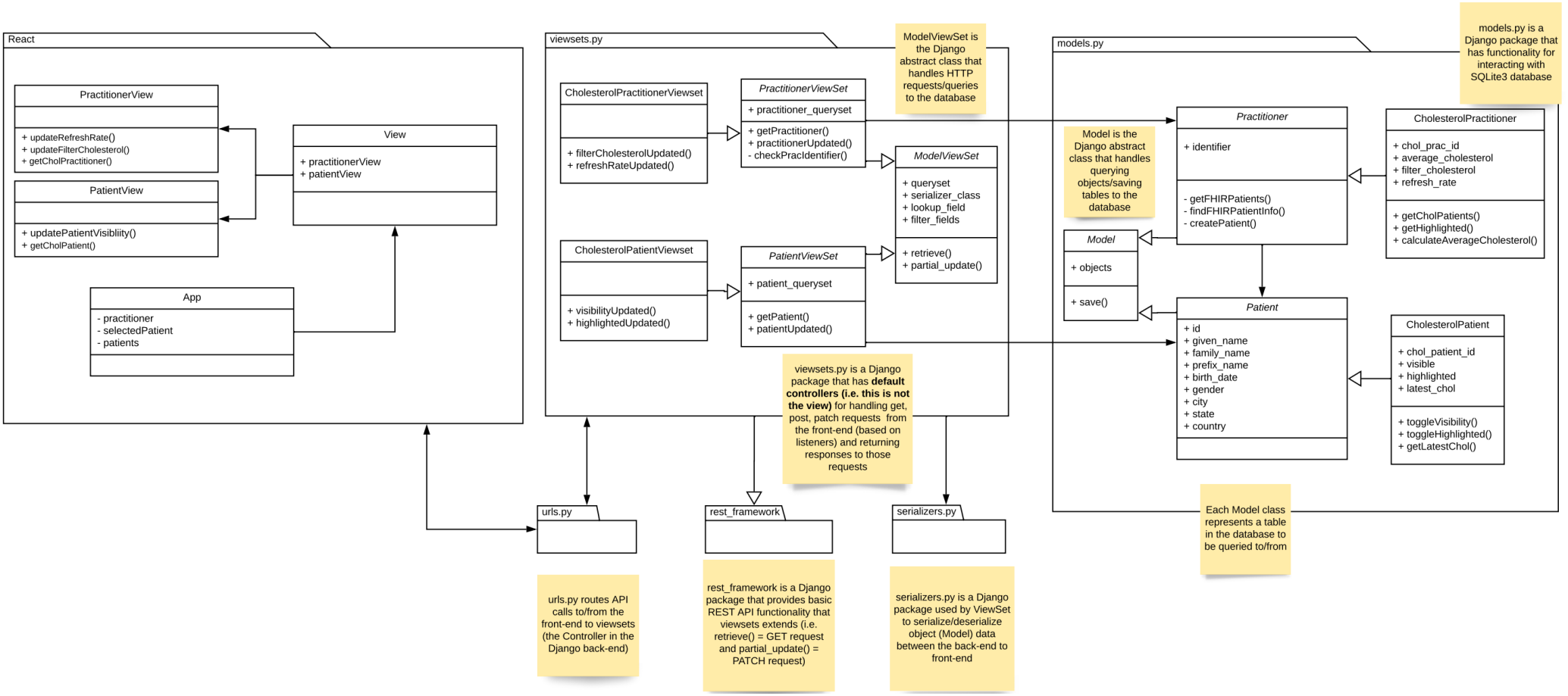
React
PractitionerView
+ updateRefreshRate()
+ updateFilterCholesterol()
+ getCholPractitioner()
PatientView
+ updatePatientVisibliity()
+ getCholPatient()
View
+ practitionerView
+ patientView
App
- practitioner
- selectedPatient
- patients
viewsets.py
ModelViewSet is the Django abstract class that handles HTTP requests/queries to the database
CholesterolPractitionerViewset
+ filterCholesterolUpdated()
+ refreshRateUpdated()
PractitionerViewSet
+ practitioner_queryset
+ getPractitioner()
+ practitionerUpdated()
- checkPracIdentifier()
ModelViewSet
+ queryset
+ serializer_class
+ lookup_field
+ filter_fields
+ retrieve()
+ partial_update()
CholesterolPatientViewset
+ visibilityUpdated()
+ highlightedUpdated()
PatientViewSet
+ patient_queryset
+ getPatient()
+ patientUpdated()
viewsets.py is a Django package that has default controllers (i.e. this is not the view) for handling get, post, patch requests from the front-end (based on listeners) and returning responses to those requests
models.py
models.py is a Django package that has functionality for interacting with SQLite3 database
Model is the Django abstract class that handles querying objects/saving tables to the database
Practitioner
+ identifier
- getFHIRPatients()
- findFHIRPatientInfo()
- createPatient()
CholesterolPractitioner
+ chol_prac_id
+ average_cholesterol
+ filter_cholesterol
+ refresh_rate
+ getCholPatients()
+ getHighlighted()
+ calculateAverageCholesterol()
Model
+ objects
+ save()
Patient
+ id
+ given_name
+ family_name
+ prefix_name
+ birth_date
+ gender
+ city
+ state
+ country
CholesterolPatient
+ chol_patient_id
+ visible
+ highlighted
+ latest_chol
+ toggleVisibility()
+ toggleHighlighted()
+ getLatestChol()
Each Model class represents a table in the database to be queried to/from
urls.py
urls.py routes API calls to/from the front-end to viewsets (the Controller in the Django back-end)
rest_framework
rest_framework is a Django package that provides basic REST API functionality that viewsets extends (i.e. retrieve() = GET request and partial_update() = PATCH request)
serializers.py
serializers.py is a Django package used by ViewSet to serialize/deserialize object (Model) data between the back-end to front-end

Design rationale

The system was designed as a full-stack web application using ReactJS for the front-end, Django (Python framework) for the back-end, and an SQLite3 database.

The general architecture of the system consisted of three main components based on the *MVC design pattern*; a Model (represented by models.py), View (represented by the React package) and a Controller (represented by viewsets.py). It was done this way to clearly separate the dataset/querying, GUI and logic of the program from one another. In this way, team members were able to work in parallel as well as minimising coupling between the various components of the program.

Within each package, existing components of the Django framework were extended to form application-specific (abstract) classes. For instance, the Django ViewSet abstract class (in combination with Django's rest_framework and serializers packages) was extended into specific PractitionerViewSet and PatientViewSet interfaces for the program.
This creation of abstract classes, conforming to the *Open-Closed* design principle, meant every component of the system was available/open to other components via a stable interface that protected its underlying implementation. The use of abstraction (and inheritance) also allowed extensibility of new functionality in future (as CholesterolPractitionerViewSet extends PractitionerViewSet to add specific cholesterol handling to the controller).

Since other dependent components/classes only know about the interfaces rather than implementation details, they serve as hinge points to add new functionality, thereby upholding the *Dependency Inversion* design principle.
One notable aspect of the program's requirements was the flexibility needed for monitoring patients and their data in future. This necessitated the separation of interfaces for Patients and Practitioners. Based on the *Interface Segregation Principle*, this effectively allows patient models and data presentation to change independently of practitioners if need be.

When designing our system, it was imperative we had proper version control so that we could use the *Release Reuse Equivalence Principle*. Thus, every package has had commits tracked based on a version control system that is publicly available - so users can view updates and determine whether they are essential for individual packages (e.g. for an updated Model, View or Controller package). In addition, by utilizing the *Common Closure Principle*, we were able to control the number of changes needed to the whole system when we had to fix a bug somewhere. In the project's case, classes were organised into packages based on their role in the MVC design/architectural pattern because they frequently change together. From this, when we needed to fix a bug, it was just related to a single package. This saves much time as when fixing a bug doesn't spawn multiple other bugs.

Our packages were designed with abstract interfaces at the fore-front to conform to the *Stable Abstractions Principle*; packages only access/know about each other through very few interfaces that other functionality then extends from, this, in turn, makes packages stable, easy to extend but hard to change. This is because, if you make a change in an unstable package, it can have side-effects on a stable one. Therefore if a package is only depending on a more stable implementation then, making a change will not affect the less stable one.

Regardless, the design does fall short in some ways.

In order to conform to the Release Reuse Equivalence and Common Closure Principles, packages had to be made relatively large. This makes them well-suited for maintenance (as maintaining one package is easier than maintaining several) but may not suit clients who need to reuse/upgrade packages in very particular use-cases (and prefer to download smaller packages). In addition, the design only incorporates a fixed set of entities that serve as hinge points. Were more entities needing to be added (such as different Hospitals or Nurses), extension of design elements would be far more tedious and a design revamp would potentially be needed. Also, due to the structure of a Django-based web application, the design does not conform to a true MVC architecture. Rather, the View cannot directly get or update the Model; it does so via the controller (urls.py and viewsets.py).