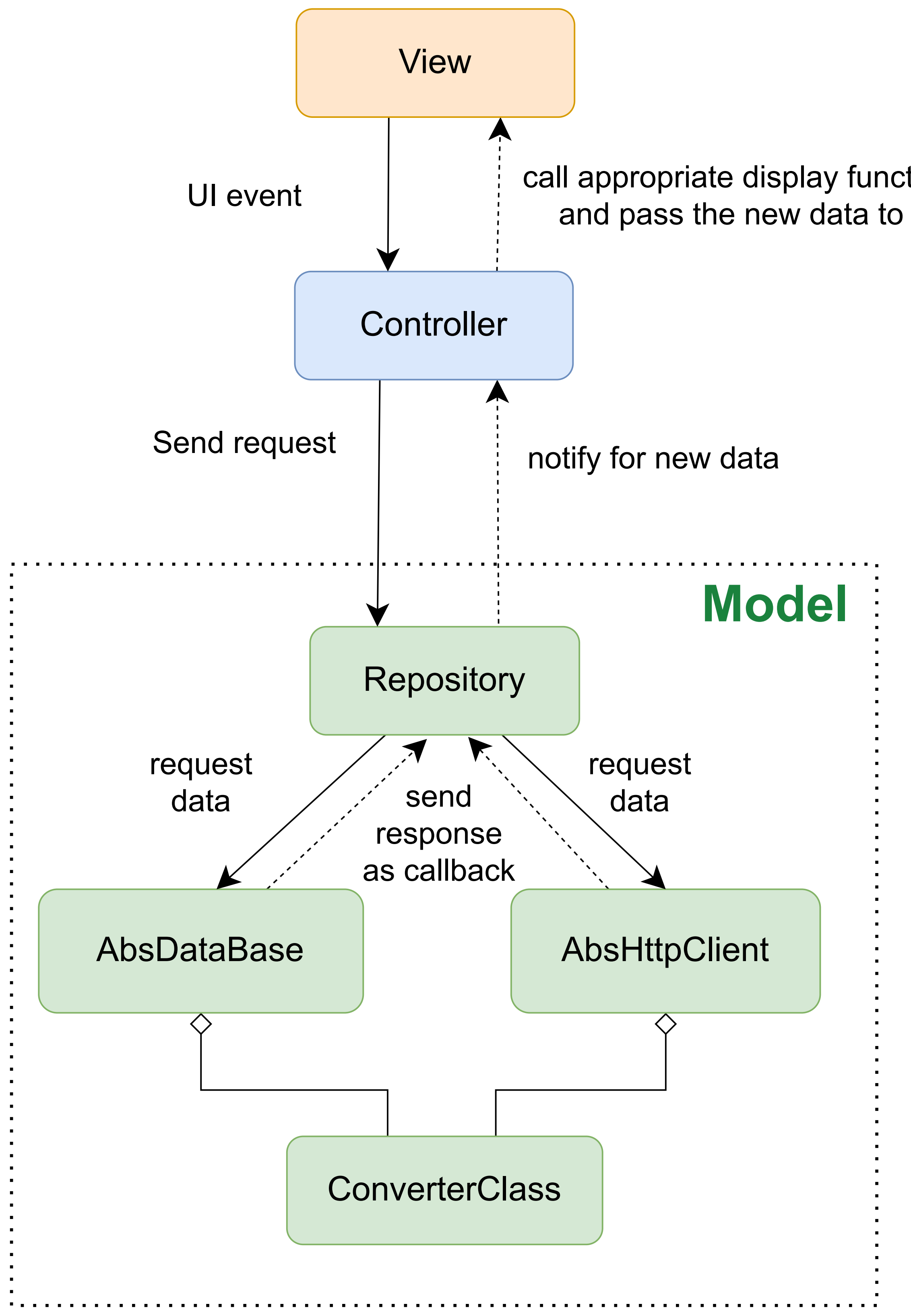
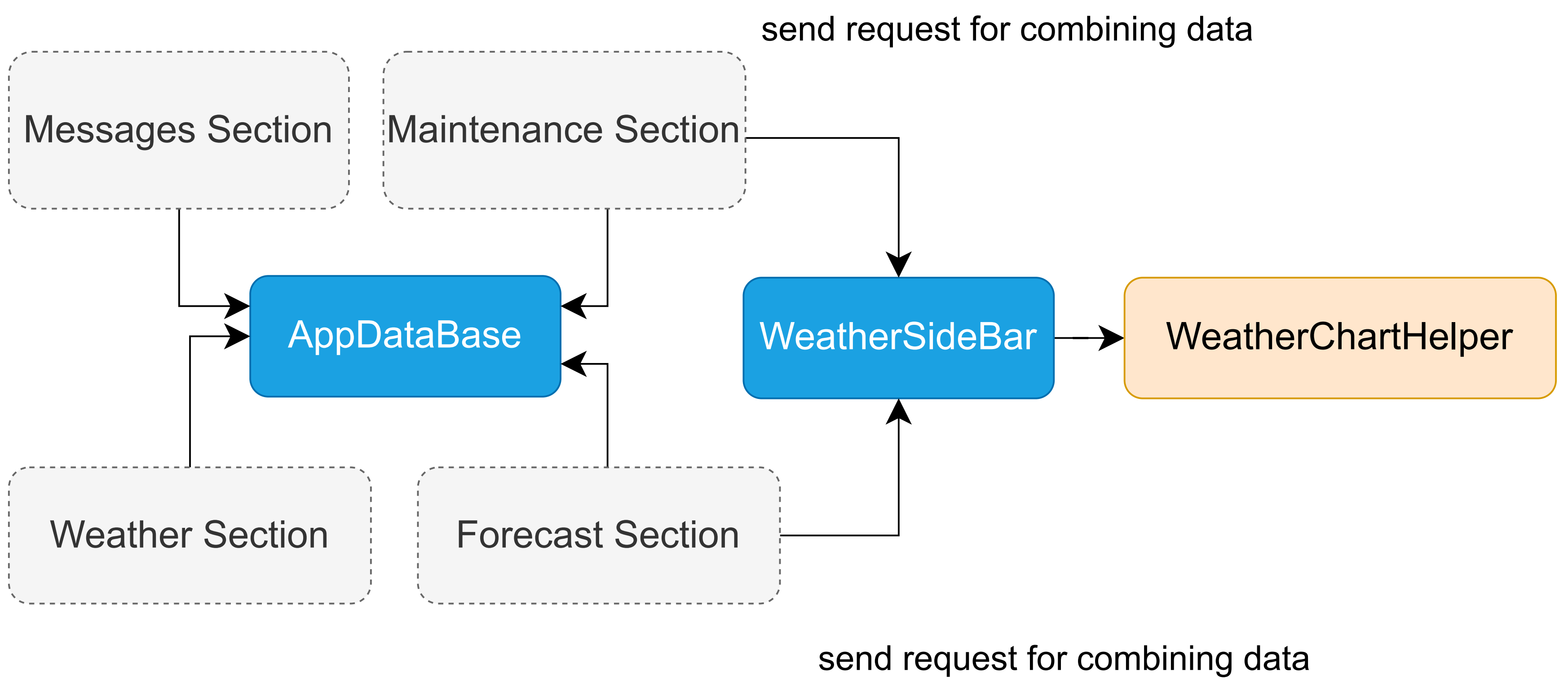
General structure of a "section" in the app



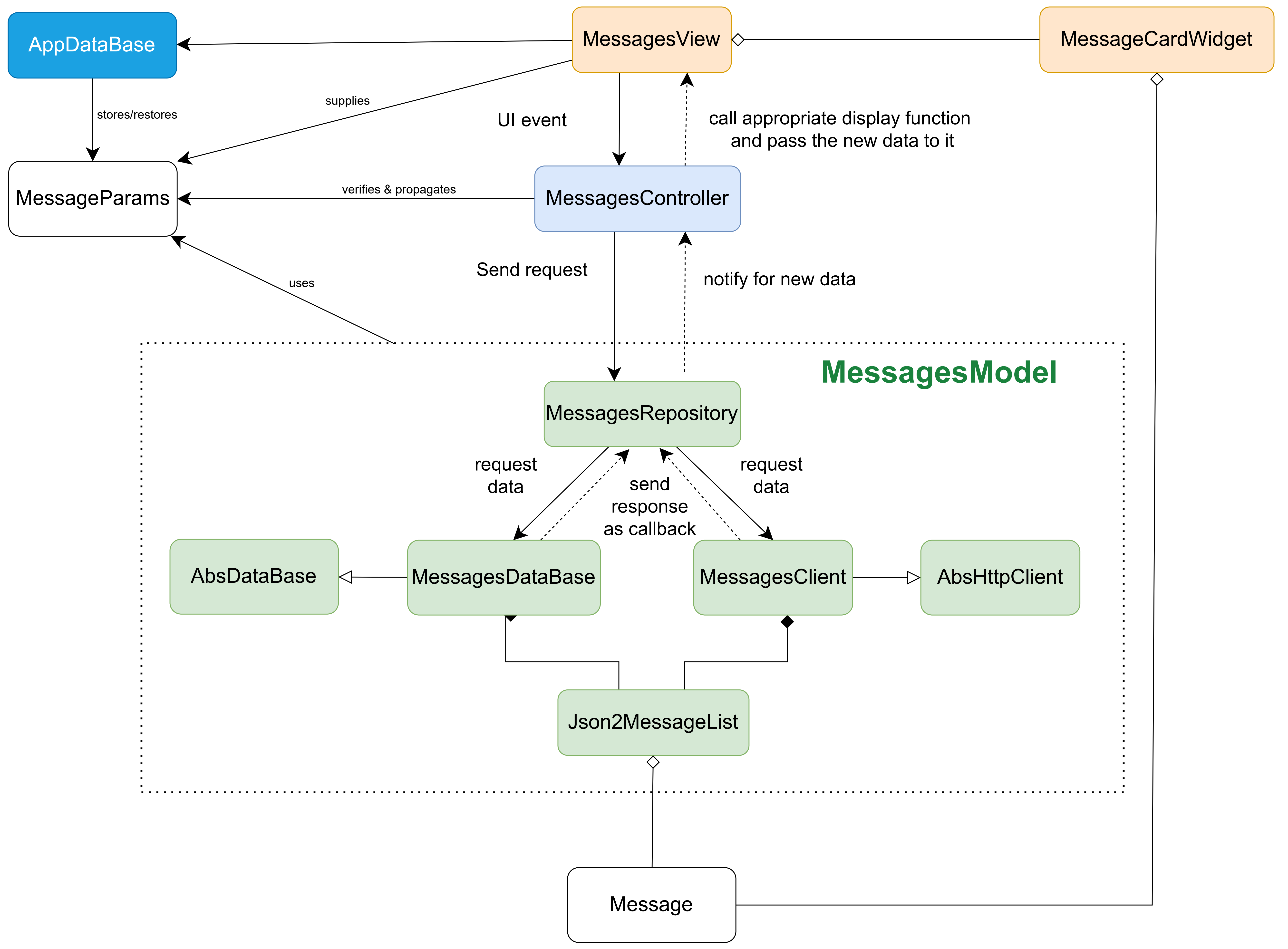
Note: details on this diagram are explained in the ***Design_document_design_decisions.pd***f document

Connection between different app sections with the weather side bar and app data base classes



Class	Description/Responsibility
AppDataBase	responsible for storing and restoring user inputs: saved locations and saved parameters
WeatherSideBar	responsible for desplaying forecast or weather observations on the sidebar, to allow combination with data traffic. Resquests to this class come from data traffic controller classes

Messages Section class diagram



MessageDataBase

- db: QFile

+ fetchMessages(filename: QString, OnMessagesReady): void
+ storeMessages(filename QString, onMessagesStored): void

MessageClient

- netManager: QNetworkAccessManager

+ fetchMessages(MessageParams, OnMessagesReady): void

MessagesRepository

- db: MessagesDataBase
- client: MessagesClient

+ fetchMessages(MessageParams, OnMessagesReady): void
+ fetchMessages(filename: QString, OnMessagesReady): void
+ fetchMessageTypes(): QList<QString>
+ storeMessages(filename QString, onMessagesStored): void

Json2MessageList

- rawData: QByteArray
- convertedData: QList<Message>

+ process(rawData: QByteArray): void
+ getMessagesList(): QList<Message>

MessageParams

- hoursInPast: int
- situationType :QString
- messagesBaseUrl : const QString

+ toHttpRequestUrl() : QString
+ fromJSON(QJsonObject): void
+ toJSON(): QJsonObject

Message

- type: QString
- title: QString
- location: QString
- features: List<QString>
- comment: QString
- startTime: QDateTime
- endTime: QDateTime

+ getType(): QString
+ getTitle(): QString
+ getLocation(): QString
+ getFeatures(): List<QString>
+ getComment(): QString
+ getStartTime(): QDateTime
+ getEndTime(): QDateTime

MessagesView

- msgController_: MessagesController *

+ setController(MessagesController *): void
+ displayCurrentMessageCards(QList<Message>): void
+ displayCurrentMessagesTitle(QString): void
+ displayComparingMessageCards(QList<Message>): void
+ displayComparingMessagesTitle(QString, QString): void
+ showSaveDataDialog(): void
+ showAlertDialog(QString): void
+ showWaitingDialog(): void
+ closeWaitingDialog(): void
+ enableSaveButton(): void
+ showDataChooserDialog(QStringList &): void
+ showSaveSuccessNoti(): void
+ saveUserInput(): void
+ restoreUserInput(): void

MessagesController

- msgRepo_: MessagesRepository &
- msgView_: MessagesView *

+ showButtonClicked(const int, const QString): void
+ saveButtonClicked(const QString): void
+ compareButtonClicked(): void
+ requestComparingData(const QString): void
+ checkUserInputs(const int, const QString): void

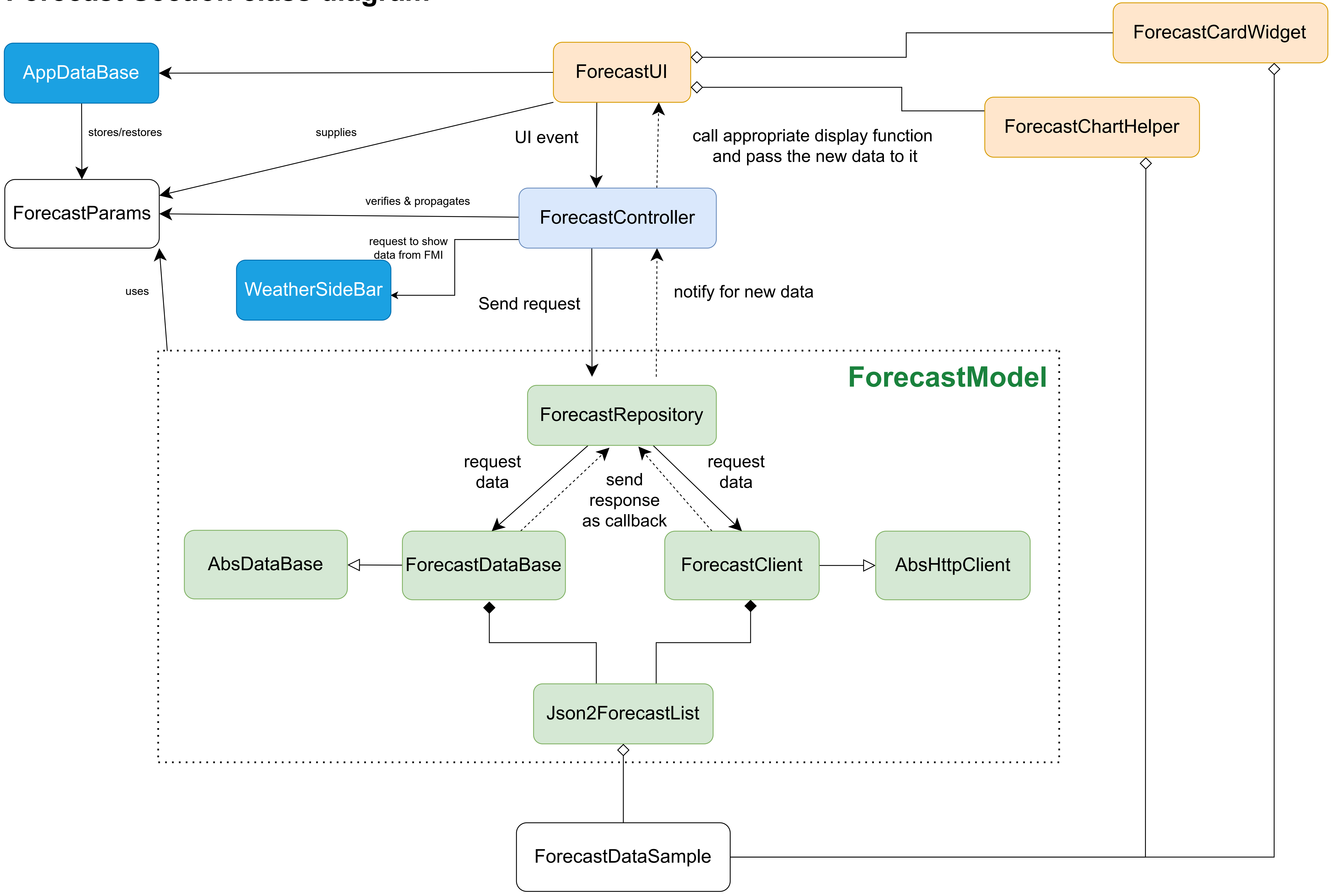
MessageCardWidget

- title_: QString
- comment_: QString
- location_: QString
- features_: QList<QString>
- startTime_: QDateTime
- endTime_: QDateTime
- email_: QString
- number_: QString

+ addCurrentDataToCard(): void
+ addComparingDataToCard(): void

Class	Description/Responsibility
MessagesView	responsible for forwading user input to the controller class (as parameters) and presenting the data from the Model class
MessagesController	responsible for checking the correctness of user inputs (parameters) coming from the view, and acts as a Mediator between the view and the model
MessagesRepository	acts as a facade for the internal classes of the model. It takes requests from the controller class and forwards that to the appropriate class
MessagesParams	a class that incapsulates the user inputs. Makes propagating those parameters easier and more maintainable between classes. Used also by database to save/restore user inputs
MessagesClient	fetches data from digitraffic and uses the right converter class to convert the data to usable c++ objects ready to be displayed.
MessagesDataBase	responsible for storing/restoring previously fetched data from digitraffic
Json2MessageList	responsible for converting raw data coming from the Client or DataBase classes to usable c++ objects.
Message	a class that holds a single message data.
MessageCardWidget	a custom widget class that has logic for displaying a message to the user. uses the previously mentioned Message class.

Forecast Section class diagram



ForecastDataBase

- db: QFile

+ fetchData(filename: QString, OnForecastDataReady): void
+ storeData(filename QString, onForecastStored): void

ForecastClient

- netManager: QNetworkAccessManager

+ fetchData(ForecastParams, OnForecastDataReady): void

MessagesRepository

- db: ForecastDataBase
- client: ForecastClient

+ fetchData(ForecastParams, OnForecastDataReady): void
+ fetchData(filename: QString, OnForecastDataReady): void
+ storeData(filename QString, onMessagesStored): void

ForecastCardWidget

- type_: QString
- condition_: QString
- forecastName_: QString
- reliability_: QString
- symbol_: QString
- airTemp_: double
- roadTemp_: double
- windSpeed_: double
- condition_: QString
- allReasons_: ForecastConditionReason

+ addDataToCard(): void
+ addComparingDataToCard(): void
+ addCurrentDataToCard(): void
+ getRoadTempCard(): QWidget *
+ getAirTempCard(): QWidget *
+ getDaylightCard(): QWidget *
+ getWindSpeedCard(): QWidget *
+ getWindDirCard(): QWidget *
+ getReliabilityCard(): QWidget *
+ getSymbolCard(): QWidget *
+ getReasonCard(): QWidget *

ForecastController

- forecastRepo_: ForecastRepository*
- forecastView_: ForecastView *

+ visualizeButtonClicked(const double, const double, const double, const double, const int): void
+ saveButtonClicked(QString): void
+ compareButtonClicked(): void
+ requestComparingData(QString): void
+ checkUserInputs(const double, const double, const double, const double): void

ForecastView

- forecastController_: ForecastController *

+ forecastController() const: ForecastController *
+ displayCurrentForecastCard(QList<ForecastDataSample>): void
+ displayComparingForecastCard(QList<ForecastDataSample>): void
+ displayCurrentForecastGraphs(QList<ForecastDataSample>): void
+ displayComparingForecastGraphs(QList<ForecastDataSample>): void
+ setController(ForecastController *): void
+ saveUserInput(): void
+ restoreUserInput(): void
+ showDataChooserDialog(QStringList &): void
+ showAlertDialog(QString): void
+ showWaitingDialog():
+ closeWaitingDialog(): void
+ showSaveDataDialog(): void
+ enableButtons(): void
+ enableSaveButton(): void
+ showTitle(): void
+ checkInputBeforeSave(ForecastParams): void
+ showSaveSuccessNoti(): void
+ showSaveButton(): void
+ location_added();: void

ForecastParams

- minLongitude: double
- minLatitude: double
- maxLongitude: double
- mqxLatitude: double
- messagesBaseUrl : const QString

+ fromJSON(QJsonObject): void
+ toJSON(): QJsonObject
+ toHttpRequestUrl() : QString

Class	Description/Responsibility
ForecastView	responsible for forwading user input to the controller class (as parameters) and presenting the data from the Model class
ForecastController	responsible for checking the correctness of user inputs (parameters) coming from the view, and acts as a Mediator between the view and the model
ForecastRepository	acts as a facade for the internal classes of the model. It takes requests from the controller class and forwards that to the appropriate class
ForecastParams	a class that incapsulates the user inputs. Makes propagating those parameters easier and more maintainable between classes. Used also by database to save/restore user inputs
ForecastClient	fetches data from digitraffic and uses the right converter class to convert the data to usable c++ objects ready to be displayed.
ForecastDataBase	responsible for storing/restoring previously fetched data from digitraffic
Json2ForecastList	responsible for converting raw data coming from the Client or DataBase classes to a list of ForecastDataSample objects.
ForecastDataSample	a class that holds the the data fetched for one forecast data sample (one response contains 6 data samples for each location)
ForecastCharthelper	helper class to draw forecast charts
ForecastCardWidget	a custom view used for presenting one forecast information from a data sample

ForecastDataSample

- id: int
- type: QString
- forecastName: QString
- daylight: bool
- roadTemperature: QString
- airTemperature: QString
- windSpeed: float
- windDirection: float
- overallRoadCondition: QString
- weatherSymbol: QString
- reliability: QString
- forecastReason: List<QString>

+ getType(): QString
+ getId(): int
+ getForecastName(): QString
+ getDaylight(): QBool
+ getRoadTemperature: QString
+ getAirTemperature: QString
+ getWindSpeed: QFloat16
+ getWindDirection: QFloat16
+ getOverallRoadCondition: QString
+ getWeatherSymbol: QString
+ getReliability: QString
+ getForecastReason: List<QString>

ForecastChartHelper

- lineSeries1: QLineSeries *
- lineSeries2: QLineSeries *
- chartView_: QChartView*
- chart_: QChart*
- x1Axis: QValueAxis*
- y1Axis: QValueAxis*
- y2Axis: QValueAxis*
- maxValue: double

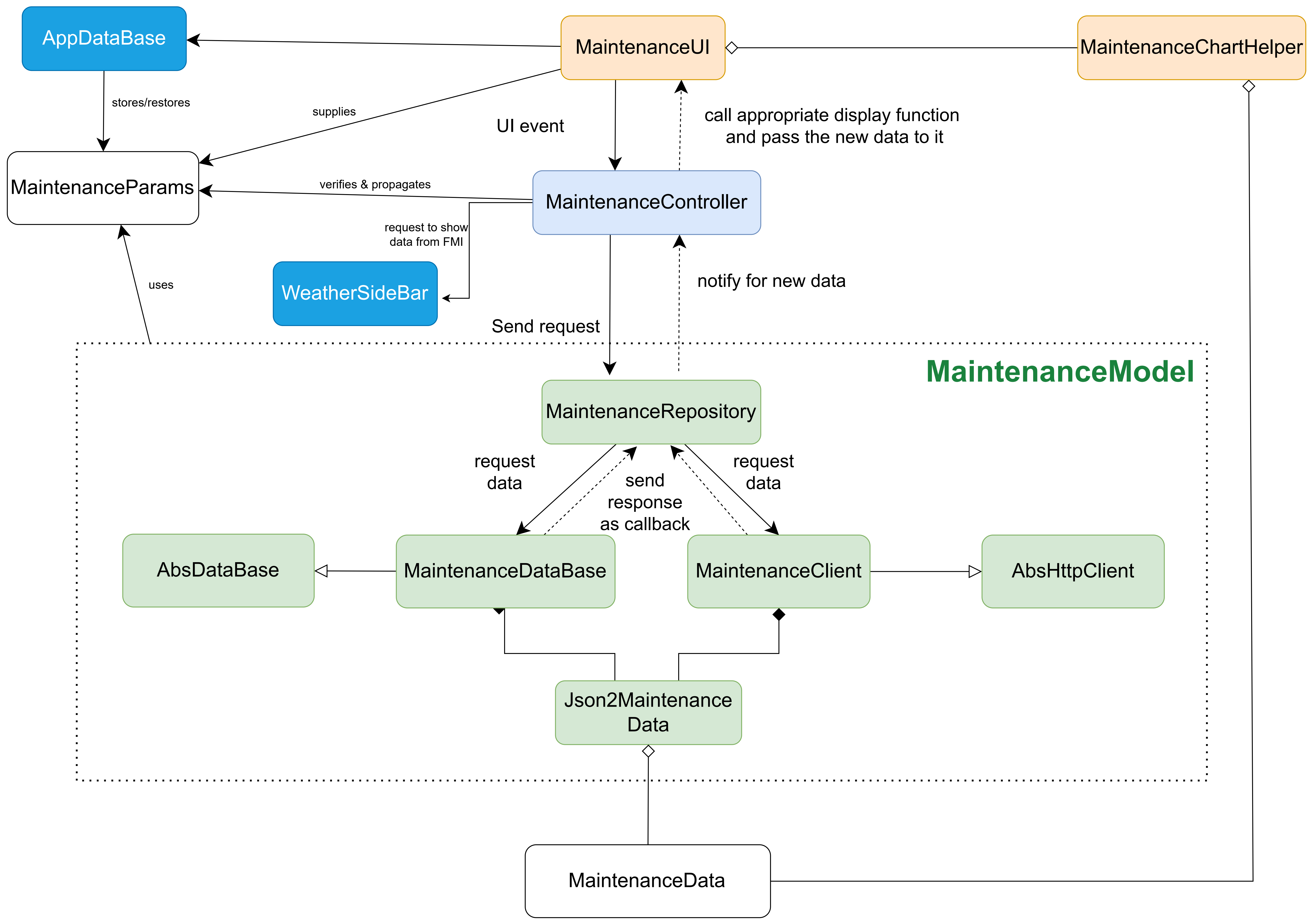
+ displayData(...): void
+ displayDataWithTwoYAxes(...): void
+ saveToPng(): void

Json2ForecastList

- rawData: QByteArray
- convertedData: QList<ForecastDataSample>

+ process(rawData: QByteArray): void
+ getForecastData(): QList<ForecastDataSample>

Maintenance Section class diagram



MaintenanceDataBase

- db: QFile

+ fetchMaintenanceData(filename: QString, OnMaintenanceDataReady): void
+ storeMaintenanceData(filename QString, OnMaintenanceDataStored): void

MaintenanceClient

- netManager: QNetworkAccessManager

+ fetchTasksCount(MaintenanceParams,OnMaintenanceDataReady): void

MaintenanceRepository

- db: MaintenanceDataBase
- client: MaintenanceClient

+ getSpecificTaskData(filename: QString, OnMaintenanceDataReady): void
+ getSpecificTaskData(MaintenanceParams, OnMaintenanceDataReady): voi
+ storeSpecificTaskData(filename: QString, OnMaintenanceDataReady): void
+ getMaintenanceTaskTypes(): QList<QString>
+ getStoredFilesList(): QStringList

MaintenanceParams

- fromTime: QDateTime
- toTime: QDateTime
- xMin: float
- yMin: float
- xMax: float
- yMax: float
- taskID: QString
- baseUrl: const QString

+ toHttpRequestUrl(): QString
+ fromJSON(QJsonObject): void
+ toJSON(): QJsonObject
+ getters and setters for all the fields above

MaintenanceData

- tasksCount: int
- tasksID: QString

+ getTasksCount() const: int
+ setTasksCount(int): void
+ getTasksID() const: const QString &
+ setTasksID(const QString &newTasksID): void

MaintenanceController

- repo: MaintenanceRepository*
- files: QStringList

+ requestMaintenanceData(MaintenanceParams): void
+ requestMaintenanceData(QString): void
+ requestStoreMaintenanceData(QString): void
+ requestSavedFilesList(): void
+ view_showAlertDialog(QString): void
+ view_showWaitingDialog(): void
+ view_showAlertDialog():void
+ view_closeWaitingDialog(): void
+ view_showSaveDataDialog(): void
+ view_showDataChooserDialog(QStringList &): void
+ view_displayMaintenanceData(MaintenanceData, MaintenanceDataSrc): void
+ view_enableSaveButton(): void
+ view_showSaveSuccessNoti(): void

MaintenanceView

- controller: MaintenanceController*

+ getController() const: MaintenanceController *
+ displayMaintenanceData(MaintenanceData, MaintenanceDataSrc): void
+ showAlertDialog(QString): void
+ showWaitingDialog(): void
+ closeWaitingDialog(): void
+ showSaveDataDialog(): void
+ showDataChooserDialog(QStringList): void
+ enableSaveButton(): void
+ showSaveSuccessNoti(): void
+ saveUserInput(): void
+ restoreUserInput(): void
+ location_added();: void

Json2MaintenanceData

- rawData: QByteArray
- convertedData: MaintenanceData

+ process(rawData: QByteArray): void
+ getMaintenanceData(): MaintenanceData

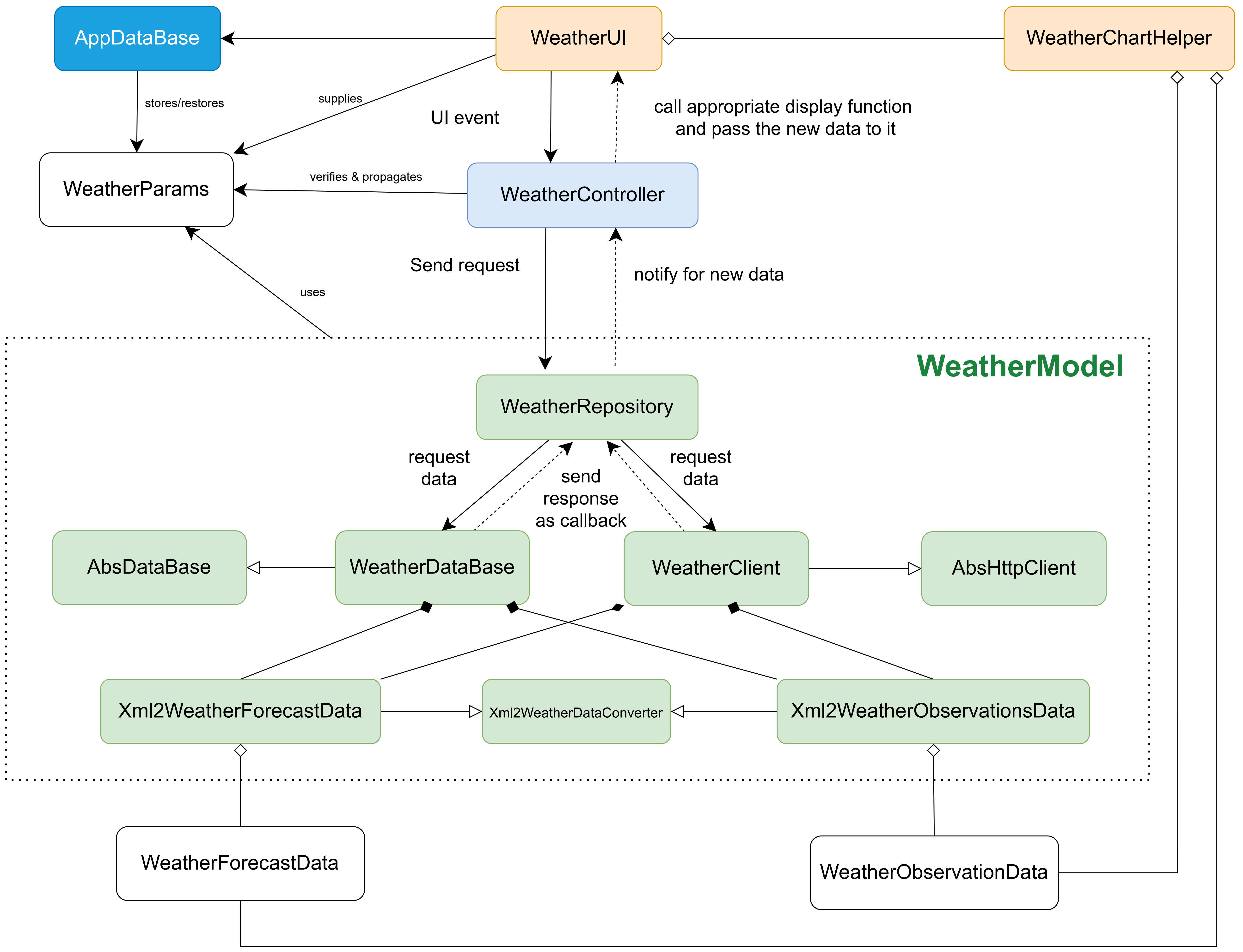
Class	Description/Responsibility
MaintenanceView	responsible for forwading user input to the controller class (as parameters) and presenting the data from the Model class
MaintenanceController	responsible for checking the correctness of user inputs (parameters) coming from the view, and acts as a Mediator between the view and the model
MaintenanceRepository	acts as a facade for the internal classes of the model. It takes requests from the controller class and forwards that to the appropriate class
MaintenanceParams	a class that incapsulates the user inputs. Makes propagating those parameters easier and more maintainable between classes. Used also by database to save/restore user inputs
MaintenanceClient	fetches data from digitraffic and uses the right converter class to convert the data to usable c++ objects ready to be displayed.
MaintenanceDataBase	responsible for storing/restoring previously fetched data from digitraffic
Json2MaintenanceList	responsible for converting raw data coming from the Client or DataBase classes to usable c++ objects.
MaintenanceData	a class that holds the count and type of a single maintenance task.
MaintenanceCharthelper	helper class to draw maintenance chart

MaintenanceChartHelper

- chartView: QChartView*
- chart: QChart*
- currentDataSet: QBarSet*
- savedDataSet: QBarSet*
- axisY: QValueAxis*
- categoriesAxis: QBarCategoryAxis*
- dataIndexMapper: QMap<QString, int>*
- barSeries: QBarSeries*

+ appendCurrentData(QString, int): void
+ appendSavedData(QString , int): void
+ clear(): void
+ getChartView(): QChartView*
+ saveToPng(): void

Weather Section overall class diagram



WeatherView
- controller: WeatherController*
+ displayMaxTemperature(QPair<QDateTime, double>): void
+ displayMinTemperature(QPair<QDateTime, double>): void
+ displayMaxWind(QPair<QDateTime, double>): void
+ displayMinWind(QPair<QDateTime, double>): void
+ displayMaxCloud(QPair<QDateTime, double>): void
+ displayMinCloud(QPair<QDateTime, double>): void
+ displayTempAverage(double): void
+ displayWindAverage(double): void
+ displayCloudAverage(double): void
+ plotFetchedObservationsData(WeatherObservationData): void
+ plotSavedObservationsData(WeatherObservationData): void
+ plotFetchedForecastData(WeatherForecastData): void
+plotSavedForecastData(WeatherForecastData): void
+ displayWaitingDialog(): void
+ closeWaitingDialog(): void
+ saveUserInput(): void
+ restoreUserInput(): void
+ showDataChooserDialog(QString): void
+ showAlertDialog(QString): void
+ showWeatherSavedNoti(): void
+ showSavePngButton(): void
+ displayObservedTempEmpty(): void
+ displayObservedWindEmpty(): void
+ displayObservedCloudEmpty(): void
+ displayPredicetedTempEmpty(): void
+ displayPredictedWindEmpty(): void
+ hideLabels(): void
+ showLabels(): void
+ location_added();: void

WeatherParams
- fromTime: QDateTime
- toTime: QDateTime
- xMin: int
- yMin: int
- xMax: int
- yMax: int
- timeStepInMinutes: int
- t2m: bool
- ws_10min: bool
- n_man: bool
- Temperature: bool
- WindSpeedMS: bool
- baseUrl: const QString
+ toObservationsHttpRequestUrl(): QString
+ toForecastHttpRequestUrl(): QString
+ toJSON(): QJsonObject
+ fromJSON(QJsonObject): void

Xml2WeatherDataConverter
- rawData: QByteArray
- paramListMap: QMap<QString, QList>
+ process(rawData: QByteArray): void
+ virtual populateParamListMap()

Xml2WeatherObservationsData
- rawData: QByteArray
- convertedData: WeatherObservationsData
+ populateParamListMap()
+ getObservationsData(): WeatherObservationsData

Class	Description/Responsibility
WeatherView	responsible for forwading user input to the controller class (as parameters) and presenting the data from the Model class (Weather observations, forecast and min/max/avg values)
WeatherController	responsible for checking the correctness of user inputs (parameters) coming from the view, and acts as a Mediator between the view and the model
WeatherRepository	acts as a facade for the internal classes of the model. It takes requests from the controller class and forwards that to the appropriate class
WeatherParams	a class that incapsulates the user inputs. Makes propagating those parameters easier and more maintainable between classes. Used also by database to save/restore user inputs
WeatherClient	fetches data from FMI and uses the right converter classes (for forecast or observations) to convert the data to usable c++ objects ready to be displayed.
WeatherDataBase	responsible for storing/restoring previously fetched data from FMI
Xml2WeatherDataConverter	This is and abstract class responsible for implementing all the logic for parsing xml data from FMI and populating lists described by child classes with the parsed values
Xml2WeatherObservationData	extends the Xml2WeatherDataConverter and provides it with tartget lists to populate and with information about which data to look for during parsing (weather observation data in this case)
Xml2WeatherForecastData	extends the Xml2WeatherDataConverter and provides it with tartget lists to populate and with information about which data to look for during parsing (weather forecast data in this case)
WeatherObservationData	a class that holds lists of weather observations data fetched from FMI
WeatherForecastData	a class that holds lists of weather forecast data fetched from FMI
WeatherCharthelper	helper class to draw weather forecast and observations chart charts

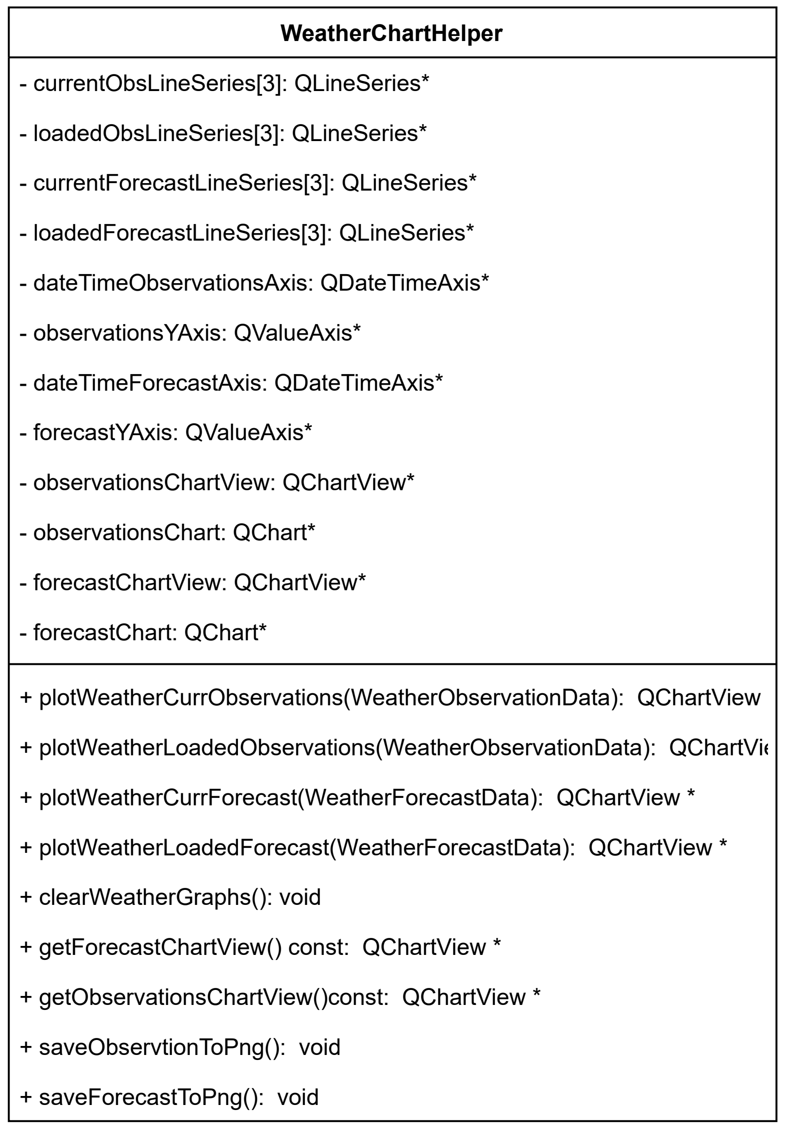
WeatherController
- weatherView; WeatherView*
- weatherRepo_: WeatherRepository
+ visualizeButtonClicked(QString, WeatherParams): void
+ saveButtonClicked(QString WeatherViewState): void
+ requestSavedFilesList(WeatherViewState): void
+ getSavedObservationData(QString): void
+ getSavedForcastData(QString): void
+ compareToSavedDataClicked(QString, WeatherViewState): void
+ averageButtonClicked(WeatherViewState) void

WeatherRepository
- db: WeatherDataBase
- client: WeatherClient
+ fetchWeatherObservations(WeatherParams, OnWeatherObservationsDataReady): void
+ fetchWeatherObservations(QString, OnWeatherObservationsDataReady): void
+ fetchWeatherForecast(WeatherParams, OnWeatherForecastDataReady): void
+ fetchWeatherForecast(QString, OnWeatherForecastDataReady): void
+ storeObservationsWeatherData(QStriing, OnWeatherDataStored): void
+ storeForecastWeatherData(QStriing, OnWeatherDataStored): void
+ getStoredObservationsFilesList(): QStringList
+ getStoredForecastFilesList(): QStringList

WeatherClient
- netManager: QNetworkAccessManager
+ fetchWeatherObservations(WeatherParams, OnWeatherObservationsDataReady):
+ fetchWeatherForecast(WeatherParams, OnWeatherForecastDataReady): void

WeatherDataBase
- db: QFile
+ fetchWeatherObservations(QString, OnWeatherObservationsDataReady): void
+ fetchWeatherForecast(QString, OnWeatherForecastDataReady): void
+ storeObservationWeatherData(QString, OnWeatherObservationsDataReady): void
+ storeForecastWeatherData(QString, OnWeatherForecastDataReady): void
+ getStoredObservationsFilesList(): QStringList
+ getStoredForecastFilesList(): QStringList

Xml2WeatherForecastData
- rawData: QByteArray
- convertedData: WeatherForecastData
+ populateParamListMap()
+ getForecastData(): WeatherForecastData



WeatherChartHelper

- currentObsLineSeries[3]: QLineSeries*
- loadedObsLineSeries[3]: QLineSeries*
- currentForecastLineSeries[3]: QLineSeries*
- loadedForecastLineSeries[3]: QLineSeries*
- dateTimeObservationsAxis: QDateTimeAxis*
- observationsYAxis: QValueAxis*
- dateTimeForecastAxis: QDateTimeAxis*
- forecastYAxis: QValueAxis*
- observationsChartView: QChartView*
- observationsChart: QChart*
- forecastChartView: QChartView*
- forecastChart: QChart*

+ plotWeatherCurrObservations(WeatherObservationData): QChartView
+ plotWeatherLoadedObservations(WeatherObservationData): QChartVie
+ plotWeatherCurrForecast(WeatherForecastData): QChartView *
+ plotWeatherLoadedForecast(WeatherForecastData): QChartView *
+ clearWeatherGraphs(): void
+ getForecastChartView() const: QChartView *
+ getObservationsChartView()const: QChartView *
+ saveObservtionToPng(): void
+ saveForecastToPng(): void