**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. This will ensure that no full release or configuration file(s) change will be required on any of the consumer sites, 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.

**End Points**

* GET Status(string webSiteName, string FeatureName): Task<ActionResult<bool>>
  + Returns 200 if successful (with either true or false), 403 not found
* GET Status(string webSiteName, string FeatureName, string groupName): Task<ActionResult<bool>>
  + Returns 200 if successful (with either true or false), 403 not found
* PUT Toggle ([FromBody] 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.

**CQRS**

I would use the CQRS (Command Query Responsibility Segregation) pattern in the Microservice (Single Responsibility Principle). Here the incoming requests are directed to either a ‘QueryHandler’ or ‘CommandHandler’ depending upon the query type that will be applied to the DB.

**Database**

A database would be used for persistence (Please see Entity Relationship Diagram). Each Handler has DB context injected into it via Dependency Injection.

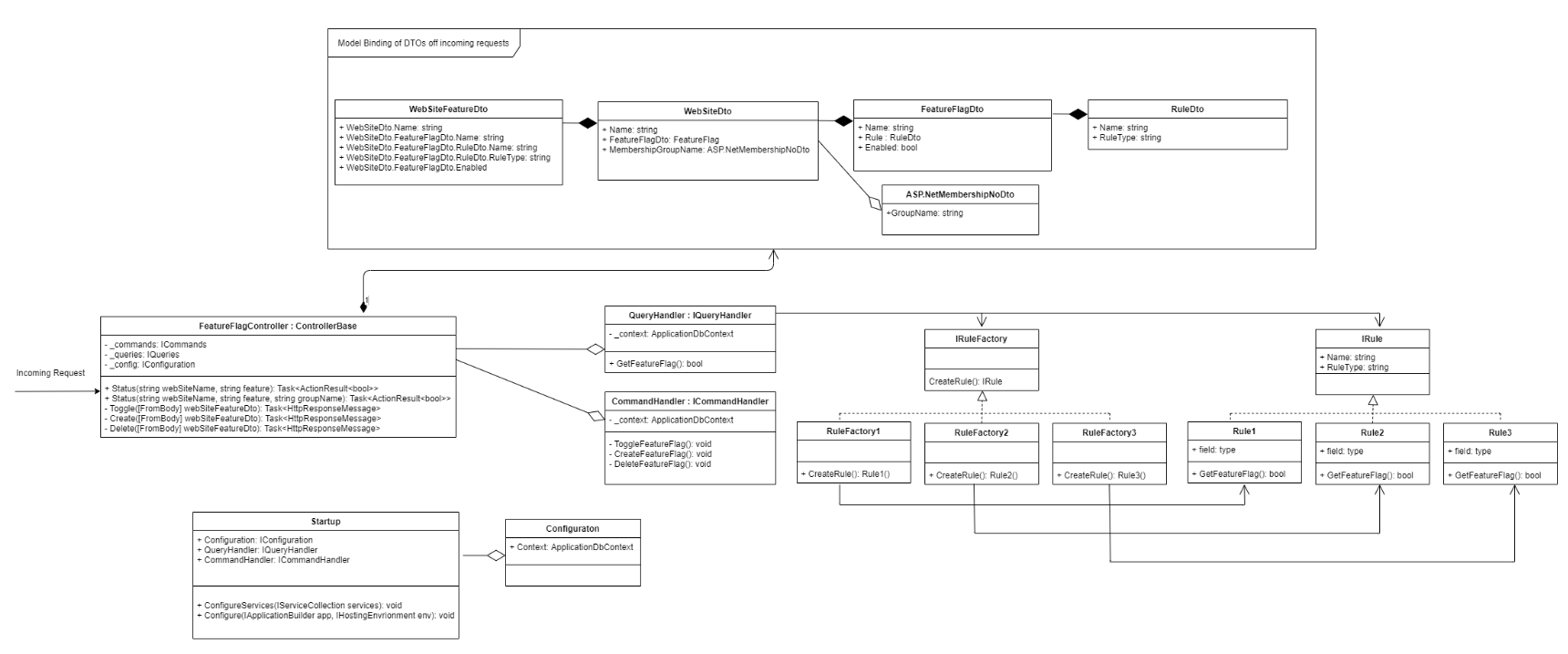
**Rules**

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

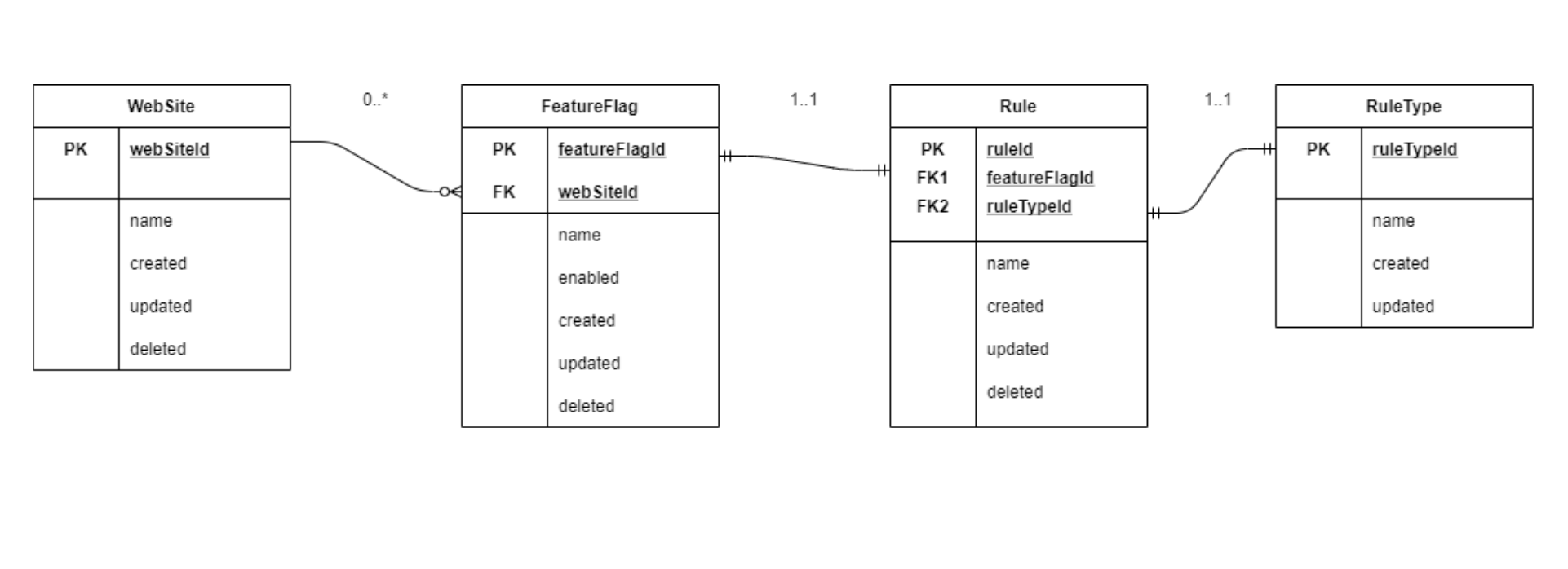
**Note:** Dependency Injection \ .Net Core FeatureManager

All of the rules outlined in the brief can be achieved through Dependency Injection and the .Net Core Library Feature Manager. 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);

}