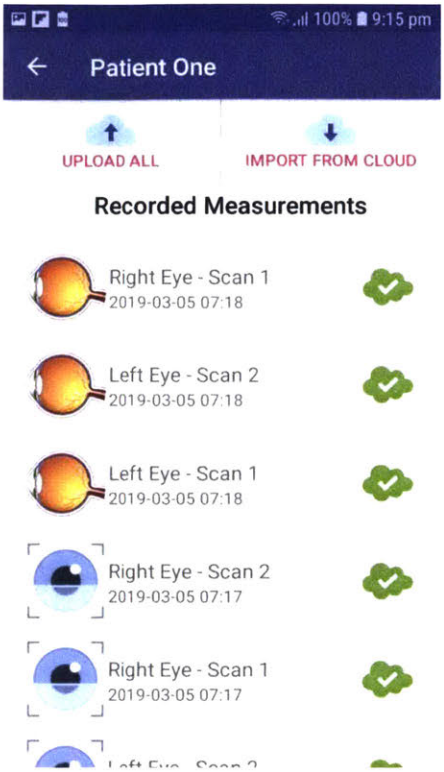

Figure 14 - View Measurements Activity

5.3.2.4.4 Clinician Diagnosis

This activity permits clinicians to enter diagnoses for a selected Patient. This page also enables the clinician to synchronize the local diagnoses with a cloud PyMedServer.

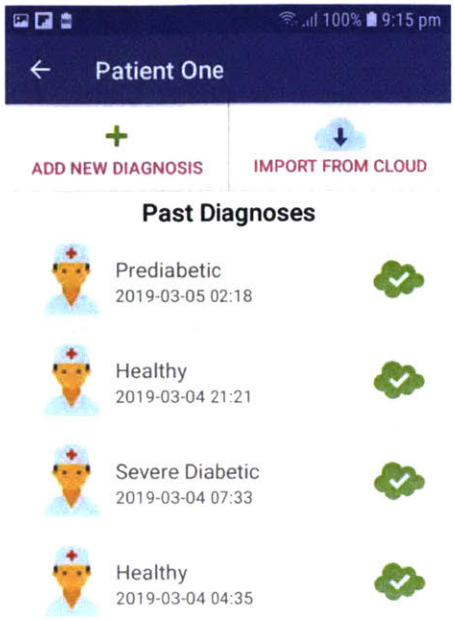

Figure 15 - Clinician Diagnosis Activity

5.4 Online-Offline Synchronization Support

As previously stated in the Database Design section of StorageLib, every resource stored in the local android database can have two versions – a local version, and a server version. The local version of each object contains enough information to permit an eventual server sync. The server version contains a read-only version of the resource as present on the server at some point in the past.

Whenever there is a discrepancy between the local version and the server version of a given resource, a red cloud is displayed on the element, and a user is

prompted to synchronize with the server. This is the basis of how PyMedServer and PyMedServer-Android achieve online-offline synchronization.

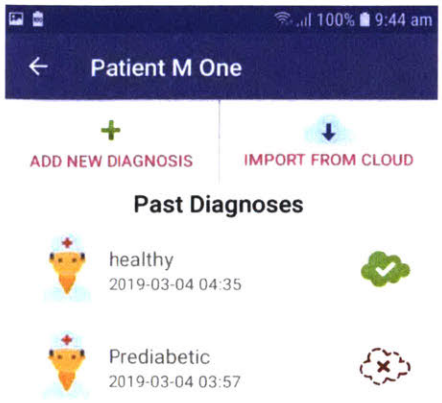

Figure 16 - Example un-synced Patient Diagnosis

6 CONCLUSION

PyMedServer is quite a promising framework, and was designed with the intention of withstanding the test of time. Within just the last year, it has been of tremendous help to the Mobile Technology Group at MIT as it has been used to power a lot of our ambitious projects, that have sometimes been under tight deadlines. Because it was built with support for Machine Learning and Background Jobs in mind, it has opened the doors to a lot of opportunities which were previously more difficult for our Group to attain. It has enabled our Group to think differently about data collection and cloud integration, by abstracting project setup and data management concerns away from plugin developers, hence redirecting the focus on the student projects themselves.

Our hope is that PyMedServer will expand to beyond just our Group, and help benefit more research groups at MIT, and perhaps across the world.

7 BIBLIOGRAPHY

[1] "Open Data Kit," [Online]. Available: https://opendatakit.org.

[2] D. Inc, "CommCare by Dimagi | Data Collection App," [Online]. Available: https://www.dimagi.com/commcare/.

[3] Amazon, "Amazon EMR - Amazon Web Services," [Online]. Available: https://aws.amazon.com/emr/.

[4] "OpenEMR," [Online]. Available: https://www.open-emr.org/.

[5] "OpenMRS," [Online]. Available: https://openmrs.org/.

[6] I. Sysoev, "NGINX | High Performance Load Balancer, Web Server; Reverse Proxy," [Online]. Available: https://www.nginx.com.

[7] Oracle, "PostgreSQL: The world's most advanced open source database," [Online]. Available: https://www.postgresql.org/.

[8] S. Sanfilippo, "Redis," [Online]. Available: https://redis.io/.

[9] "NGINX Reverse Proxy," [Online]. Available: https://docs.nginx.com/nginx/admin-guide/web-server/reverse-proxy/.

[10] "Deploying static files," [Online]. Available: https://docs.djangoproject.com/en/2.2/howto/static-files/deployment/.

[11] "NGINX SSL Termination," [Online]. Available: https://docs.nginx.com/nginx/admin-guide/security-controls/terminating-ssl-http/.

[12] "The Web framework for perfectionists with deadlines | Django," [Online]. Available: https://www.djangoproject.com/.

[13] NetGuru, "Top 10 Django Apps and Why Companies Are Betting on This Framework," [Online]. Available: https://www.netguru.com/blog/top-10-django-apps-and-why-companies-are-betting-on-this-framework.

[14] "Archive of security issues | Django," [Online]. Available: https://docs.djangoproject.com/en/2.2/releases/security/.

[15] "Settngs | Django," [Online]. Available: https://docs.djangoproject.com/en/2.2/ref/settings/.

[16] "Models | Django," [Online]. Available: https://docs.djangoproject.com/en/2.2/topics/db/models/.

[17] T. Christie, "Django REST framework," [Online]. Available: https://www.django-rest-framework.org/.

[18] "Serializers - Django REST framework," [Online]. Available: https://www.django-rest-framework.org/api-guide/serializers/.

[19] "Gunicorn 'Green Unicorn'," [Online]. Available: https://gunicorn.org/.

[20] "Celery: Distributed Task Queue," [Online]. Available: http://www.celeryproject.org/.

[21] RedisLabs, "6.4.1 First-in, first-out queues | Redis," [Online]. Available: https://redislabs.com/ebook/part-2-core-concepts/chapter-6-application-components-in-redis/6-4-task-queues/6-4-1-first-in-first-out-queues/.

[22] "django-cachalot," [Online]. Available: https://django-cachalot.readthedocs.io/en/latest/introduction.html.

[23] "Supervisor: A Process Control System," [Online]. Available: http://supervisord.org/.

[24] "Python Fabric," [Online]. Available: http://www.fabfile.org/.

[25] "Django Extensions," [Online]. Available: https://django-extensions.readthedocs.io/en/latest/.

[26] Google, "Google Python Style Guide," [Online]. Available: http://google.github.io/styleguide/pyguide.html.

[27] Google, "Yet Another Python Formatter | YAPF," [Online]. Available: https://github.com/google/yapf.

[28] S. Myint, "Docformatter," [Online]. Available: https://pypi.org/project/docformatter/.

[29] T. E. Crosley, "Isort | A Python utility / library to sort imports.," [Online]. Available: https://github.com/timothycrosley/isort.

[30] M. T. Lab, "Diabetes Questionnaire," [Online]. Available: https://play.google.com/store/apps/details?id=com.mobiletechnologylab.diabetes_questionnaire.

[31] Google, " Java serialization/deserialization library to convert Java Objects into JSON and back," [Online]. Available: https://github.com/google/gson.

[32] Square, "OkHttp," [Online]. Available: https://square.github.io/okhttp/.

[33] "Room Persistence Library | Android Developers," [Online]. Available: https://developer.android.com/topic/libraries/architecture/room.