Lighter Documentation

Release Minimum Viable Product

Ethan Hunter

Jul 21, 2019

Overview

Lighter is a fully-featured web framework for Java. It's primarily designed for RESTful services, but it can be used for all sorts of other things too!

Lighter is different than most other Java web frameworks. Lighter stands inbetween micro-frameworks like Spark Java and giant monolithic frameworks like Spring Boot. Like a micro-framework, Lighter is small and doesn't come with a lot of dependencies. It lets you choose your own serialization, persistance, and dependency injection solutions. Like a monolithic framework, Lighter provides declarative configuration and high-level abstractions.

Lighter achieves this by working at *compile-time instead of run-time*. Lighter uses almost no reflection. Instead, it depends on annotation processors to provide high level constructs. This allows Lighter's abstractions to have close to zero cost.

Lighter aims to be the anti-framework framework. Whenever possible, it achieves abstraction without magic. When it does use magic, Lighter focuses on making it inutitive and easy to follow. Lighter avoids pulling in dependencies it doesn't have to and lets the developer structure their application. Lighter provides abstraction without incuring performance penalties or clarity costs.

For complete API documentation, check out the javadocs (link TBD).

TESE DOCS ARE STILL A WORK IN PROGRESS. Almost every page still has a lot of work that needs to be done. Many pages have not been started.

CHAPTER 1

Introduction

This section will go into detail about Lighter's design and architecture.

Lighter's goals:

1. Easy to use. Lighter's APIs should be intuitive and straight forward. Behavior should be easy to reason about.
2. Safe. Lighter should find errors at compile-time, not run-time. Lighter should take advantge of Java's type system to provide type safe interfaces.
3. Performant. Lighter should start-up quickly and incur little overhead at runtime.
4. Testable. Lighter applications should be easy to test. Lighter should not force applications to use complex test constructs just to run unit-tests.
5. Modular. Lighter should not make decisions for the application. It should be easy to switch out components as required. Lighter shouldn't pull in 10s of megabytes of dependencies.

This documentation is intended to provide details about how to use Lighter and about how Lighter works. For API reference, refer to the `javadoc`.

Note: *(from the author, Spaceman1701)*

Currently, Lighter's Minimum Viable Product version is available. This version is designed to demonstrate the feasability of Lighter. However, it does not have the final feature set or the final APIs. Many of the APIs are awkward to use or broken.

With this said, Lighter's MVP version works quite well. It performs very well when compared to Spring Boot and is usable for real-world applications. I've decided I'm going to keep working on Lighter. I'll be using it for personal projects whenever I can. I'll also continue to improve the feature set and APIs. I hope that at some point I'll be able to consider Lighter as more than a proof of concept.

If somehow you've come accross these docs and you want to contribute to Lighter, head over to GitHub. I'd love contributions. As I move forward, I'll be tracking features and issues on GitHub.

CHAPTER 2

Lighter Architecture

2.1 Basics

Lighter is built of two libraries: The runtime library, `lighter-core`, and the compile-time library, `lighter-compiler`.

Lighter core provides the the stock JBoss Undertow implementations of the core Lighter APIs. It also defines the declarative annotation API.

Lighter compiler consumes the declarative annotation API. It is responsible for providing compile-time verification and for generating application-speicifc implementations for Lighter's abstractions. The compiler uses compile-time reification of the application to do verification and code generating.

2.2 Terminology

Some important terms for the rest of this documentation.

Lighter The Lighter Web Framework. This will be used to mean the framework as a whole (as opposed to individual components).'

The Application The actual application that is built using Lighter. This term is used to mean "any application" as opposed to refering to a speicifc one. The Application consumes Lighter.

Application Developer The developer who is using Lighter to implement her application.

Lighter Core (the Core) The Lighter runtime library and APIs. The stock implementation is `lighter-core`. When used in this documentation, it usually refers to the stock implementation.

Lighter Compiler (the Compiler) The Lighter compile-time library. The stock implementation is `lighter-compiler`. For what it's worth, implementations of Lighter Compiler do not necessarily need to be compile-time only. Essentially, the Lighter Compiler is an invisible provider for the implementations of Lighter's high-level abstractions. Since the Compiler is invisible to the application, it's implementation ins't necessarily important. In the future, reflection-based implementations or byte-code weaving-based implementations might be possible. When used in this documentation, it usually refers to the stock implementation.

Lighter Backend The runtime-level implementation of Lighter. This refers to any Lighter APIs that are implemented by runtime code. The most significant component of the Lighter backend is the web server.

2.3 Interop Between Lighter Core and Compiler

Both libraries are built as seperate entities. In fact, it's possible to use Lighter Core without using Lighter Compiler. However, many of the high-level abstractions that make Lighter pleasent to use are not available without the compiler. Since the application only depends on Lighter Core, this decoupling means that it's possible for different implementations of Lighter Compiler to be used without changing the application.

Since Lighter is currently in MVP, the actual interop between the Core and Compiler components is very limited. However, future versions of Lighter will have a well-defined API for both components to use. This formalization of the API has a couple benifits.

1. Improved versioning. It would be possible to version the Core and Compiler components independantly.
2. Support for application libraries. A Core-Compiler API would be able to provide functionality for already-compiled applications to be used as libraries
3. Multiple Compiler implementations. This one's obivious. It'd be interesting to see reflection-based implementations in the future, for example.

2.4 Compiler Components

The Lighter is iterative. Most iterations do some form of reification on The Application's code. Other iterations attempt to verify that some set of invariants hold in the reified model. As such, it makes sense to break the compiler into components defined by which reified model they use.

- **Annotation Model Components - a lightweight model which represents the locations and data of each Lighter annotation i**
  - Annotation Validators - validation that annotations are placed correctly and do not have data errors
  - Model Builder - uses the Annotation model to build a more detailed model
- **Application Model - a detailed reified model of The Application's structure**
  - Model Validators - validation that the model represents a legal application that will work at runtime
  - Dependency Collection - collect all of the non-Lighter classes that are required by the model
  - Request Guard Collection - collects request guards that might be used by The Application
  - Controller Generation - code generation for HTTP endpoint controllers. Produces a new model based on the generated code
- **Generated Code Model - reified model of the application refering to actual generated objects**
  - Reverse Injection Generation - generate an injector for handling application depedencies
  - Route Configuration Generation - generate code for configuration for using generated endpoint handlers

To aid with each of these components, the Compiler also contains components for managing and reporting erros, generating dynamic code, and defining compilation steps.

Currently, the "Generated Code Model" lacks proper definition in the actual implementation of Lighter Compiler. Cleaning up the compiler code will be a major focus in future versions of Lighter.

The Lighter Compiler is very complex. More detail about each of these major components will be provided further in the documentation.

2.5 Core Components

Lighter's APIs can be subdivided into a couple catagories.

- Declarative - annotations used to identify components of the application.
- Request and Response - used to construct and represent HTTP requests and responses
- TypeAdapter - pluggable API for defining serialization and deserialization procedures
- Injection - pluggable API for depedency management
- Autoconfig - API for using configuration generated by Lighter Compiler.

Lighter Core also provides the backend implementation. The stock Lighter Core implementation provides a backend which uses JBoss Undertow as a web server.

CHAPTER 3

Getting Started With Lighter

This section will provide a small tutorial to help newcomers learn how to use Lighter.

CHAPTER 4

Using Lighter

This section will provided overviews for all of the concepts required to develop real applications with Lighter. This isn't API documentation. Instead, each page will contain a detailed overview of one of Lighter's core concepts. This will be useful for determining what features can solve which problems. Each page will also contain examples where appropriate.

CHAPTER 5

Endpoints

Endpoints are the core of any Lighter application. They allow the application to interact with the outside world. In Lighter, endpoints are methods that are identified using an endpoint annotation. In the current MVP version of Lighter the endpoint annotations are `@Get`, `@Post`, `@Put`, and `@Delete`. Each of these annotations corresponds to an HTTP method.

Endpoints must always return a `Response`. See the *Response API* docs page for details about constructing responses.

5.1 Endpoint Annotations

All of the endpoint annotations have the same API. Each one has an optional `value` field which can be used to define a path template stub that the endpoint method should respond to. The full path template that defines the endpoint is constructed by prepending the endpoint's Resource Controller path stub to the stub provided in the endpoint annotation. See more about this in the Resource Contollers section below.

In order to handle HTTP requests, Lighter matches the request method and path against the set of endpoint methods and path templates in the application. Method parameters are fulfilled by path parameters, query parameters, and the request body.

5.2 Resource Controllers

Every endpoint method must be a member of a `@ResourceController` annotated class. Resource Controllers are plain Java classes. Resource Controllers must specify a path template stub that will be prepended to all of their members. This is useful as it avoids the necessity of rewriting parts of a the template multiple times for related endpoints.

Resource Controllers will be instaintiated by Lighter. Thus, they must be instaintiable by the `InjectionObjectFactory`. See the docs page on the Injection API for details.

5.3 Path Template Syntax

Path template syntax is similar to other web frameworks. Templates can contains three types of components: Normal, Parameter, and Wildcard. Normal components match components exactly equal to themsevles. A path template made of only Normal components would match only paths that are identical to it. Parmaeter components will match anything and bind it to the provided name. Parmaeters are denoted by surrounding a name with `{` and `}`. Every parameter as a type which is inferred from the method signature. Wildcard components are denoted by a `*` and greedly match any number of components.

Here are some examples:

The template `foo/bar/123` will match exactly the path `foo/bar/123` and nothing else.

The template `foo/bar/{id}` will match any path with exactly 3 components that begins with `foo/bar/`. The third component of the path will be bound to the name "id".

The template `foo/bar/*` will match any path that begins with `foo/bar/`.

The template `foo/*/bar` will match any path that begins with `foo` and ends with `bar`

5.4 Query Parameters

HTTP query parameter bindinds can be specified in a similar way to path Parmaeter components. However, query parameters do not appear as part of the path template. Instead, the `@QueryParameters` annotation is used to provide a list of name bindings. Since the names of query parameters are exposed as part of the applications API, Lighter allows external and internal names of query parameters to differ. The external (*exposed*) name is what HTTP calls should use. The internal (*mapped*) name should match the name of the parameter on the Java endpoint method.

Query parmaeter names are specified using an array of Strings. Exposed names and mapped names are seperated by a `:`. If only one name is provided, Lighter assumes the exposed name is identical to the mapped name.

Here are some examples:

The parameter `foo:bar` specifies an exposed name `foo` which maps to a parameter on the Java method named `bar`

The parameter `foo` specifies an exposed name `foo` which maps to a parmaeter on the Java method named `foo`

Similar to path Parameters, query parameter types are inferred from the Java method.

5.5 Accessing the Request Body

The request body content can be mapped to any method parameter by annotating it with `@Body`. The type of the body content is infered from the method.

5.6 Parameter Type Inference

All endpoint parameter types are inferred from the Java method signature. Any Java type can be used as long as the application `TypeAdapterFactory` is capable of producing a `TypeAdater` for the type. Query and path parameters are assumed to have a MIME Media Type of `text/plain`. The Media Type of the request body is determined by the Content-Type header.

If a method parameter is optional (i.e. an error should not occur if Lighter can not provide data for the parameter), it should have a type of `java.util.Optional` (or one of the allied Optional types provided in the standard library).

Since Lighter performs type inference at compile time, it is able to use the generic type parameter of `Optional` for serialization and deserialization logic.

Lighter **will never** provide a `null` value for a method parameter. If a non-Optional parameter can not be provided for any reason, Lighter will throw an error.

CHAPTER 6

Response API

The response API allows your application to return data to the outside world. Since every endpoint must return a response, the API is designed to be very concise. However, applications will have extemely variable requirements for Response structure, so the API also allows a great deal of flexibility. In addition to this, Responses must be easy to use in unit tests.

The main class that applications will interact with is the `Reponse` class. To the user, `Response` is a Plain Old Java Object. `Response` is immutable and method calls have no side effects. In addition to `Response`, applications will interact with instances of the `ResponseDecorator` functional interface. the `Response#with` method provides a fluent API for adding decorators to the `Response` object. This is the primary way to build custom responses.

Lighter also provides the `Responses` static factory class with utility methods for constructing common HTTP response types. `Responses` has methods for constructing `3xx - Redirect`, JSON content, and no content responses.

The Response API is type safe. The `Response` class type parameter is used to represent the type of the response body content. `ResponseDecorator` application can change the type parameter. This allows chains of decorator application to maintain type safety. `java.lang.Void` is used to represent an empty response.

6.1 Using Response

The `Response` class does not contain the serialized data. Instead, it contains a reference to the Java object that will be serialized. Lighter uses the top level `TypeAdapterFactory` to serialize the content. Lighter ensures that the type is serialized with the correct MIME Media Type by reading the Content-Type header on the response.

When using the `Response` for unit testing endpoints, the Java object is directly available.

6.2 Standard Response Decorators

Lighter provides a few standard response decorators. These allow most required responses to be constructed. Since `ResponseDecorator` is a functional interface, lambda functions can also be used.

The provided decorators are:

- HeaderResponse - adds a header to the response
- StatusResponse - sets the response status code
- JsonContent - adds an object as the response body and sets the content-type header to `application/json`

CHAPTER 7

Request Guards

Request guards allow your application to define preconditions to endpoint execution. This feature is inspired by one of Ligther's primary inspirations: Rocket Web Framework.

Note: This feature is currently in early stages of development. Expect lots of changes.

Request Guards are special endpoint method dependencies that are not constructed directly from the request. Instead, Request Guards are constructed by application defined logic using a `RequestGuardFactory`. Request Gaurds are identified using the `RequestGuard` marker interface. RequestGuardFactories are identified using the `RequestGuardFactory` interface and the `@ProducesRequestGuard` annotation.

Note: The RequestGuardFactory API is an area that is targeted for change in the future. It is very awkward to require both an interface and annotation to mark RequestGuardFactories.

Since Request Guards are constructed by application logic, they can be used to define custom pre-requesite conditions for endpoints. In order to use a Request Guard, the endpoint method must simply add a parameter of a `RequestGuard` type. Lighter will determine how to fulfill that requirement at compile time.

Note: Currently, Lighter does not support Optional Request Guards. This feature will be added soon.

Note: Request guard errors current cause a `500 - Internal Server Error`. In the future, the API will allow more control over how Request Guard construction errors occur.

RequestGuards are the idiomatic way to implement authentication and other cross-cutting concerns.

CHAPTER 8

Injection API

Control how your classes get instantiated by Lighter. The primary class applications will interact with is the `InjectionObjectFactory`. This functional interface is designed to provide a implementation agnostic API for dependency injection containers. The interface is very simple as it is only used when Lighter needs to construct a class for the application.

The interface is designed to match the Guice `Injector#newInstance` method.

The other class used for dependency construction is the `ReverseInjector`. Implementations of `ReverseInjector` provide an instance of `InjectionObjectFactory`. Lighter will automatically generate an implementation of `ReverseInjector` that has a setter for every dependency Lighter will need to construct. The auto generated implementation conforms to Java Beans and javax.Inject standards for dependency Injection. This implementation can be used as a configuration bean with dependency injection frameworks that do not have an Injector class (like Dagger 2).

CHAPTER 9

Lighter API

Construct and interact with the `Lighter` instance. The `Lighter` object represents the application itself. Lighter instances can only be constructed using the `Lighter.Builder` API. This fluent API provides many configuration options for Lighter.

Both `Lighter` and `Lighter.Builder` are interfaces which define what configuration options and operations all Lighter backends must support. Backends can choose to implement extra operations. The Undertow backend (which is currently the only backend), provides only the required methods.

Lighter runs asycronously. `Lighter#start` returns as soon as the server is started. This allows the main thread to be used for controlling Lighter.

CHAPTER 10

Automatic Configuration

Control how generated code is used. Autoconfiguration can be accessed using the `AutoConfigurationFactory`. This singleton factory class can be used to access the configuration objects that Lighter generates at compile time. Normally, applications will access this class to load the route configuration instance to pass to `Lighter.Builder`.