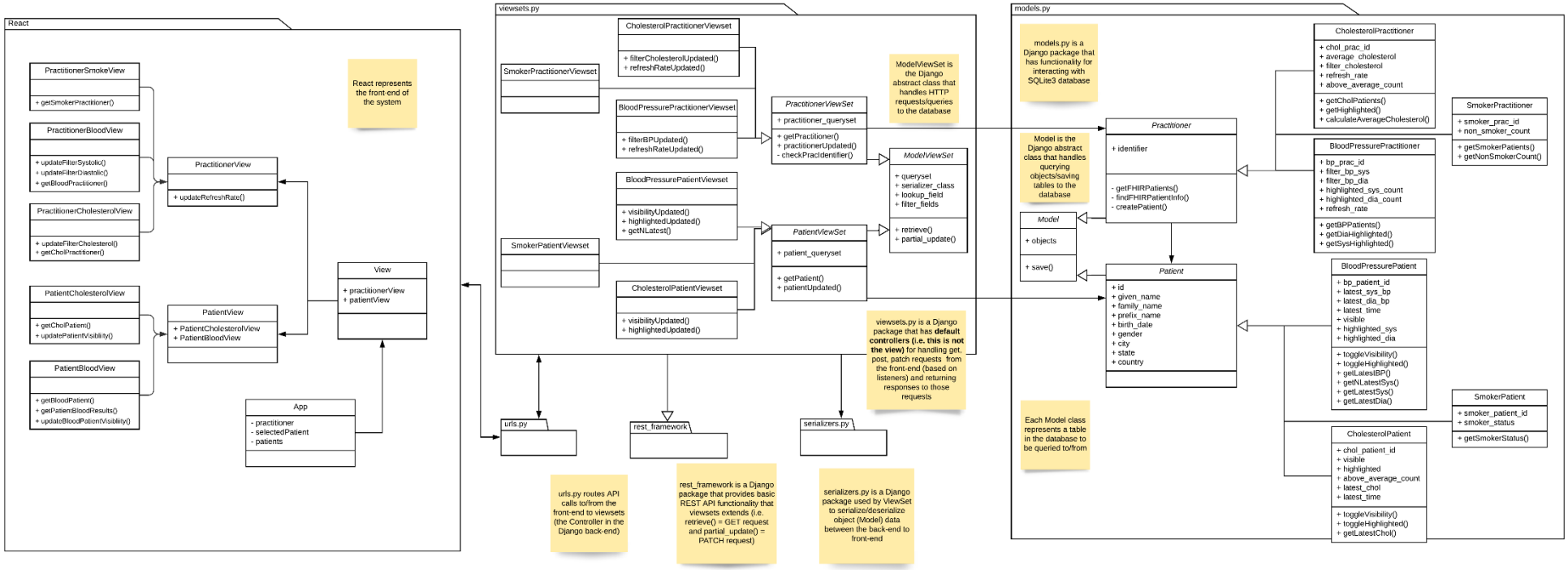
React
PractitionerSmokeView
+ getSmokerPractitioner()
PractitionerBloodView
+ updateFilterSystolic()
+ updateFilterDiastolic()
+ getBloodPractitioner()
PractitionerCholesterolView
+ updateFilterCholesterol()
+ getCholPractitioner()
PatientCholesterolView
+ getCholPatient()
+ updatePatientVisibility()
PatientBloodView
+ getBloodPatient()
+ getPatientBloodResults()
+ updateBloodPatientVisibility()
PractitionerView
+ updateRefreshRate()
PatientView
+ PatientCholesterolView
+ PatientBloodView
View
+ practitionerView
+ patientView
App
- practitioner
- selectedPatient
- patients
React represents the front-end of the system
viewsets.py
SmokerPractitionerViewset
CholesterolPractitionerViewset
+ filterCholesterolUpdated()
+ refreshRateUpdated()
BloodPressurePractitionerViewset
+ filterBPUpdated()
+ refreshRateUpdated()
PractitionerViewSet
+ practitioner_queryset
+ getPractitioner()
+ practitionerUpdated()
- checkPracIdentifier()
ModelViewSet
+ queryset
+ serializer_class
+ lookup_field
+ filter_fields
+ retrieve()
+ partial_update()
ModelViewSet is the Django abstract class that handles HTTP requests/queries to the database
SmokerPatientViewset
BloodPressurePatientViewset
+ visibilityUpdated()
+ highlightedUpdated()
+ getNLatest()
PatientViewSet
+ patient_queryset
+ getPatient()
+ patientUpdated()
CholesterolPatientViewset
+ visibilityUpdated()
+ highlightedUpdated()
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
urls.py
rest_framework
serializers.py
urls.py routes API calls to/from the front-end to viewsets (the Controller in the Django back-end)
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 is a Django package used by ViewSet to serialize/deserialize object (Model) data between the back-end to front-end
models.py
models.py is a Django package that has functionality for interacting with SQLite3 database
Practitioner
+ identifier
- getFHIRPatients()
- findFHIRPatientInfo()
- createPatient()
Model
+ objects
+ save()
Model is the Django abstract class that handles querying objects/saving tables to the database
Patient
+ id
+ given_name
+ family_name
+ prefix_name
+ birth_date
+ gender
+ city
+ state
+ country
Each Model class represents a table in the database to be queried to/from
CholesterolPractitioner
+ chol_prac_id
+ average_cholesterol
+ filter_cholesterol
+ refresh_rate
+ above_average_count
+ getCholPatients()
+ getHighlighted()
+ calculateAverageCholesterol()
SmokerPractitioner
+ smoker_prac_id
+ non_smoker_count
+ getSmokerPatients()
+ getNonSmokerCount()
BloodPressurePractitioner
+ bp_prac_id
+ filter_bp_sys
+ filter_bp_dia
+ highlighted_sys_count
+ highlighted_dia_count
+ refresh_rate
+ getBPPatients()
+ getDiaHighlighted()
+ getSysHighlighted()
BloodPressurePatient
+ bp_patient_id
+ latest_sys_bp
+ latest_dia_bp
+ latest_time
+ visible
+ highlighted_sys
+ highlighted_dia
+ toggleVisibility()
+ toggleHighlighted()
+ getLatestBP()
+ getNLatestSys()
+ getLatestSys()
+ getLatestDia()
SmokerPatient
+ smoker_patient_id
+ smoker_status
+ getSmokerStatus()
CholesterolPatient
+ chol_patient_id
+ visible
+ highlighted
+ above_average_count
+ latest_chol
+ latest_time
+ toggleVisibility()
+ toggleHighlighted()
+ getLatestChol()

Design rationale

Implementing the new requirements to the existing codebase was a simple affair due to the integration of various design principles and patterns into the program's design at inception. No major changes or refactoring needed to be made to the overall architecture of the system. As such, the general *MVC* design pattern including a *Model* (represented by models.py), *View* (represented by the React package) and a *Controller* (represented by viewsets.py) was retained due to its effectiveness in allowing team members to work in parallel, minimising coupling between system components as well as separating system concerns (i.e. databases, GUI, and program logic).

Seeing total cholesterol observations of all monitored patients required a simple extension of front-end View classes and required no change to the existing back-end API which provided the same data. All that needed to be extended was the front-end at the PractitionerView and PatientView hinge points, conforming to the *Dependency Inversion Principle* and the *Open-Closed Principle*. In this way, the back-end still depended on abstract interfaces rather than concrete implementations.

Monitoring blood pressure (both systolic and diastolic) shared various similarities with their cholesterol counterparts. Hence, all that was required was an extension of the core Practitioner and Patient functionality into BloodPressurePractitioner and BloodPressurePatient classes (in viewsets.py and models.py) to capture specific blood-pressure related functionality, with all shared functionality being retrieved from the core, abstract classes in those packages themselves. Because BloodPressure and Cholesterol retrieve separate data and deal with it differently (with different filtering methods, and server calls), separate interfaces (classes) were created for both these to reduce coupling between them and the system, conforming to the *Interface Segregation Principle*.

This allowed highlighting functionality to remain individualised according to the different fields required (e.g. cholesterol or systolic blood pressure or diastolic blood pressure), instead of being a global variable for all Patients. Retrieving and graphing blood pressure values was once more encapsulated into BloodPressurePatient on the back-end and extended from abstract View classes on the front-end.

These design choices resulted in more stable packages, as all additional functionality was extended at previously provided hinge points and kept within the bounds of individual packages. As packages had abstract interfaces as their main interaction points with other packages, stable packages were essentially those that were abstract in nature, conforming to the *Stable Abstractions Principle* as well as the *Stable Dependencies Principle*.

However, the extension of the system did strengthen some existing flaws. Most notably, packages were increased in size rather than broken down into smaller sub-packages. While this promoted the *Release Reuse Equivalence* and *Common Closure* principles by making packages easier to maintain, it made package reuse a tougher affair as clients who need to reuse/upgrade packages in the future will have to download larger, more unnecessary code.