**Part 2 - Technical Design Exercise**

**Introduction to Design Approach**

My approach for the task would be an API based Microservice. Here we abstract the functionality to its own service to ensure that no full release or configuration file(s) change will be required on any of the web sites or servers; it is de-coupled. Feature Flags could be setup for each of the web sites as required within the proposed Service, each site could have 0..\* FeatureFlags configured. Additionally, the fact a Feature Flagging Dashboard will be built in the future steered me towards a Microservice. When the Dashboard is constructed, it can be done using the Endpoints outlined here.

**End Points**

* GET Status(string website Name, string FeatureName): Task<ActionResult<bool>>
  + Returns 200 if successful (with either true or false), 403 not found
* PUT Toggle ([From Body] webSiteFeatureDto): Task<HttpResponseMessage>
  + Returns 204 if successful, 403 not found, 401 not authorised
* POST Create ([FromBody] webSiteFeatureDto): Task<HttpResponseMessage>
  + Returns 204 if successful, 400 bad request, 401, not authorised
* DELETE Delete([FromBody] webSiteFeatureDto): Task<HttpResponseMessage>
  + Returns 204 if successful, 403 not found, 401 not authorised

Consumers of the API should be defensive. If no response / 408 Timeout, assume feature is disabled. The endpoints will bind to a Model View which is a set of Models representing the Feature Flag.

**Query Handler / Command Handler**

I would separate out reading and writing to a ‘QueryHandler’ and ‘CommandHandler’ respectively (Single Responsibility Principle). The incoming requests would be routed here from the Controller via the specific Endpoints. I would stop short of implementing the CQRS (Command Query Responsibility Segregation) pattern because of the scope of the problem. If it became more complex in the future, this is something that could be considered.

**Database**

A database would be used for persistence (refer to Entity Relationship Diagram). Each Handler has DB Context injected into it via Dependency Injection. I have not indicated a specific database but would favour a Relational one due to the level of querying that would be involved for each rule.

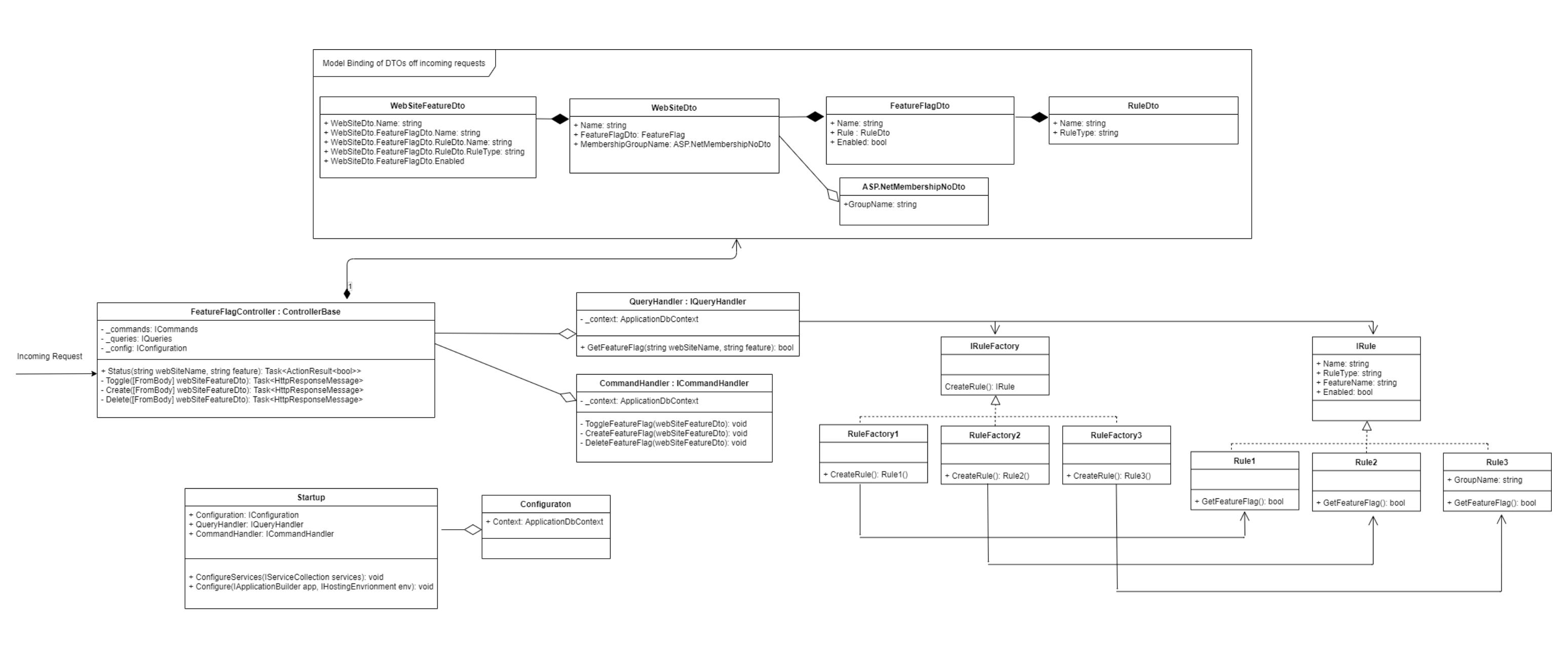
**Rules**

The different rules can be implemented using the Abstract Factory Method pattern. This will ensure that the code is open for extension but closed for modification (Open Close Principle), which is important because the design requirements state further rules will be needed in the future.

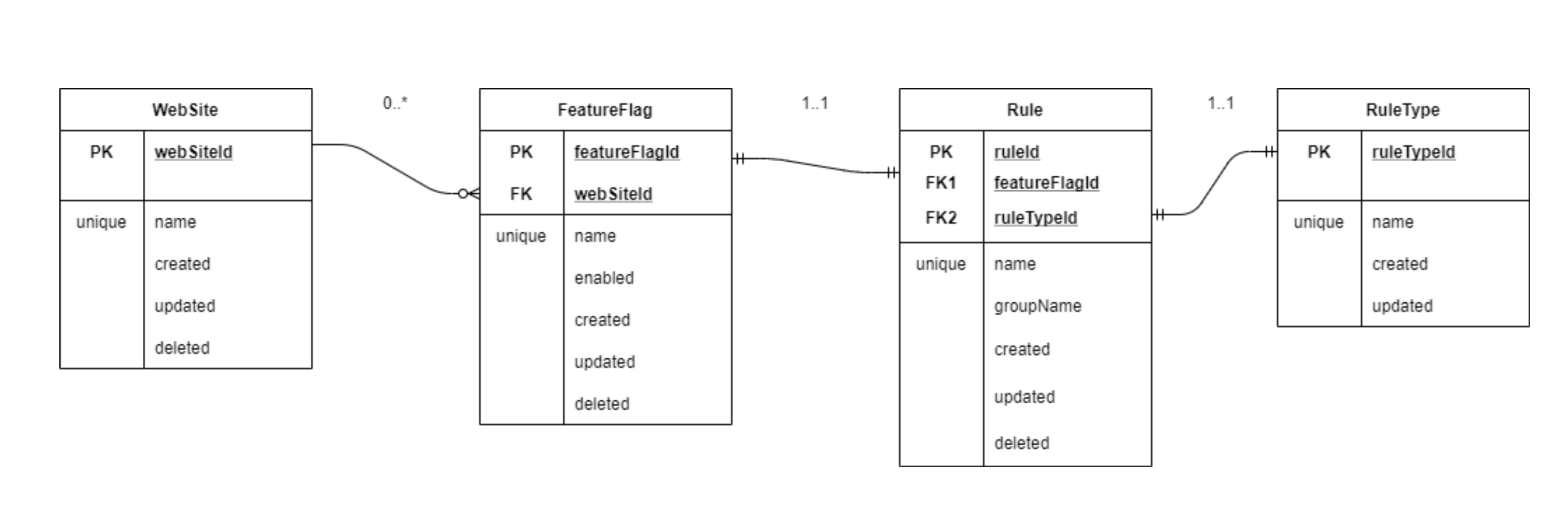
**Note:** Dependency Injection \ .Net Core FeatureManagement

All of the rules outlined in the brief can be achieved through Dependency Injection and the .Net Core Library FeatureManagement, however this is predominantly used with data configured in the ‘appsettings.json’ files. The requirement here was to update the configured rules (e.g. toggle on and off) so I have opted to use a separate Database for persistence.

**Class Diagram**



**Entity Relationship Diagram**



**How to Use the new Framework**

using (var client = new HttpClient())

{

var uri = "http://featuretoggle/v1/featureflag/status?webSiteName=<value>&featureName=<value>";

var response = client.GetAsync(uri);

}