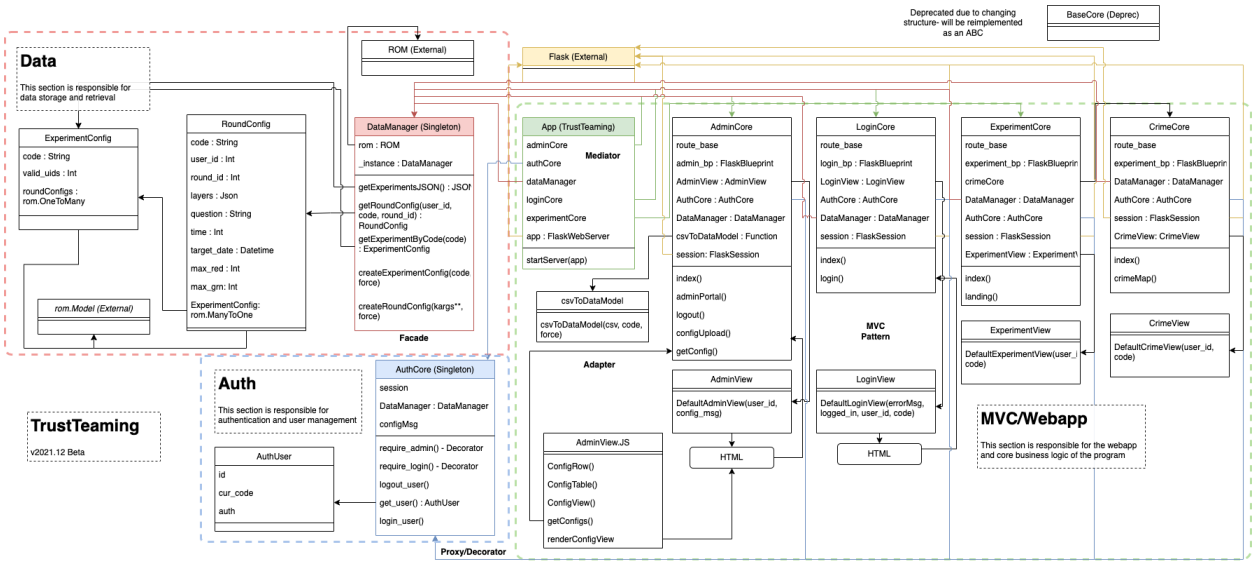
Trust Teaming

Michał Bodzianowski

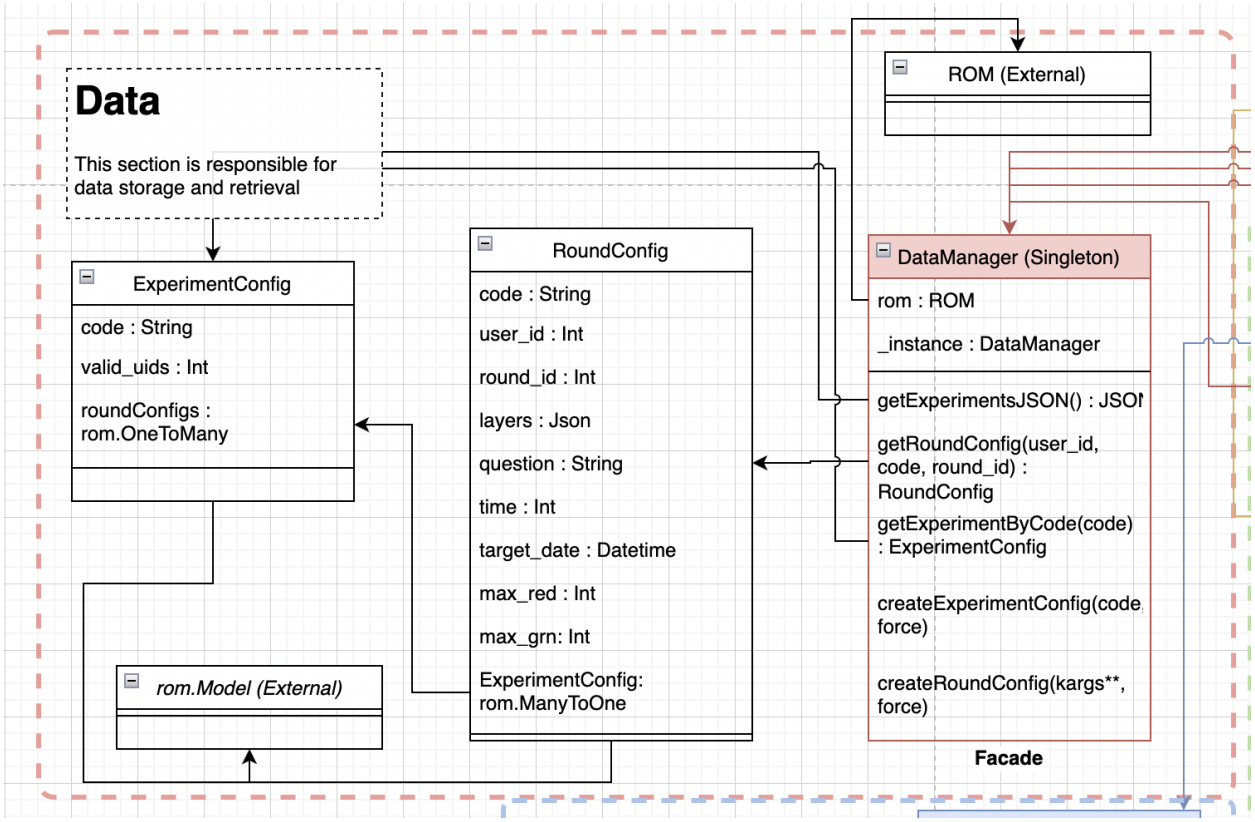
Final State of System

Of the major requirements, the framework was mostly satisfied. The core business logic remains unimplemented, but it has the support to do so. There is a robust authentication and login system. There is an admin management system, complete with file uploads and logic for that. And a framework to easily add and implement the business logic. The codebase is extremely well organized, and OOAD was performed throughout the development, which meant that while raw development speed was slowed down, the codebase is incredibly untangled and manageable.

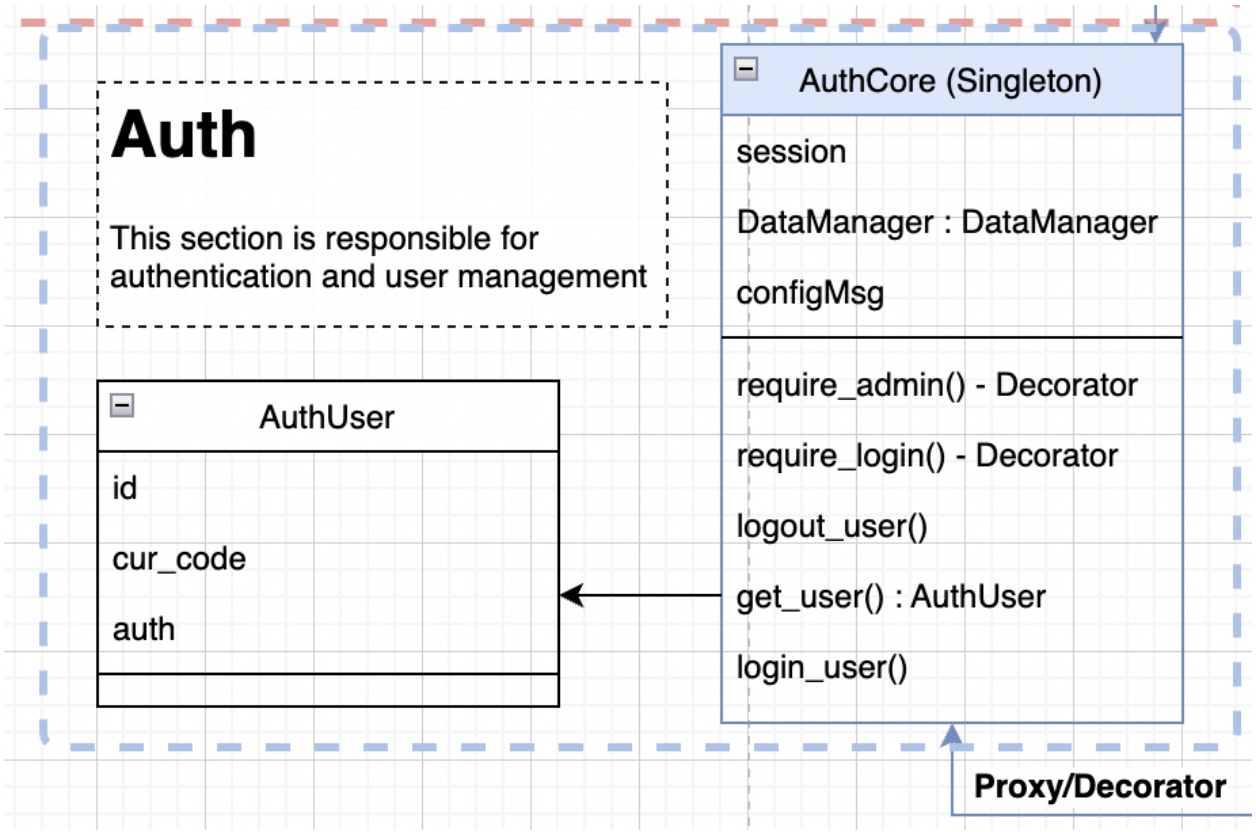
Final Diagram



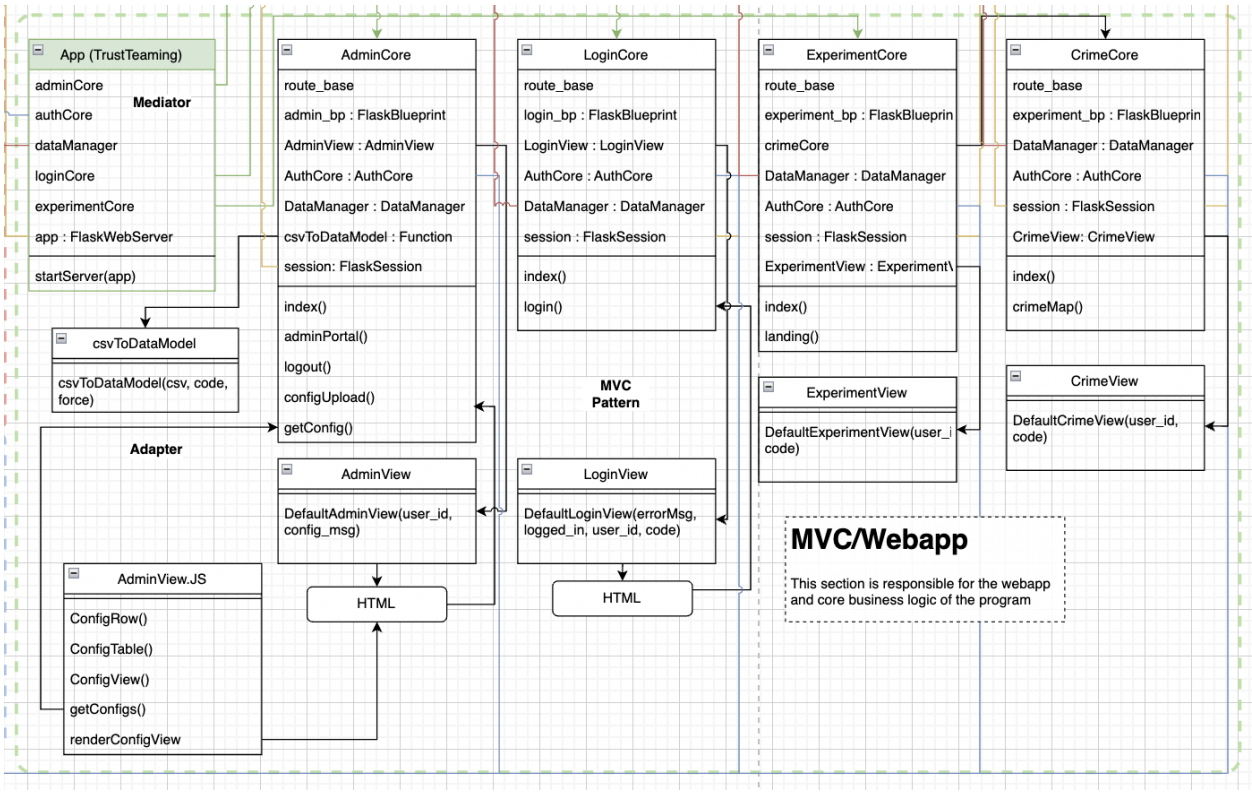
Data


Data
This section is responsible for data storage and retrieval
ROM (External)
ExperimentConfig
code : String
valid_uids : Int
roundConfigs : rom.OneToMany
RoundConfig
code : String
user_id : Int
round_id : Int
layers : Json
question : String
time : Int
target_date : Datetime
max_red : Int
max_grn: Int
ExperimentConfig: rom.ManyToOne
DataManager (Singleton)
rom : ROM
_instance : DataManager
getExperimentsJSON() : JSON
getRoundConfig(user_id, code, round_id) : RoundConfig
getExperimentByCode(code) : ExperimentConfig
createExperimentConfig(code force)
createRoundConfig(kargs**, force)
Facade
rom.Model (External)

Auth

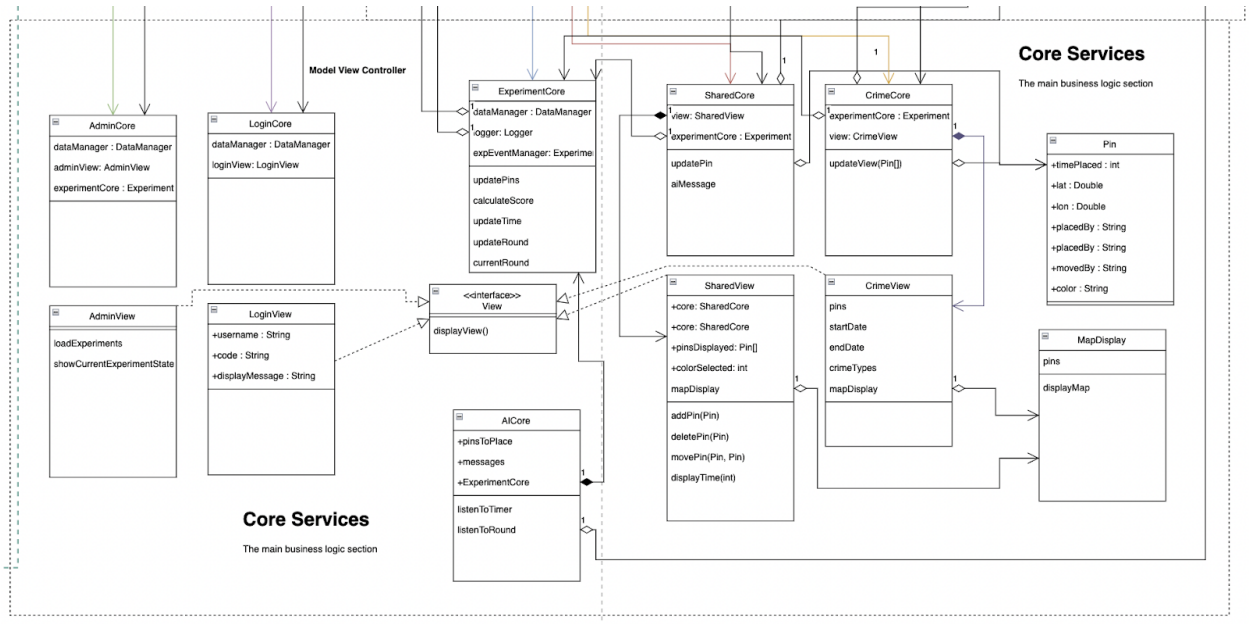

Auth
This section is responsible for authentication and user management
AuthCore (Singleton)
session
DataManager : DataManager
configMsg
require_admin() - Decorator
require_login() - Decorator
logout_user()
get_user() : AuthUser
login_user()
AuthUser
id
cur_code
auth
Proxy/Decorator

MVC/Webapp

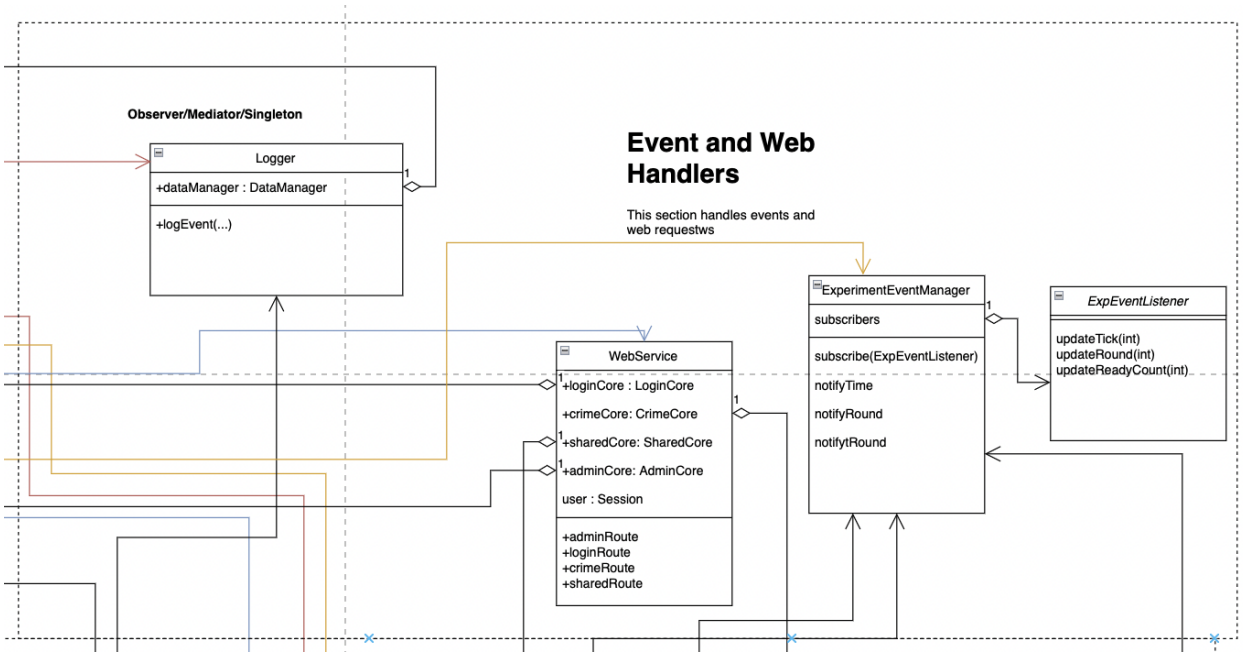

App (TrustTeaming)
adminCore
authCore
dataManager
loginCore
experimentCore
app : FlaskWebServer
startServer(app)
Mediator
AdminCore
route_base
admin_bp : FlaskBlueprint
AdminView : AdminView
AuthCore : AuthCore
DataManager : DataManager
csvToDataModel : Function
session: FlaskSession
index()
adminPortal()
logout()
configUpload()
getConfig()
LoginCore
route_base
login_bp : FlaskBlueprint
LoginView : LoginView
AuthCore : AuthCore
DataManager : DataManager
session : FlaskSession
index()
login()
ExperimentCore
route_base
experiment_bp : FlaskBlueprin
crimeCore
DataManager : DataManager
AuthCore : AuthCore
session : FlaskSession
ExperimentView : Experiment\
index()
landing()
CrimeCore
route_base
experiment_bp : FlaskBlueprin
DataManager : DataManager
AuthCore : AuthCore
session : FlaskSession
CrimeView: CrimeView
index()
crimeMap()
csvToDataModel
csvToDataModel(csv, code, force)
Adapter
MVC Pattern
ExperimentView
DefaultExperimentView(user_i code)
CrimeView
DefaultCrimeView(user_id, code)
AdminView
DefaultAdminView(user_id, config_msg)
LoginView
DefaultLoginView(errorMsg, logged_in, user_id, code)
AdminView.JS
ConfigRow()
ConfigTable()
ConfigView()
getConfigs()
renderConfigView
HTML
HTML
MVC/Webapp
This section is responsible for the webapp and core business logic of the program

Previous Class Diagram

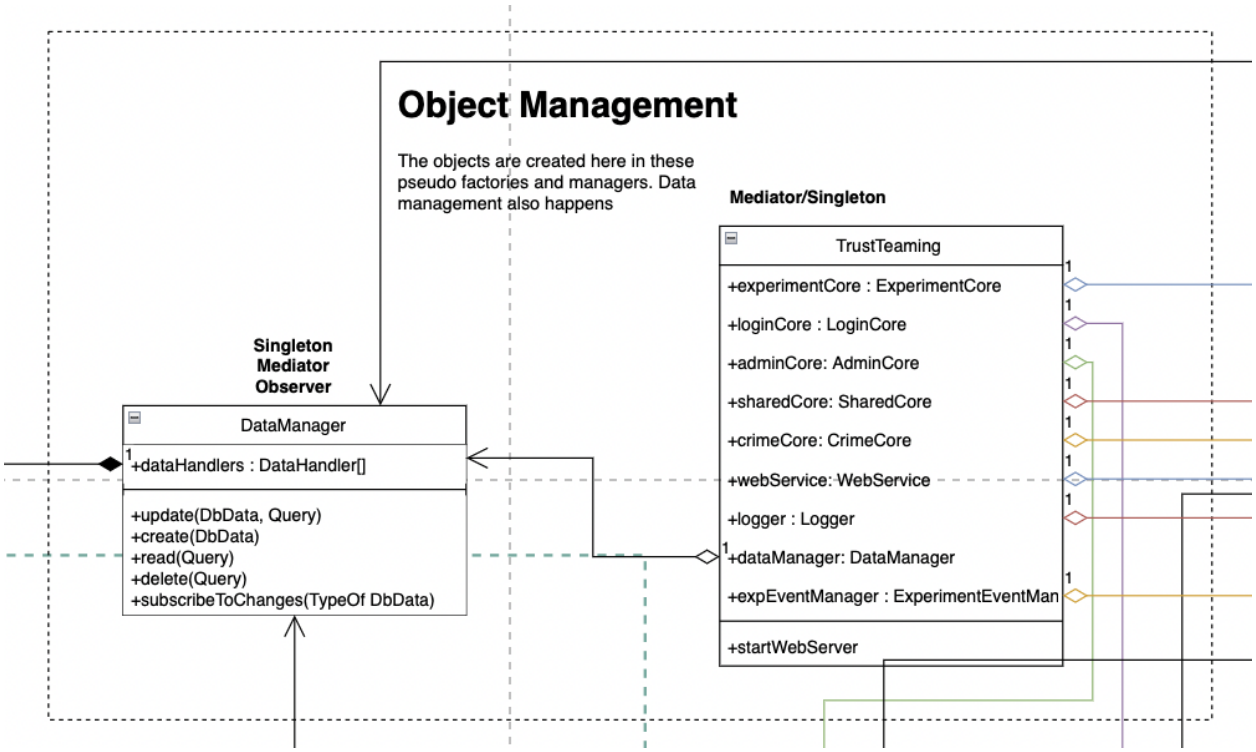
Core Services



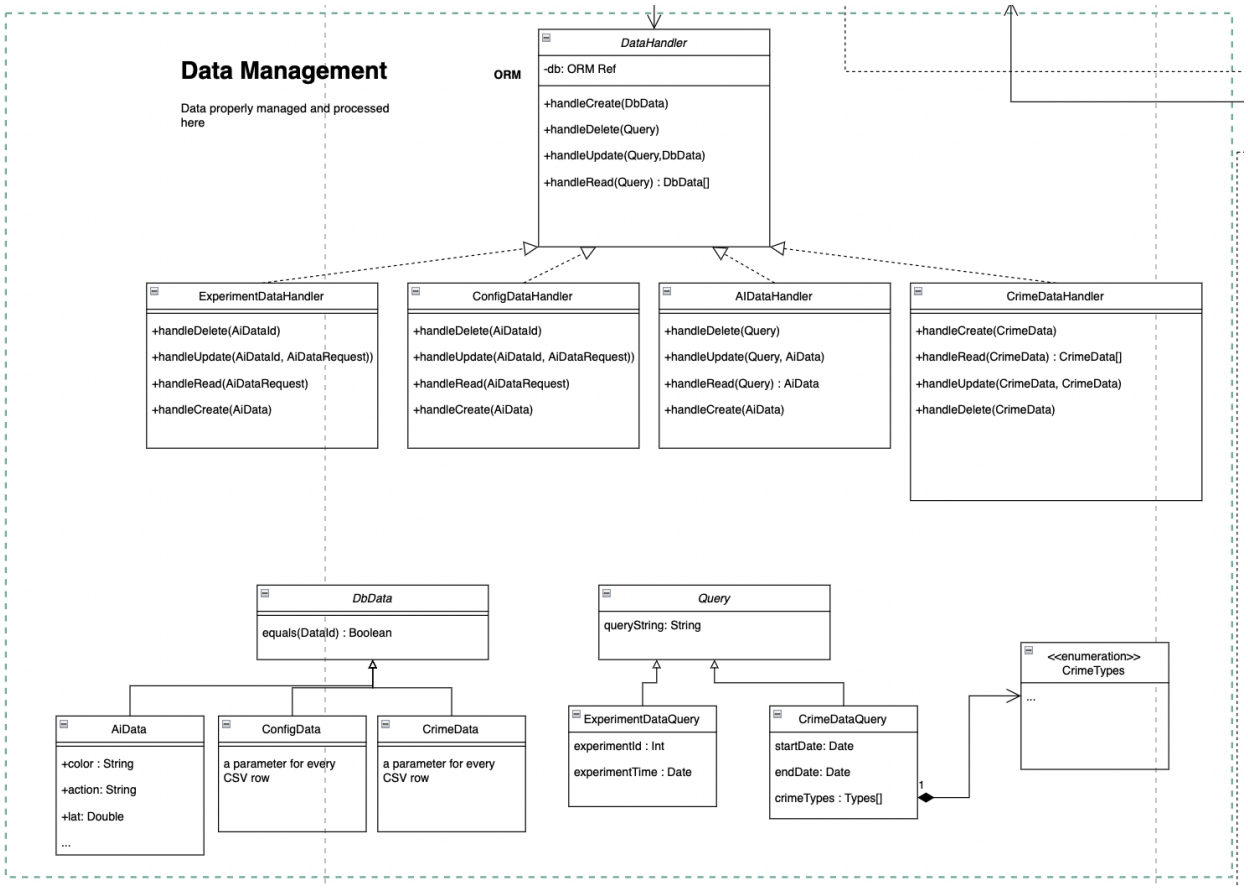
Event and Web Handlers



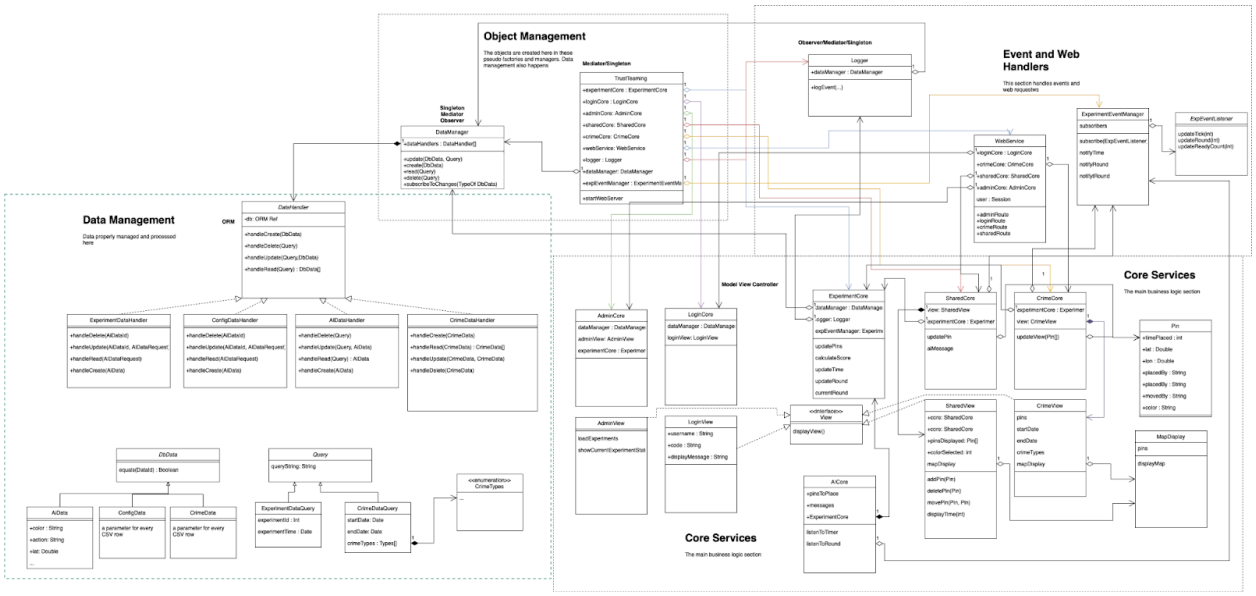
Object Management



Data Management



Birds-Eye View



Discussion on Differences

Overall, I followed most of what I could from the initial diagram. The largest difference is the exclusion of the logging/event handling features as I was unable to get to that point of the project. Across the board, there is simplification of the design thanks to the robustness of the ROM (Python-Redis ORM library). I used it in place of the Query/DbData classes I was designing. The Flask library had some help in implementing the Web Services section, so some of the parameters and functions changed there as well. Overall, however, the designs are not too dissimilar despite the tweaks in flows that occurred in implementation.

Third Party Code Statement

All third party code used is included as a comment above that section. Only very specific solutions to problems or helpful algorithms were copied- all other code is original work. In addition, I've specified the libraries used in `requirements.txt` and `package.json`.

The largest libraries used are ROM https://pypi.org/project/rom/ and Flask https://flask.palletsprojects.com/en/2.0.x/

OOAD Design Process

1. Designing and Architecting is hard! Especially for large scale programs, it's easy to fall into the temptation to include tight coupling and "just-get-it-done". Especially as I was scrambling to reach a time limit, I found myself going against my design and forgoing abstractions I "should" have been doing just to get something working. Perhaps some more time would alleviate this urge to skip the boilerplate code and go straight to functionality- even if it's not always the most optimal?
2. 3rd party libraries are incredibly hard to architect for, and then when it comes to implementations, so many things can go wrong. Many libraries are not as robust as they seem, and can cause all sorts of dependency hell. In my case, I had initially use two Flask add-on libraries, `flask-login` and `flask-classful`. Initially they both worked great, but when I tried to integrate them they were just not compatible at all. I ended up writing my own Auth solution, and replaced `flask-classful` with the `Blueprints` feature from the main `Flask` library- and this worked wonderfully.
3. Having things well-organized in file structure is a blessing! The previous implementation of this testbed that my project is replacing had all the code in just two python files. It made debugging absolute hell, and there's a reason we're replacing it- it simply refuses to work and there's no good way to figure out why without unravelling a 2000 line file. With every file in my implementation nearing only about 100 lines, its so much easier to

find buggy logic and fix it. I can also just follow my own patterns and easily create more cores, sites, and functionality on a whim. OOAD is great!