cafe.connor.fun & Lighter Web Framework
Project Part 6: Final Report

1. Name

Ethan Hunter

2. Description

cafe.connor.fun is a webapp for finding cafes nearby where you earn points for visiting cafes. Cafe.connor.fun acts as a test case/demo application for the Lighter Web Framework which is a web framework written from scratch to be easy-to-use, compile-time safe, and highly performant. Instead of using reflection, Lighter uses compile time code generation to write together the application. It provides an annotation based API similar to JAX-RS. The Lighter GitHub repository (which is a submodule of this repository) contains a readme with more information about how Lighter works and why it is interesting.

3. Features Implemented

UC-ID	Description
UC-01	Users should be able to view nearby cafes (partial)
UC-02	Details of any cafe should be accessible to the user
UC-04	User account details must be visible to each users. User should only be able to view their own account
UC-05	Users can check-in at a cafe to earn points
UC-06	Users can view the potential score of checking in at a cafe

4. Features not implemented

UC-ID	Description
UC-03	Cafes should be searchable - the search should prioritize by location
UC-07	Users can view high-score tables that show the top scores among all users

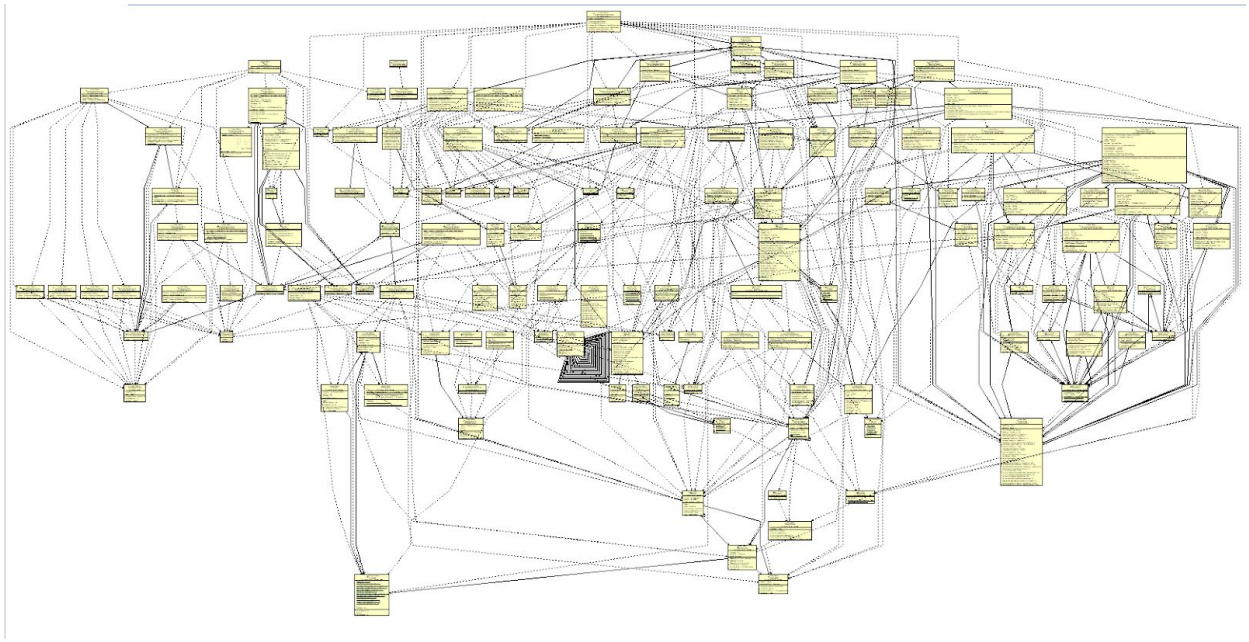
5. Final Class Diagram and Changes

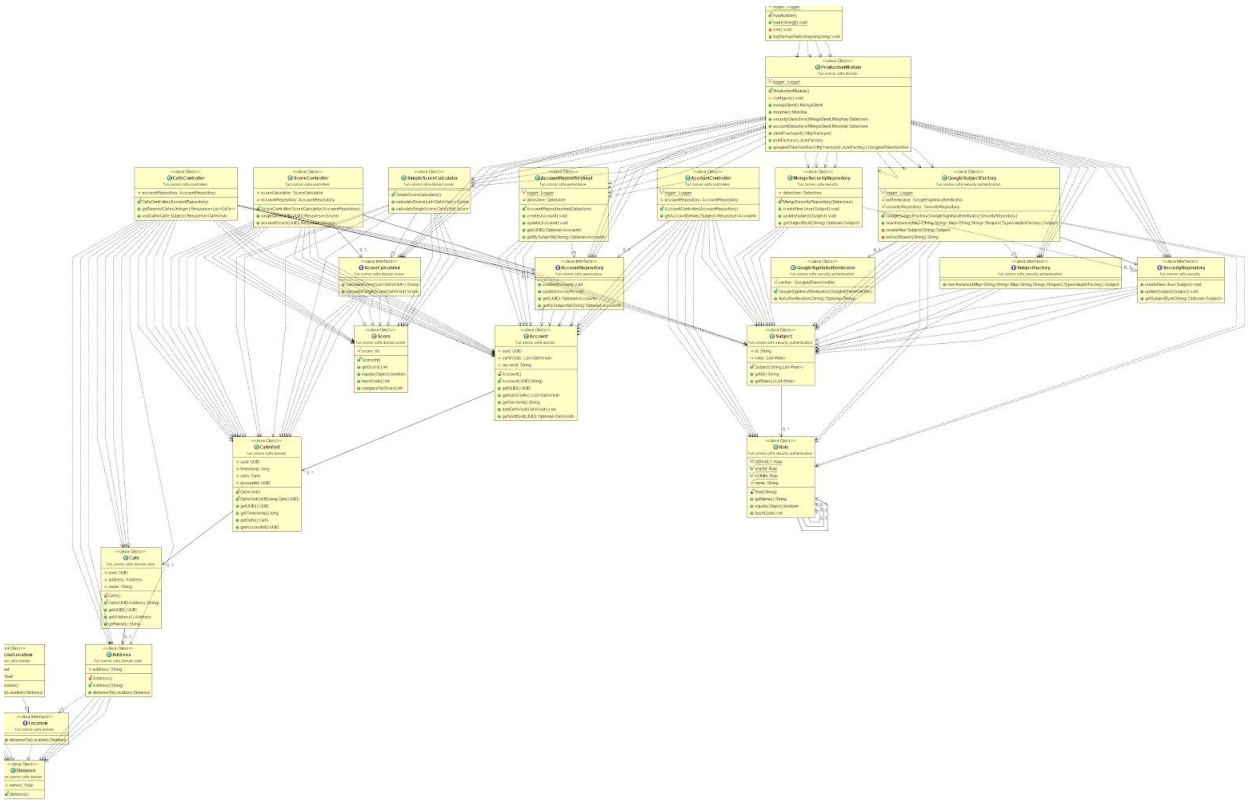
The design of cafe.connor.fun did not change very much. The domain model was simplified somewhat when it became apparently that UC-07 and UC-03 were out of scope for the project.

Lighter changed a lot more than cafe.connor.fun did. Some aspects of Lighter's design (such as the Response API and the TypeAdapter API) were redesigned to be more usable. Other aspects of Lighter changed to allow for better dependency management (Lighter was refactored so that most logic can be included as compile-only dependencies). The code generation package in Lighter was also refactor to be more flexible (and is now implemented using composable elements and generators). There was also feature creep during the development of Lighter where new features were added because of unforeseen requirements or because I felt they were required to build a satisfactory API. An example of this is the addition of the RequestGuard API. These features were inspired by similar API (with the same name), in the Rocket web framework.

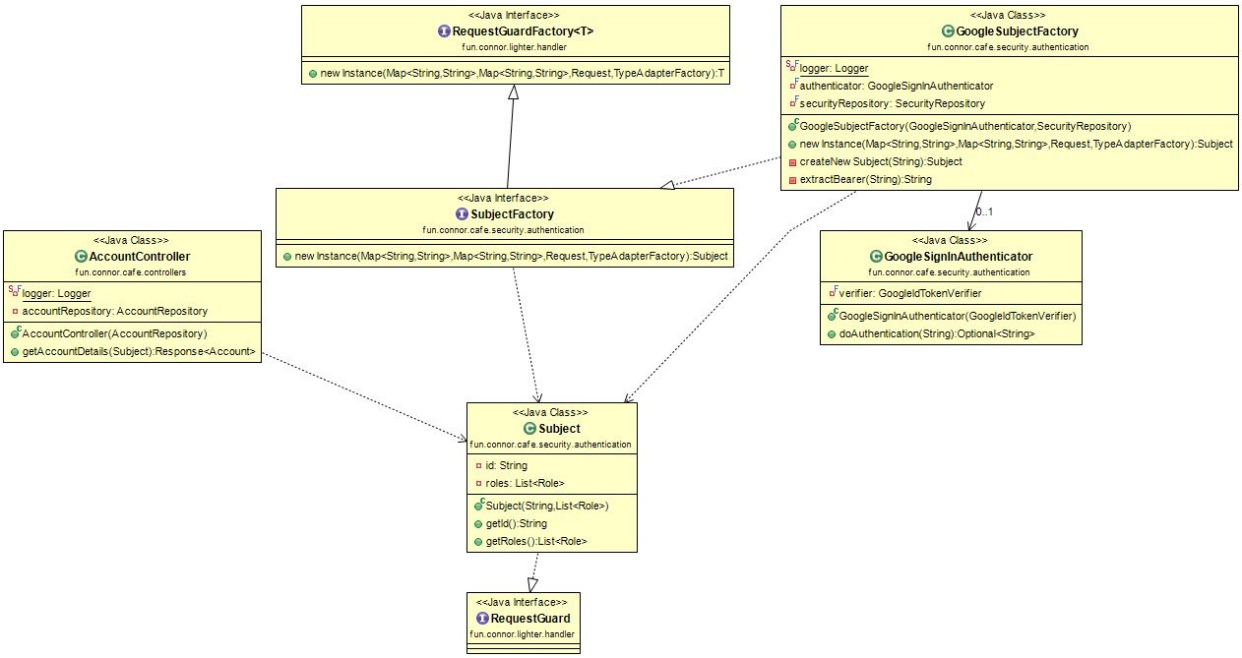
Even with these changes, designing the system at the beginning was very helpful. While implementation details changed and individual components were reworked, the broad strokes design of Lighter did not really change during development. In fact, much of the original class diagram and the final class diagram are the same. New features merely expanded on what was already there. For cafe.connor.fun, very little changed from the original design. This made implementing cafe.connor.fun fairly straightforward. This was good because cafe.connor.fun's implementation started late in development (as Lighter became a larger and larger part of the project over time).

The entire class diagram:

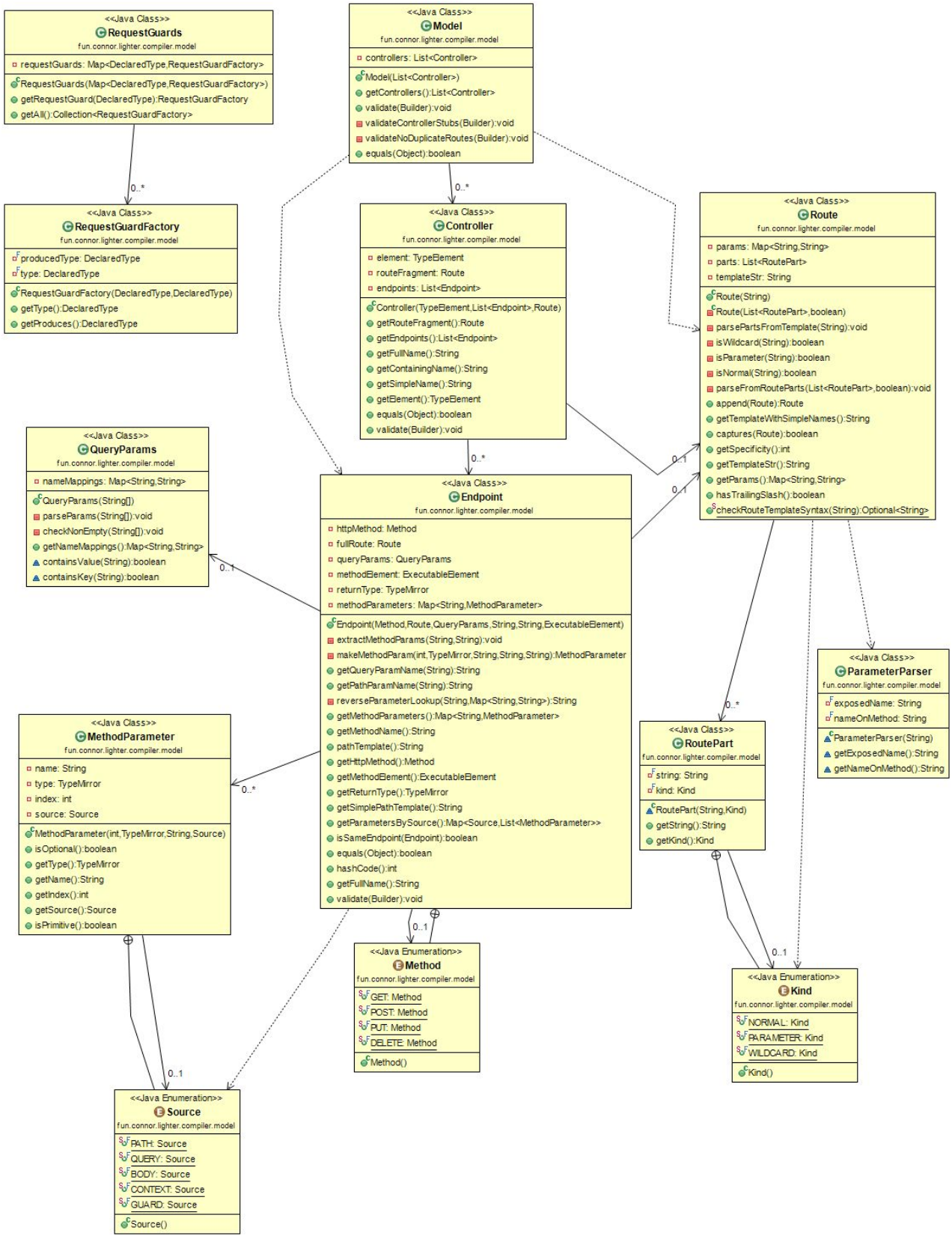




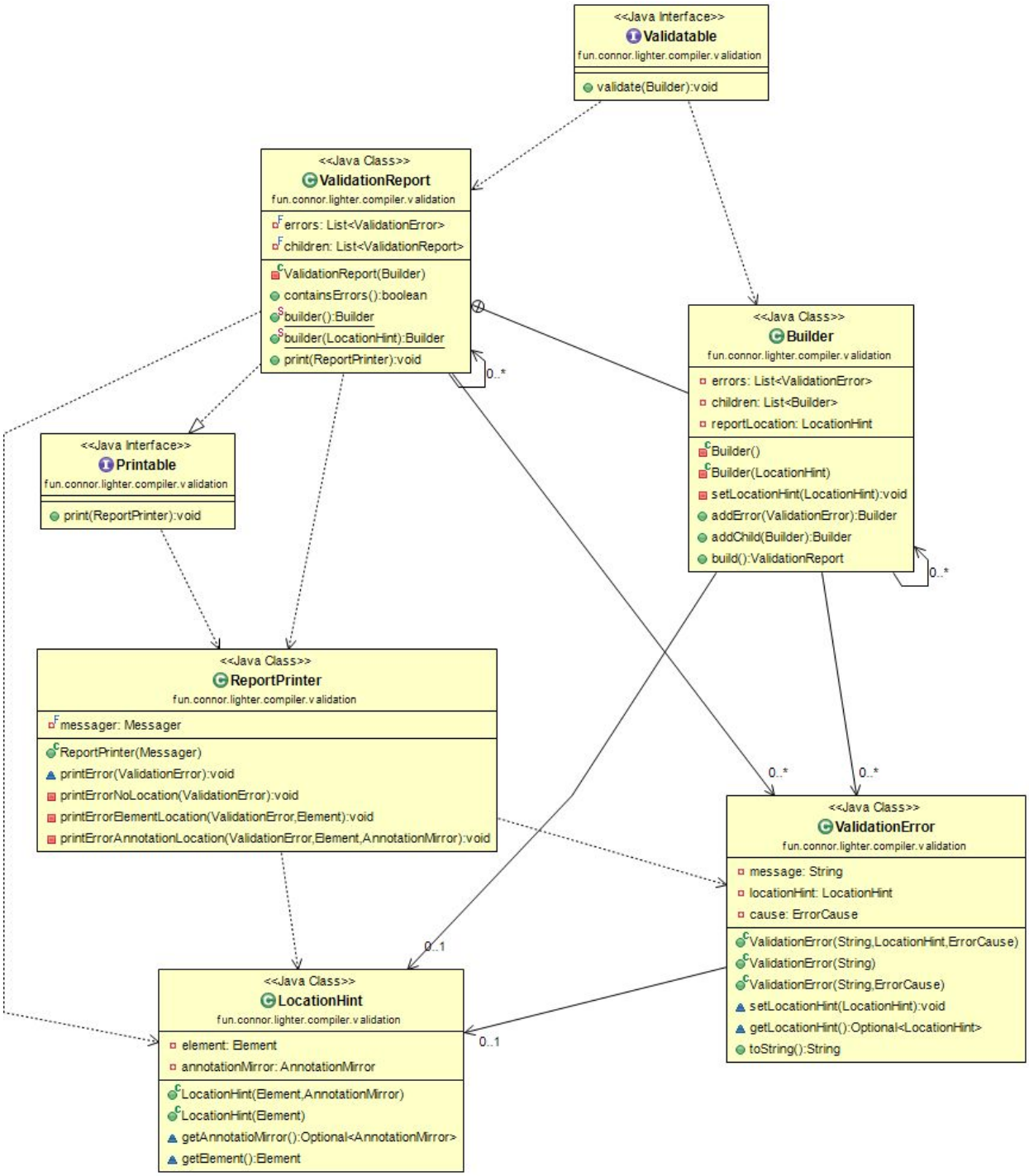
Some zoomed in parts of the diagram that are neat:



Above is the cafe.connor.fun usage of RequestGuards for authentication. Demonstrates how the RequestGuard API allows the Account control to be decoupled from the authentication logic it depends on.


<<Java Class>>
RequestGuards
fun.connor.lighter.compiler.model
requestGuards: Map<DeclaredType,RequestGuardFactory>
RequestGuards(Map<DeclaredType,RequestGuardFactory>)
getRequestGuard(DeclaredType):RequestGuardFactory
getAll():Collection<RequestGuardFactory>
<<Java Class>>
Model
fun.connor.lighter.compiler.model
controllers: List<Controller>
Model(List<Controller>)
getControllers():List<Controller>
validate(Builder):void
validateControllerStubs(Builder):void
validateNoDuplicateRoutes(Builder):void
equals(Object):boolean
0..*
<<Java Class>>
RequestGuardFactory
fun.connor.lighter.compiler.model
producedType: DeclaredType
type: DeclaredType
RequestGuardFactory(DeclaredType,DeclaredType)
getType():DeclaredType
getProduces():DeclaredType
<<Java Class>>
Controller
fun.connor.lighter.compiler.model
element: TypeElement
routeFragment: Route
endpoints: List<Endpoint>
Controller(TypeElement,List<Endpoint>,Route)
getRouteFragment():Route
getEndpoints():List<Endpoint>
getFullName():String
getContainingName():String
getSimpleName():String
getElement():TypeElement
equals(Object):boolean
validate(Builder):void
<<Java Class>>
Route
fun.connor.lighter.compiler.model
params: Map<String,String>
parts: List<RoutePart>
templateStr: String
Route(String)
Route(List<RoutePart>,boolean)
parsePartsFromTemplate(String):void
isWildcard(String):boolean
isParameter(String):boolean
isNormal(String):boolean
parseFromRouteParts(List<RoutePart>,boolean):void
append(Route):Route
getTemplateWithSimpleNames():String
captures(Route):boolean
getSpecificity():int
getTemplateStr():String
getParams():Map<String,String>
hasTrailingSlash():boolean
checkRouteTemplateSyntax(String):Optional<String>
<<Java Class>>
QueryParams
fun.connor.lighter.compiler.model
nameMappings: Map<String,String>
QueryParams(String[])
parseParams(String[]):void
checkNonEmpty(String[]):void
getNameMappings():Map<String,String>
containsValue(String):boolean
containsKey(String):boolean
0..1
<<Java Class>>
Endpoint
fun.connor.lighter.compiler.model
httpMethod: Method
fullRoute: Route
queryParams: QueryParams
methodElement: ExecutableElement
returnType: TypeMirror
methodParameters: Map<String,MethodParameter>
Endpoint(Method,Route,QueryParams,String,String,ExecutableElement)
extractMethodParams(String,String):void
makeMethodParam(int,TypeMirror,String,String,String):MethodParameter
getQueryParamName(String):String
getPathParamName(String):String
reverseParameterLookup(String,Map<String,String>):String
getMethodParameters():Map<String,MethodParameter>
getMethodName():String
pathTemplate():String
getHttpMethod():Method
getMethodElement():ExecutableElement
getReturnType():TypeMirror
getSimplePathTemplate():String
getParametersBySource():Map<Source,List<MethodParameter>>
isSameEndpoint(Endpoint):boolean
equals(Object):boolean
hashCode():int
getFullName():String
validate(Builder):void
0..1
0..1
<<Java Class>>
ParameterParser
fun.connor.lighter.compiler.model
exposedName: String
nameOnMethod: String
ParameterParser(String)
getExposedName():String
getNameOnMethod():String
<<Java Class>>
MethodParameter
fun.connor.lighter.compiler.model
name: String
type: TypeMirror
index: int
source: Source
MethodParameter(int,TypeMirror,String,Source)
isOptional():boolean
getType():TypeMirror
getName():String
getIndex():int
getSource():Source
isPrimitive():boolean
0..*
<<Java Class>>
RoutePart
fun.connor.lighter.compiler.model
string: String
kind: Kind
RoutePart(String,Kind)
getString():String
getKind():Kind
0..*
<<Java Enumeration>>
Method
fun.connor.lighter.compiler.model
GET: Method
POST: Method
PUT: Method
DELETE: Method
Method()
<<Java Enumeration>>
Kind
fun.connor.lighter.compiler.model
NORMAL: Kind
PARAMETER: Kind
WILDCARD: Kind
Kind()
0..1
<<Java Enumeration>>
Source
fun.connor.lighter.compiler.model
PATH: Source
QUERY: Source
BODY: Source
CONTEXT: Source
GUARD: Source
Source()

Above is the diagram of the compile time application model classes. Each class represents an implicit component of the application generated from the user's code. Interesting to note here is how the RequestGuard model is completely separate from the rest of the application model.

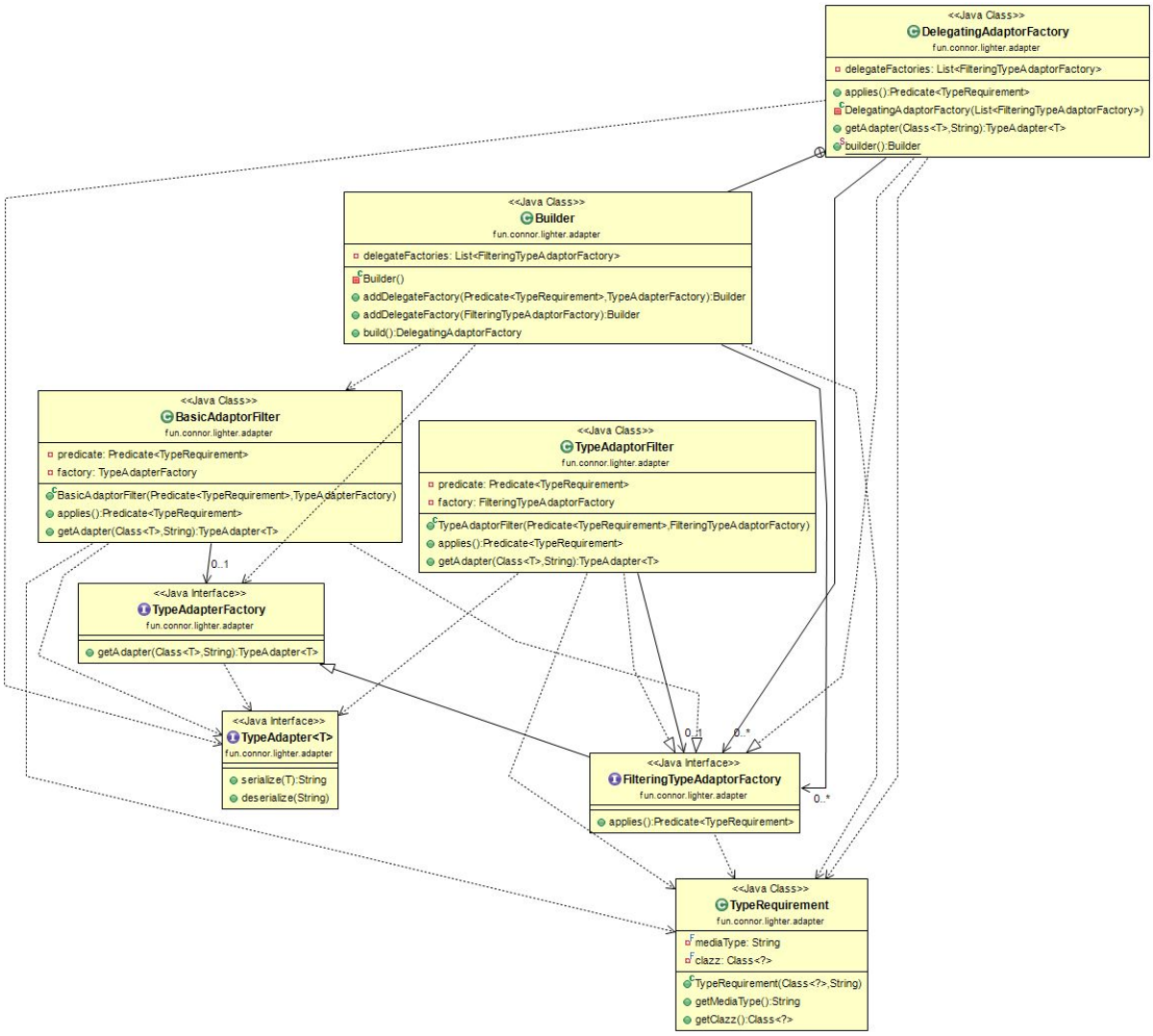


Above is the compiler validation report package. The classes here use a composite pattern to provide a validation tree that can accurately locate the source of errors during compile time.

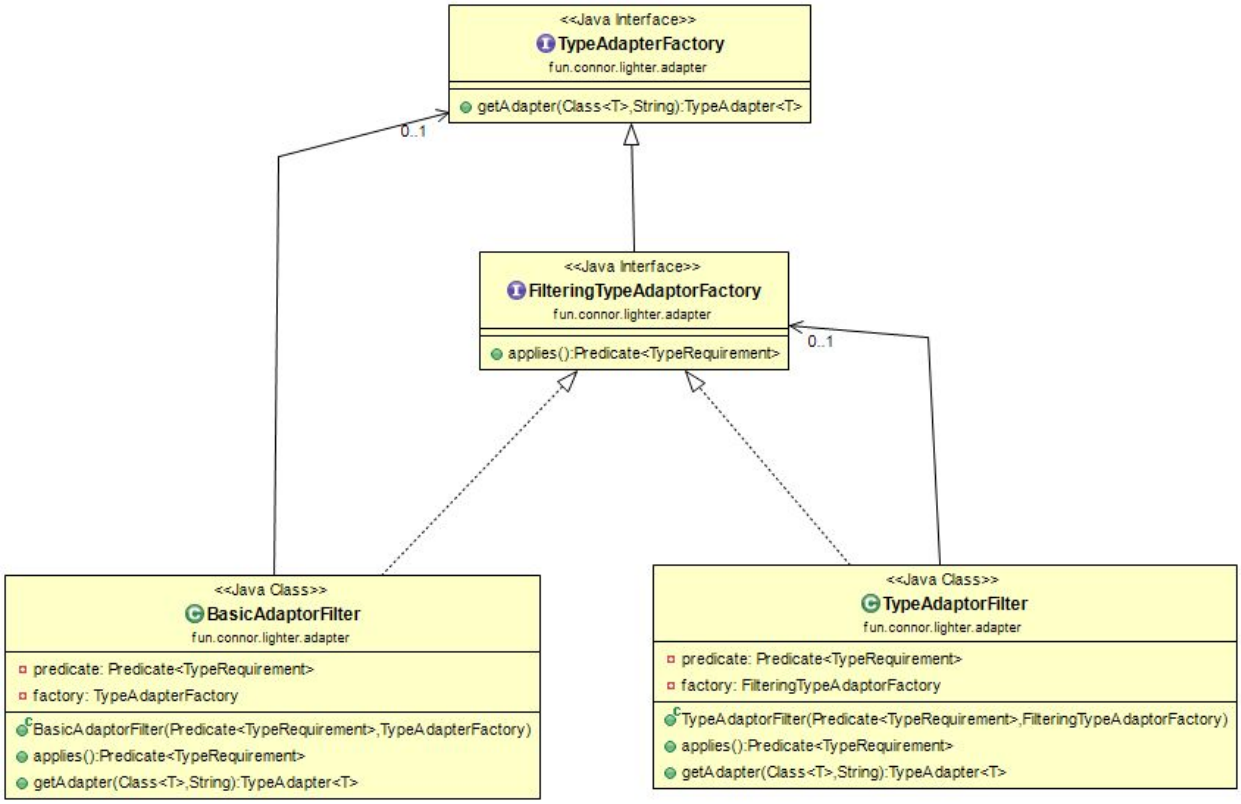
6. Design Patterns

Since my project is made up of more than 200 classes, it is not feasible for me to go over every design pattern. Instead, I will go over a couple interesting ones.

TypeAdapter API: Decorator, Factory, Delegate, Composite

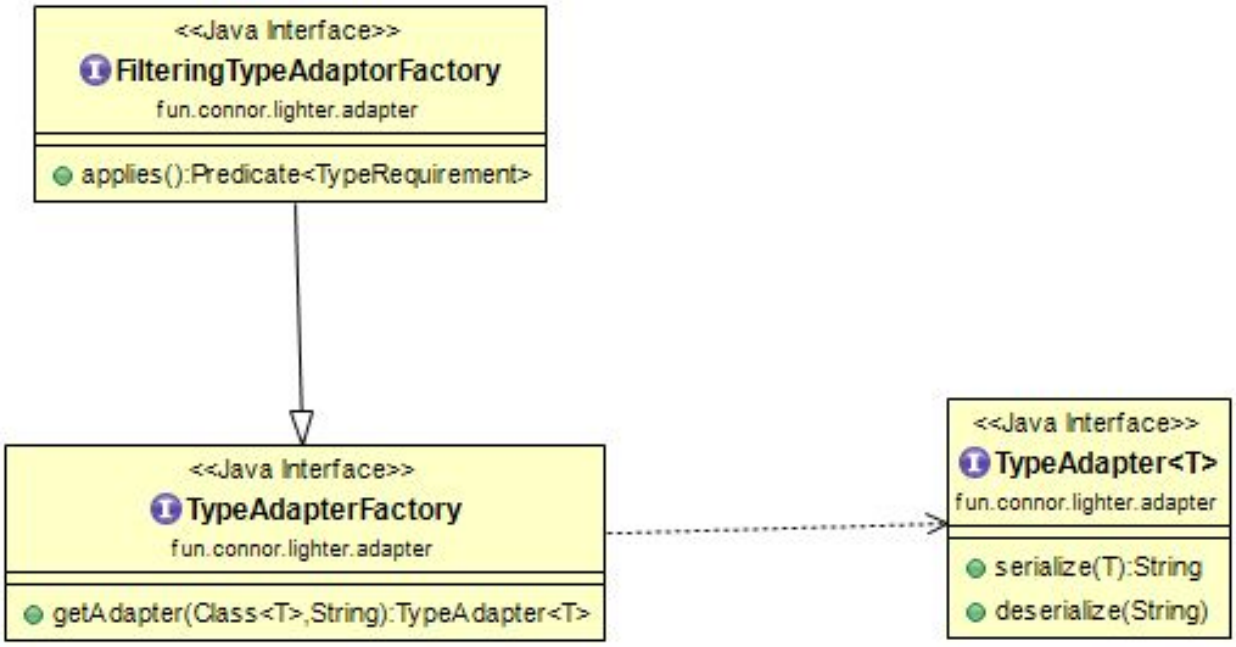


These 8 classes actually implement a Decorator, Factory, Delegate, and Composite pattern. This is difficult to see from the class diagram above, so I created subsets of this diagram below that demonstrate this better.

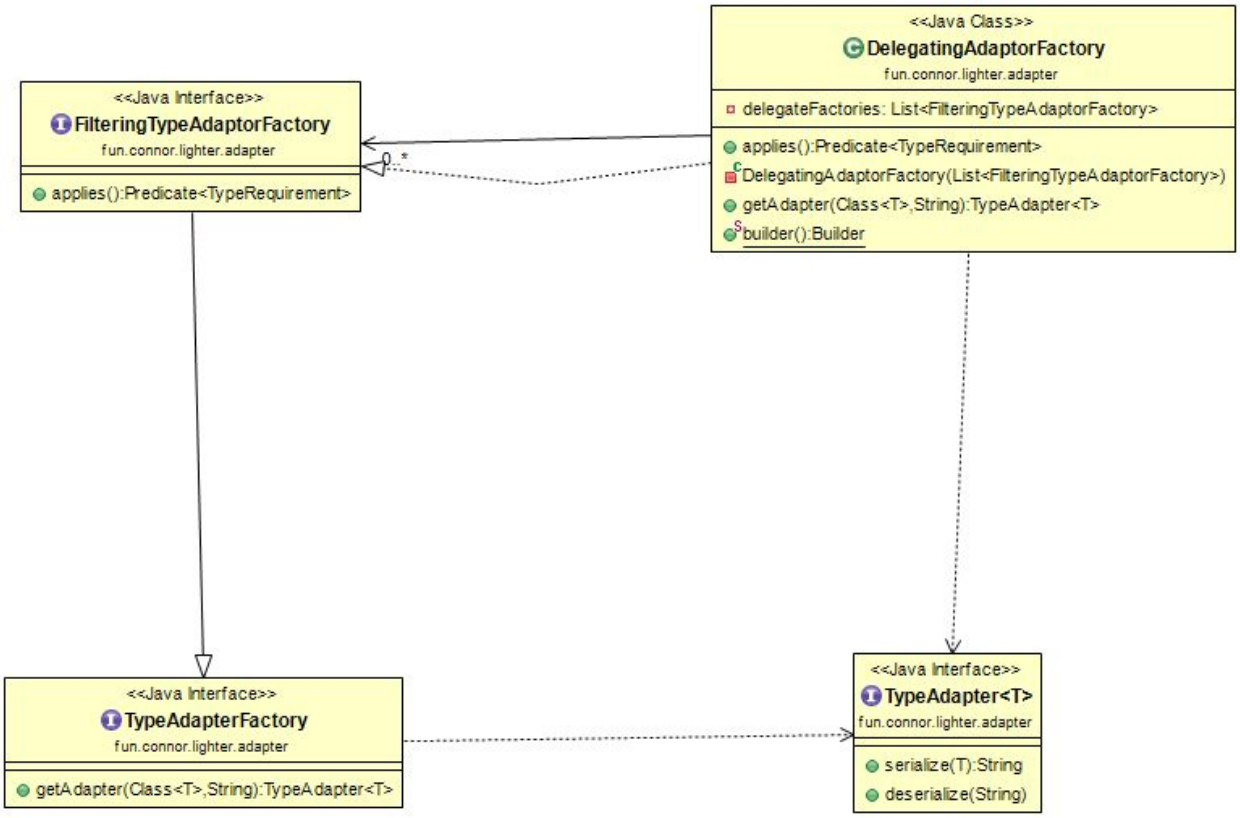


The above diagram shows that the TypeAdapterFilter forms a decorator for FilteringTypeAdapterFactory. This pattern is modified a bit to include the superinterface TypeAdapterFactory. The BasicAdaptorFilter acts as a decorator that converts a TypeAdapterFactory into its subinterface. The reason for this modification is to improve the API these classes provide. It is important to note that this API is intended to be programmed against. The application developer is expected to implement FilteringTypeAdaptorFactory with custom classes. Lighter includes some basic implementations of these that are not included in this diagram to improve clarity.

Another interesting thing to note is that the TypeAdaptorFilter is a concrete class. Different decorators are constructed through composition, not inheritance. This is made possible by using Java 8 FunctionalIntefaces. Different filters can be created by providing different java.util.Predicates during construction.



This simple breakdown shows that the TypeAdapterFactory is a factory pattern. Once again, no concrete implementations are shown for clarity.

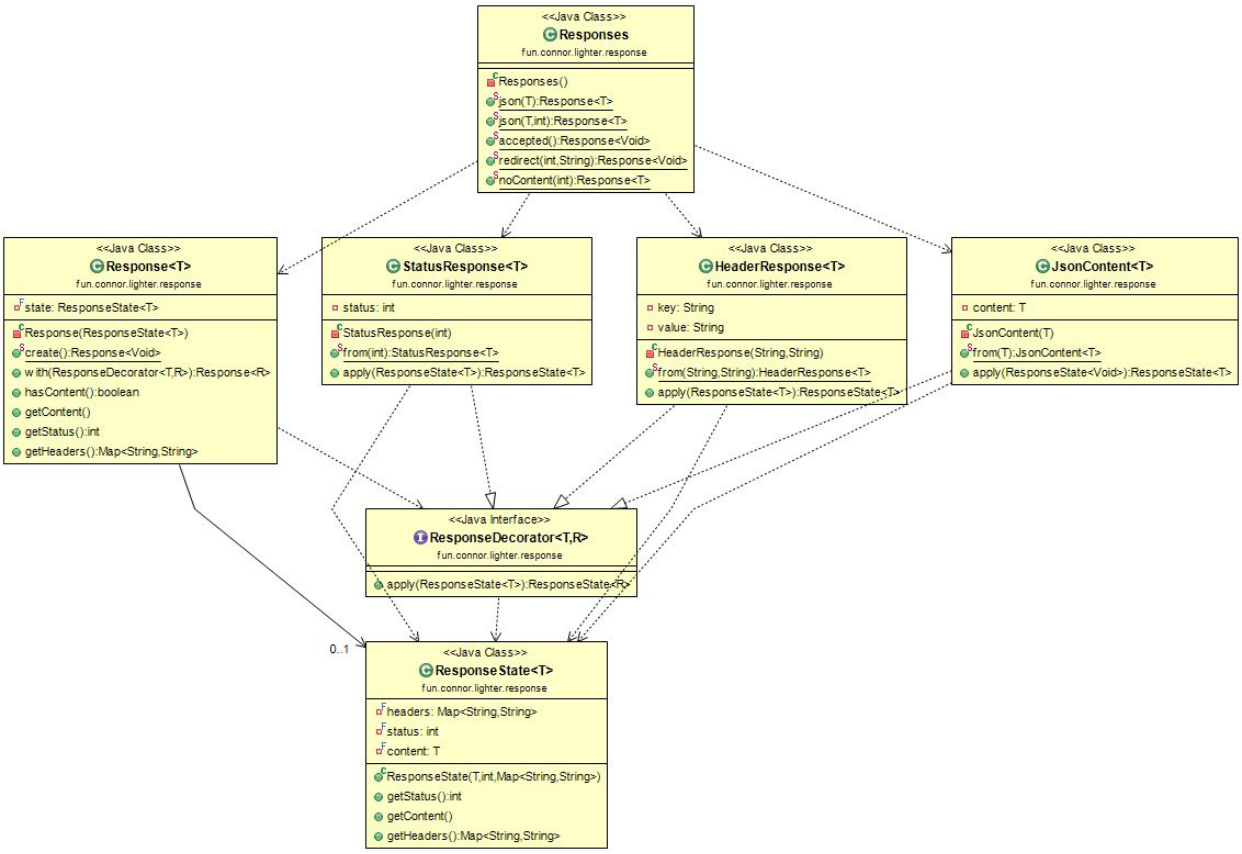


This diagram shows that the FilteringTypeAdaptorFactory and the DelegatingAdapterFactory form a composite pattern. The DelegatingAdapterFactory is a FilteringTypeAdapterFactory and is also a composition of them. This composition is used to build the JavaTypesAdapterFactory

in the Lighter provided TypeAdapter implementations. The combination of this TypeAdapterFilter and the DelegatingAdaperFactory allow arbitrary predicate logic to be expressed through simple object composition. Since a single TypeAdapterFactory is used to provide all of the TypeAdapters for a Lighter application, this flexibility is very important.

This class diagram also demonstrates that these classes follow a Delegation pattern. The DelegatingAdapterFactory, as the name implies, delegates the construction of TypeAdapters to the factories it contains.

Response API: State, Decorator (sorta)

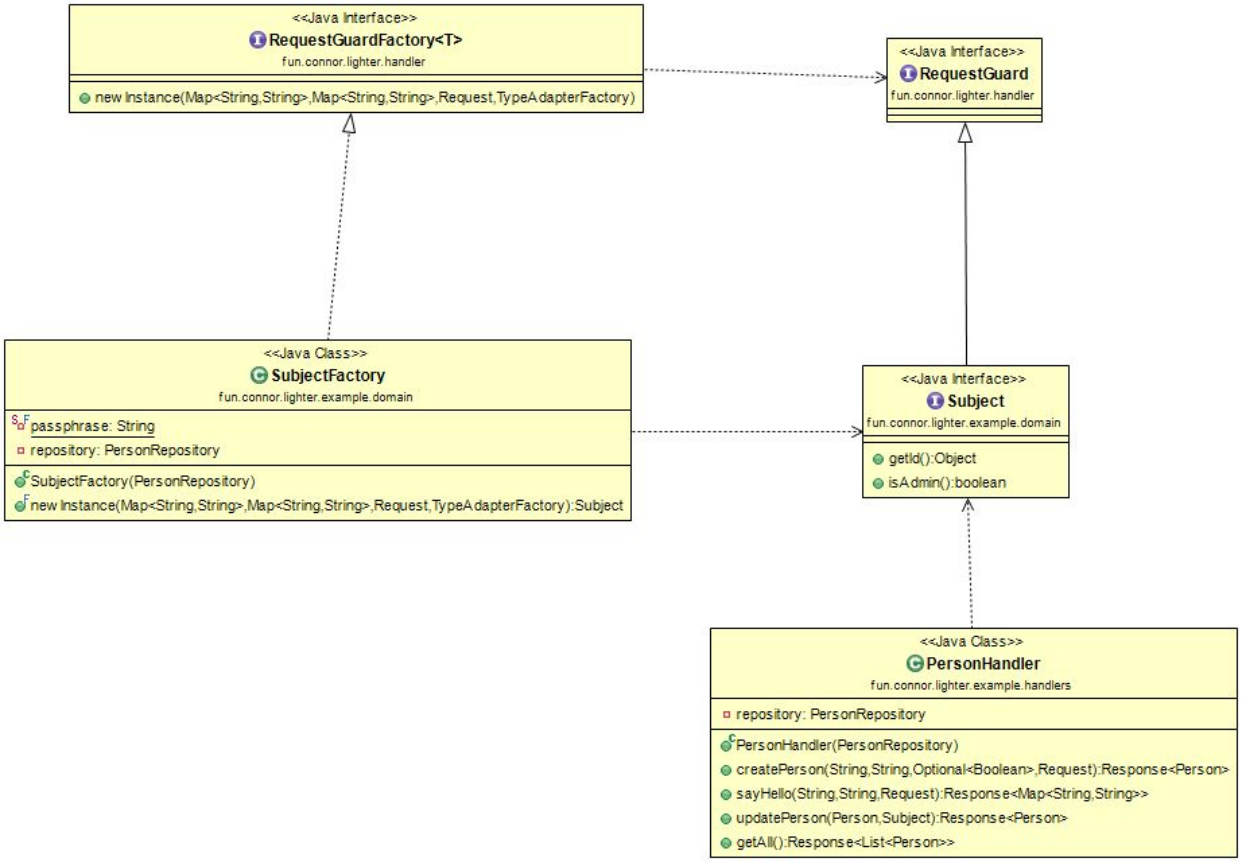


The Response API uses a simplified version of the state pattern. While the state pattern is typically used to decouple an object's state with it's interface, it is used here to allow the object to provide a more readable, fluent interface. This package also contains an interface called "ResponseDecorator". This name is a bit misleading as the class does not behave as a decorator. Instead, ResponseDecorator solves the same problem as a decorator: applying it to a Response object modifies the object's behavior. In order to do this, the response decorator replaces the underlying state of the Response with a state it constructs. Implementing the API this way allows for more human readable API calls than a normal decorator pattern:

```
return Response.create()
        .with(JsonContent.from(content))
        .with(StatusResponse.from(status));
```

This is equivalent to creating a response and decorating it with a JsonContentDecorator and a StatusResponseDecorator. However, I believe the fluent version of the API is significantly more readable.

RequestGuard API: Factory



This diagram shows the RequestGuard, RequestGuardFactory, an implementation of both these interfaces, and a consumer of the implementation. The diagram shows that the consumer of the Subject (which implements RequestGuard) has no relationship to the SubjectFactory. Thus the factory pattern is used to decouple the construction logic for the request guard from the consumer of the request guard. This allows for cross-cutting concerns to be implemented as RequstGuard factories without the potential for their logic to become tightly coupled with endpoint handler implementations. The factory pattern used here also allows Lighter to control when RequestGuards are constructed while still allowing applications to define their own construction logic. This API is named after the API in the Rocket web framework that inspired its development.

7. What Did I Learn

Working in this project taught me a lot about creating designs that a resistant to changing requirements and changing implementations. While developing Lighter in particular, requirements for what the framework should do and how the APIs should work was constantly changing. On top of that, the compile-time code generation added an extra layer of overhead when trying to change any of Lighter's core interfaces. This caused me to focus on creating APIs and packages that were resistant to change. In some cases, I was successful. The Request API was completely rewritten from scratch without requiring any other changes in the entire framework. In other cases, I wasn't. Making a small change to the TypeAdapter interface requires hours of work modifying code generators. This process has helped me understand what types of design are flexible enough to change and which are not. Lighter is also a relatively large project. Designing Lighter's APIs to be modular and loosely coupled while still being consistent proved to be difficult and taught me a lot about object oriented API design.