Dropwizard Documentation

Release @project.version@

Coda Hale

Oct 09, 2019

Contents

Dropwizard pulls together **stable**, **mature** libraries from the Java ecosystem into a **simple**, **light-weight** package that lets you focus on *getting things done*.

Dropwizard has *out-of-the-box* support for sophisticated **configuration**, **application metrics**, **logging**, **operational tools**, and much more, allowing you and your team to ship a *production-quality* web service in the shortest time possible.

CHAPTER 1

Getting Started

***Getting Started* will guide you through the process of creating a simple Dropwizard Project: Hello World. Along the way, we'll explain the various underlying libraries and their roles, important concepts in Dropwizard, and suggest some organizational techniques to help you as your project grows. (Or you can just skip to the fun part.)**

1.1 Overview

Dropwizard straddles the line between being a library and a framework. Its goal is to provide performant, reliable implementations of everything a production-ready web application needs. Because this functionality is extracted into a reusable library, your application remains lean and focused, reducing both time-to-market and maintenance burdens.

1.1.1 Jetty for HTTP

Because you can't be a web application without HTTP, Dropwizard uses the Jetty HTTP library to embed an incredibly tuned HTTP server directly into your project. Instead of handing your application off to a complicated application server, Dropwizard projects have a `main` method which spins up an HTTP server. Running your application as a simple process eliminates a number of unsavory aspects of Java in production (no PermGen issues, no application server configuration and maintenance, no arcane deployment tools, no class loader troubles, no hidden application logs, no trying to tune a single garbage collector to work with multiple application workloads) and allows you to use all of the existing Unix process management tools instead.

1.1.2 Jersey for REST

For building RESTful web applications, we've found nothing beats Jersey (the JAX-RS reference implementation) in terms of features or performance. It allows you to write clean, testable classes which gracefully map HTTP requests to simple Java objects. It supports streaming output, matrix URI parameters, conditional `GET` requests, and much, much more.

1.1.3 Jackson for JSON

In terms of data formats, JSON has become the web's *lingua franca*, and Jackson is the king of JSON on the JVM. In addition to being lightning fast, it has a sophisticated object mapper, allowing you to export your domain models directly.

1.1.4 Metrics for metrics

The Metrics library rounds things out, providing you with unparalleled insight into your code's behavior in your production environment.

1.1.5 And Friends

In addition to Jetty, Jersey, and Jackson, Dropwizard also includes a number of libraries to help you ship more quickly and with fewer regrets.

- Guava, which, in addition to highly optimized immutable data structures, provides a growing number of classes to speed up development in Java.
- Logback and slf4j for performant and flexible logging.
- Hibernate Validator, the JSR 349 reference implementation, provides an easy, declarative framework for validating user input and generating helpful and i18n-friendly error messages.
- The Apache HttpClient and Jersey client libraries allow for both low- and high-level interaction with other web services.
- JDBI is the most straightforward way to use a relational database with Java.
- Liquibase is a great way to keep your database schema in check throughout your development and release cycles, applying high-level database refactorings instead of one-off DDL scripts.
- Freemarker and Mustache are simple templating systems for more user-facing applications.
- Joda Time is a very complete, sane library for handling dates and times.

Now that you've gotten the lay of the land, let's dig in!

1.2 Setting Up Using Maven

We recommend you use Maven for new Dropwizard applications. If you're a big Ant / Ivy, Buildr, Gradle, SBT, Leiningen, or Gant fan, that's cool, but we use Maven, and we'll be using Maven as we go through this example application. If you have any questions about how Maven works, Maven: The Complete Reference should have what you're looking for.

You have three alternatives from here:

1. Create a project using dropwizard-archetype:

```
mvn archetype:generate -DarchetypeGroupId=io.dropwizard.archetypes -
↪DarchetypeArtifactId=java-simple -DarchetypeVersion=[REPALCE WITH A VALID␣
↪DROPWIZARD VERSION]
```

2. Look at the dropwizard-example
3. Follow the tutorial below to see how you can include it in your existing project

1.2.1 Tutorial

First, add a `dropwizard.version` property to your POM with the current version of Dropwizard (which is @project.version@):

```
<properties>
    <dropwizard.version>INSERT VERSION HERE</dropwizard.version>
</properties>
```

Add the `dropwizard-core` library as a dependency:

```
<dependencies>
    <dependency>
        <groupId>io.dropwizard</groupId>
        <artifactId>dropwizard-core</artifactId>
        <version>${dropwizard.version}</version>
    </dependency>
</dependencies>
```

Alright, that's enough XML. We've got a Maven project set up now, and it's time to start writing real code.

1.3 Creating A Configuration Class

Each Dropwizard application has its own subclass of the `Configuration` class which specifies environment-specific parameters. These parameters are specified in a YAML configuration file which is deserialized to an instance of your application's configuration class and validated.

The application we'll be building is a high-performance Hello World service, and one of our requirements is that we need to be able to vary how it says hello from environment to environment. We'll need to specify at least two things to begin with: a template for saying hello and a default name to use in case the user doesn't specify their name.

Here's what our configuration class will look like, full example conf here:

```
package com.example.helloworld;

import io.dropwizard.Configuration;
import com.fasterxml.jackson.annotation.JsonProperty;
import org.hibernate.validator.constraints.NotEmpty;

public class HelloWorldConfiguration extends Configuration {
    @NotEmpty
    private String template;

    @NotEmpty
    private String defaultName = "Stranger";

    @JsonProperty
    public String getTemplate() {
        return template;
    }

    @JsonProperty
    public void setTemplate(String template) {
        this.template = template;
    }
```

(continues on next page)

(continued from previous page)

```
    @JsonProperty
    public String getDefaultName() {
        return defaultName;
    }

    @JsonProperty
    public void setDefaultName(String name) {
        this.defaultName = name;
    }
}
```

There's a lot going on here, so let's unpack a bit of it.

When this class is deserialized from the YAML file, it will pull two root-level fields from the YAML object: `template`, the template for our Hello World saying, and `defaultName`, the default name to use. Both `template` and `defaultName` are annotated with `@NotEmpty`, so if the YAML configuration file has blank values for either or is missing `template` entirely an informative exception will be thrown, and your application won't start.

Both the getters and setters for `template` and `defaultName` are annotated with `@JsonProperty`, which allows Jackson to both deserialize the properties from a YAML file but also to serialize it.

Note: The mapping from YAML to your application's `Configuration` instance is done by Jackson. This means your `Configuration` class can use all of Jackson's object-mapping annotations. The validation of `@NotEmpty` is handled by Hibernate Validator, which has a wide range of built-in constraints for you to use.

Our YAML file will then look like the below, full example yml here:

```
template: Hello, %s!
defaultName: Stranger
```

Dropwizard has *many* more configuration parameters than that, but they all have sane defaults so you can keep your configuration files small and focused.

So save that YAML file in the directory you plan to run the fat jar from (see below) as `hello-world.yml`, because we'll be getting up and running pretty soon, and we'll need it. Next up, we're creating our application class!

1.4 Creating An Application Class

Combined with your project's `Configuration` subclass, its `Application` subclass forms the core of your Dropwizard application. The `Application` class pulls together the various bundles and commands which provide basic functionality. (More on that later.) For now, though, our `HelloWorldApplication` looks like this:

```
package com.example.helloworld;

import io.dropwizard.Application;
import io.dropwizard.setup.Bootstrap;
import io.dropwizard.setup.Environment;
import com.example.helloworld.resources.HelloWorldResource;
import com.example.helloworld.health.TemplateHealthCheck;

public class HelloWorldApplication extends Application<HelloWorldConfiguration> {
    public static void main(String[] args) throws Exception {
        new HelloWorldApplication().run(args);
```

(continues on next page)

(continued from previous page)

```
    }

    @Override
    public String getName() {
        return "hello-world";
    }

    @Override
    public void initialize(Bootstrap<HelloWorldConfiguration> bootstrap) {
        // nothing to do yet
    }

    @Override
    public void run(HelloWorldConfiguration configuration,
                    Environment environment) {
        // nothing to do yet
    }

}
```

As you can see, `HelloWorldApplication` is parameterized with the application's configuration type, `HelloWorldConfiguration`. An `initialize` method is used to configure aspects of the application required before the application is run, like bundles, configuration source providers, etc. Also, we've added a `static main` method, which will be our application's entry point. Right now, we don't have any functionality implemented, so our `run` method is a little boring. Let's fix that!

1.5 Creating A Representation Class

Before we can get into the nuts-and-bolts of our Hello World application, we need to stop and think about our API. Luckily, our application needs to conform to an industry standard, RFC 1149, which specifies the following JSON representation of a Hello World saying:

```
{
  "id": 1,
  "content": "Hi!"
}
```

The `id` field is a unique identifier for the saying, and `content` is the textual representation of the saying. (Thankfully, this is a fairly straight-forward industry standard.)

To model this representation, we'll create a representation class:

```
package com.example.helloworld.api;

import com.fasterxml.jackson.annotation.JsonProperty;
import org.hibernate.validator.constraints.Length;

public class Saying {
    private long id;

    @Length(max = 3)
    private String content;

    public Saying() {
```

(continues on next page)

(continued from previous page)

```
        // Jackson deserialization
    }

    public Saying(long id, String content) {
        this.id = id;
        this.content = content;
    }

    @JsonProperty
    public long getId() {
        return id;
    }

    @JsonProperty
    public String getContent() {
        return content;
    }
}
```

This is a pretty simple POJO, but there are a few things worth noting here.

First, it's immutable. This makes `Saying` instances *very* easy to reason about in multi-threaded environments as well as single-threaded environments. Second, it uses the JavaBeans standard for the `id` and `content` properties. This allows Jackson to serialize it to the JSON we need. The Jackson object mapping code will populate the `id` field of the JSON object with the return value of `#getId()`, likewise with `content` and `#getContent()`. Lastly, the bean leverages validation to ensure the content size is no greater than 3.

Note: The JSON serialization here is done by Jackson, which supports far more than simple JavaBean objects like this one. In addition to the sophisticated set of annotations, you can even write your custom serializers and deserializers.

Now that we've got our representation class, it makes sense to start in on the resource it represents.

1.6 Creating A Resource Class

Jersey resources are the meat-and-potatoes of a Dropwizard application. Each resource class is associated with a URI template. For our application, we need a resource which returns new `Saying` instances from the URI `/hello-world`, so our resource class looks like this:

```
package com.example.helloworld.resources;

import com.example.helloworld.api.Saying;
import com.codahale.metrics.annotation.Timed;

import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.Produces;
import javax.ws.rs.QueryParam;
import javax.ws.rs.core.MediaType;
import java.util.concurrent.atomic.AtomicLong;
import java.util.Optional;

@Path("/hello-world")
@Produces(MediaType.APPLICATION_JSON)
```

(continues on next page)

(continued from previous page)

```
public class HelloWorldResource {
    private final String template;
    private final String defaultName;
    private final AtomicLong counter;

    public HelloWorldResource(String template, String defaultName) {
        this.template = template;
        this.defaultName = defaultName;
        this.counter = new AtomicLong();
    }

    @GET
    @Timed
    public Saying sayHello(@QueryParam("name") Optional<String> name) {
        final String value = String.format(template, name.orElse(defaultName));
        return new Saying(counter.incrementAndGet(), value);
    }
}
```

Finally, we're in the thick of it! Let's start from the top and work our way down.

`HelloWorldResource` has two annotations: `@Path` and `@Produces`. `@Path("/hello-world")` tells Jersey that this resource is accessible at the URI `/hello-world`, and `@Produces(MediaType.APPLICATION_JSON)` lets Jersey's content negotiation code know that this resource produces representations which are `application/json`.

`HelloWorldResource` takes two parameters for construction: the `template` it uses to produce the saying and the `defaultName` used when the user declines to tell us their name. An `AtomicLong` provides us with a cheap, thread-safe way of generating unique(ish) IDs.

> **Warning:** Resource classes are used by multiple threads concurrently. In general, we recommend that resources be stateless/immutable, but it's important to keep the context in mind.

`#sayHello(Optional<String>)` is the meat of this class, and it's a fairly simple method. The `@QueryParam("name")` annotation tells Jersey to map the `name` parameter from the query string to the `name` parameter in the method. If the client sends a request to `/hello-world?name=Dougie`, `sayHello` will be called with `Optional.of("Dougie")`; if there is no `name` parameter in the query string, `sayHello` will be called with `Optional.absent()`. (Support for Guava's `Optional` is a little extra sauce that Dropwizard adds to Jersey's existing functionality.)

Note: If the client sends a request to `/hello-world?name=`, `sayHello` will be called with `Optional.of("")`. This may seem odd at first, but this follows the standards (an application may have different behavior depending on if a parameter is empty vs nonexistent). You can swap `Optional<String>` parameter with `NonEmptyStringParam` if you want `/hello-world?name=` to return "Hello, Stranger!" For more information on resource parameters see *the documentation*

Inside the `sayHello` method, we increment the counter, format the template using `String.format(String, Object...)`, and return a new `Saying` instance.

Because `sayHello` is annotated with `@Timed`, Dropwizard automatically records the duration and rate of its invocations as a Metrics Timer.

Once `sayHello` has returned, Jersey takes the `Saying` instance and looks for a provider class which can write `Saying` instances as `application/json`. Dropwizard has one such provider built in which allows for producing

and consuming Java objects as JSON objects. The provider writes out the JSON and the client receives a `200 OK` response with a content type of `application/json`.

1.6.1 Registering A Resource

Before that will actually work, though, we need to go back to `HelloWorldApplication` and add this new resource class. In its `run` method we can read the template and default name from the `HelloWorldConfiguration` instance, create a new `HelloWorldResource` instance, and then add it to the application's Jersey environment:

```
@Override
public void run(HelloWorldConfiguration configuration,
                Environment environment) {
    final HelloWorldResource resource = new HelloWorldResource(
        configuration.getTemplate(),
        configuration.getDefaultName()
    );
    environment.jersey().register(resource);
}
```

When our application starts, we create a new instance of our resource class with the parameters from the configuration file and hand it off to the `Environment`, which acts like a registry of all the things your application can do.

Note: A Dropwizard application can contain *many* resource classes, each corresponding to its own URI pattern. Just add another `@Path`-annotated resource class and call `register` with an instance of the new class.

Before we go too far, we should add a health check for our application.

1.7 Creating A Health Check

Health checks give you a way of adding small tests to your application to allow you to verify that your application is functioning correctly in production. We **strongly** recommend that all of your applications have at least a minimal set of health checks.

Note: We recommend this so strongly, in fact, that Dropwizard will nag you should you neglect to add a health check to your project.

Since formatting strings is not likely to fail while an application is running (unlike, say, a database connection pool), we'll have to get a little creative here. We'll add a health check to make sure we can actually format the provided template:

```
package com.example.helloworld.health;

import com.codahale.metrics.health.HealthCheck;

public class TemplateHealthCheck extends HealthCheck {
    private final String template;

    public TemplateHealthCheck(String template) {
        this.template = template;
    }

```

(continues on next page)

(continued from previous page)

```
    @Override
    protected Result check() throws Exception {
        final String saying = String.format(template, "TEST");
        if (!saying.contains("TEST")) {
            return Result.unhealthy("template doesn't include a name");
        }
        return Result.healthy();
    }
}
```

`TemplateHealthCheck` checks for two things: that the provided template is actually a well-formed format string, and that the template actually produces output with the given name.

If the string is not a well-formed format string (for example, someone accidentally put `Hello, %s%` in the configuration file), then `String.format(String, Object...)` will throw an `IllegalFormatException` and the health check will implicitly fail. If the rendered saying doesn't include the test string, the health check will explicitly fail by returning an unhealthy `Result`.

1.7.1 Adding A Health Check

As with most things in Dropwizard, we create a new instance with the appropriate parameters and add it to the `Environment`:

```
@Override
public void run(HelloWorldConfiguration configuration,
                Environment environment) {
    final HelloWorldResource resource = new HelloWorldResource(
        configuration.getTemplate(),
        configuration.getDefaultName()
    );
    final TemplateHealthCheck healthCheck =
        new TemplateHealthCheck(configuration.getTemplate());
    environment.healthChecks().register("template", healthCheck);
    environment.jersey().register(resource);
}
```

Now we're almost ready to go!

1.8 Building Fat JARs

We recommend that you build your Dropwizard applications as "fat" JAR files — single `.jar` files which contain *all* of the `.class` files required to run your application. This allows you to build a single deployable artifact which you can promote from your staging environment to your QA environment to your production environment without worrying about differences in installed libraries. To start building our Hello World application as a fat JAR, we need to configure a Maven plugin called `maven-shade`. In the `<build><plugins>` section of your `pom.xml` file, add this:

```
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-shade-plugin</artifactId>
    <version>2.3</version>
    <configuration>
```

(continues on next page)

(continued from previous page)

```
            <createDependencyReducedPom>true</createDependencyReducedPom>
            <filters>
                <filter>
                    <artifact>*:*</artifact>
                    <excludes>
                        <exclude>META-INF/*.SF</exclude>
                        <exclude>META-INF/*.DSA</exclude>
                        <exclude>META-INF/*.RSA</exclude>
                    </excludes>
                </filter>
            </filters>
        </configuration>
        <executions>
            <execution>
                <phase>package</phase>
                <goals>
                    <goal>shade</goal>
                </goals>
                <configuration>
                    <transformers>
                        <transformer implementation="org.apache.maven.plugins.shade.
↪resource.ServicesResourceTransformer"/>
                        <transformer implementation="org.apache.maven.plugins.shade.
↪resource.ManifestResourceTransformer">
                            <mainClass>com.example.helloworld.HelloWorldApplication</
↪mainClass>
                        </transformer>
                    </transformers>
                </configuration>
            </execution>
        </executions>
</plugin>
```

This configures Maven to do a couple of things during its `package` phase:

- Produce a `pom.xml` file which doesn't include dependencies for the libraries whose contents are included in the fat JAR.
- Exclude all digital signatures from signed JARs. If you don't, then Java considers the signature invalid and won't load or run your JAR file.
- Collate the various `META-INF/services` entries in the JARs instead of overwriting them. (Neither Dropwizard nor Jersey works without those.)
- Set `com.example.helloworld.HelloWorldApplication` as the JAR's `MainClass`. This will allow you to run the JAR using `java -jar`.

Warning: If your application has a dependency which *must* be signed (e.g., a JCA/JCE provider or other trusted library), you have to add an exclusion to the `maven-shade-plugin` configuration for that library and include that JAR in the classpath.

Warning: Since Dropwizard is using the Java ServiceLoader functionality to register and load extensions, the minimizeJar option of the *maven-shade-plugin* will lead to non-working application JARs.

1.8.1 Versioning Your JARs

Dropwizard can also use the project version if it's embedded in the JAR's manifest as the `Implementation-Version`. To embed this information using Maven, add the following to the `<build><plugins>` section of your `pom.xml` file:

```
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-jar-plugin</artifactId>
    <version>2.4</version>
    <configuration>
        <archive>
            <manifest>
                <addDefaultImplementationEntries>true</
↪addDefaultImplementationEntries>
            </manifest>
        </archive>
    </configuration>
</plugin>
```

This can be handy when trying to figure out what version of your application you have deployed on a machine.

Once you've got that configured, go into your project directory and run `mvn package` (or run the `package` goal from your IDE). You should see something like this:

```
[INFO] Including org.eclipse.jetty:jetty-util:jar:7.6.0.RC0 in the shaded jar.
[INFO] Including com.google.guava:guava:jar:10.0.1 in the shaded jar.
[INFO] Including com.google.code.findbugs:jsr305:jar:1.3.9 in the shaded jar.
[INFO] Including org.hibernate:hibernate-validator:jar:4.2.0.Final in the shaded jar.
[INFO] Including javax.validation:validation-api:jar:1.0.0.GA in the shaded jar.
[INFO] Including org.yaml:snakeyaml:jar:1.9 in the shaded jar.
[INFO] Replacing original artifact with shaded artifact.
[INFO] Replacing /Users/yourname/Projects/hello-world/target/hello-world-0.0.1-
↪SNAPSHOT.jar with /Users/yourname/Projects/hello-world/target/hello-world-0.0.1-
↪SNAPSHOT-shaded.jar
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 8.415s
[INFO] Finished at: Fri Dec 02 16:26:42 PST 2011
[INFO] Final Memory: 11M/81M
[INFO] ------------------------------------------------------------------------
```

Congratulations! You've built your first Dropwizard project! Now it's time to run it!

1.9 Running Your Application

Now that you've built a JAR file, it's time to run it.

In your project directory, run this:

```
java -jar target/hello-world-0.0.1-SNAPSHOT.jar
```

You should see something like the following:

```
usage: java -jar hello-world-0.0.1-SNAPSHOT.jar
       [-h] [-v] {server} ...

positional arguments:
  {server}              available commands

optional arguments:
  -h, --help            show this help message and exit
  -v, --version         show the service version and exit
```

Dropwizard takes the first command line argument and dispatches it to a matching command. In this case, the only command available is `server`, which runs your application as an HTTP server. The `server` command requires a configuration file, so let's go ahead and give it *the YAML file we previously saved*:

```
java -jar target/hello-world-0.0.1-SNAPSHOT.jar server hello-world.yml
```

You should see something like the following:

```
INFO  [2011-12-03 00:38:32,927] io.dropwizard.cli.ServerCommand: Starting hello-world
INFO  [2011-12-03 00:38:32,931] org.eclipse.jetty.server.Server: jetty-7.x.y-SNAPSHOT
INFO  [2011-12-03 00:38:32,936] org.eclipse.jetty.server.handler.ContextHandler:␣
↪started o.e.j.s.ServletContextHandler{/,null}
INFO  [2011-12-03 00:38:32,999] com.sun.jersey.server.impl.application.
↪WebApplicationImpl: Initiating Jersey application, version 'Jersey: 1.10 11/02/2011␣
↪03:53 PM'
INFO  [2011-12-03 00:38:33,041] io.dropwizard.setup.Environment:

    GET     /hello-world (com.example.helloworld.resources.HelloWorldResource)

INFO  [2011-12-03 00:38:33,215] org.eclipse.jetty.server.handler.ContextHandler:␣
↪started o.e.j.s.ServletContextHandler{/,null}
INFO  [2011-12-03 00:38:33,235] org.eclipse.jetty.server.AbstractConnector: Started␣
↪BlockingChannelConnector@0.0.0.0:8080 STARTING
INFO  [2011-12-03 00:38:33,238] org.eclipse.jetty.server.AbstractConnector: Started␣
↪SocketConnector@0.0.0.0:8081 STARTING
```

Your Dropwizard application is now listening on ports `8080` for application requests and `8081` for administration requests. If you press `^C`, the application will shut down gracefully, first closing the server socket, then waiting for in-flight requests to be processed, then shutting down the process itself.

However, while it's up, let's give it a whirl! Click here to say hello! Click here to get even friendlier!

So, we're generating sayings. Awesome. But that's not all your application can do. One of the main reasons for using Dropwizard is the out-of-the-box operational tools it provides, all of which can be found on the admin port.

If you click through to the metrics resource, you can see all of your application's metrics represented as a JSON object.

The threads resource allows you to quickly get a thread dump of all the threads running in that process.

Hint: When a Jetty worker thread is handling an incoming HTTP request, the thread name is set to the method and URI of the request. This can be *very* helpful when debugging a poorly-behaving request.

The healthcheck resource runs the *health check class we wrote*. You should see something like this:

```
* deadlocks: OK
* template: OK
```

`template` here is the result of your `TemplateHealthCheck`, which unsurprisingly passed. `deadlocks` is a built-in health check which looks for deadlocked JVM threads and prints out a listing if any are found.

1.10 Next Steps

Well, congratulations. You've got a Hello World application ready for production (except for the lack of tests) that's capable of doing 30,000-50,000 requests per second. Hopefully, you've gotten a feel for how Dropwizard combines Jetty, Jersey, Jackson, and other stable, mature libraries to provide a phenomenal platform for developing RESTful web applications.

There's a lot more to Dropwizard than is covered here (commands, bundles, servlets, advanced configuration, validation, HTTP clients, database clients, views, etc.), all of which is covered by the *User Manual*.

CHAPTER 2

User Manual

This goal of this document is to provide you with all the information required to build, organize, test, deploy, and maintain Dropwizard-based applications. If you're new to Dropwizard, you should read the Getting Started guide first.

2.1 Dropwizard Core

The `dropwizard-core` module provides you with everything you'll need for most of your applications.

It includes:

- Jetty, a high-performance HTTP server.
- Jersey, a full-featured RESTful web framework.
- Jackson, the best JSON library for the JVM.
- Metrics, an excellent library for application metrics.
- Guava, Google's excellent utility library.
- Logback, the successor to Log4j, Java's most widely-used logging framework.
- Hibernate Validator, the reference implementation of the Java Bean Validation standard.

Dropwizard consists mostly of glue code to automatically connect and configure these components.

2.1.1 Organizing Your Project

If you plan on developing a client library for other developers to access your service, we recommend you separate your projects into three Maven modules: `project-api`, `project-client`, and `project-application`.

`project-api` should contain your *Representations*; `project-client` should use those classes and an *HTTP client* to implement a full-fledged client for your application, and `project-application` should provide the actual application implementation, including *Resources*.

To give a concrete example of this project structure, let's say we wanted to create a Stripe-like API where clients can issue charges and the server would echo the charge back to the client. `stripe-api` project would hold our `Charge` object as both the server and client want to work with the charge and to promote code reuse, `Charge` objects are stored in a shared module. `stripe-app` is the Dropwizard application. `stripe-client` abstracts away the raw HTTP interactions and deserialization logic. Instead of using a HTTP client, users of `stripe-client` would just pass in a `Charge` object to a function and behind the scenes, `stripe-client` will call the HTTP endpoint. The client library may also take care of connection pooling, and may provide a more friendly way of interpreting error messages. Basically, distributing a client library for your app will help other developers integrate more quickly with the service.

If you are not planning on distributing a client library for developers, one can combine `project-api` and `project-application` into a single project, which tends to look like this:

- `com.example.myapplication`:
  - `api`: *Representations*. Request and response bodies.
  - `cli`: *Commands*
  - `client`: *Client* code that accesses external HTTP services.
  - `core`: Domain implementation; where objects not used in the API such as POJOs, validations, crypto, etc, reside.
  - `jdbi`: *Database* access classes
  - `health`: *Health Checks*
  - `resources`: *Resources*
  - `MyApplication`: The *application* class
  - `MyApplicationConfiguration`: *configuration* class

2.1.2 Application

The main entry point into a Dropwizard application is, unsurprisingly, the `Application` class. Each `Application` has a **name**, which is mostly used to render the command-line interface. In the constructor of your `Application` you can add *Bundles* and *Commands* to your application.

2.1.3 Configuration

Dropwizard provides a number of built-in configuration parameters. They are well documented in the example project's configuration and *configuration refererence*.

Each `Application` subclass has a single type parameter: that of its matching `Configuration` subclass. These are usually at the root of your application's main package. For example, your User application would have two classes: `UserApplicationConfiguration`, extending `Configuration`, and `UserApplication`, extending `Application<UserApplicationConfiguration>`.

When your application runs *Configured Commands* like the `server` command, Dropwizard parses the provided YAML configuration file and builds an instance of your application's configuration class by mapping YAML field names to object field names.

Note: If your configuration file doesn't end in `.yml` or `.yaml`, Dropwizard tries to parse it as a JSON file.

To keep your configuration file and class manageable, we recommend grouping related configuration parameters into independent configuration classes. If your application requires a set of configuration parameters in order to connect to a message queue, for example, we recommend that you create a new `MessageQueueFactory` class:

```
public class MessageQueueFactory {
    @NotEmpty
    private String host;

    @Min(1)
    @Max(65535)
    private int port = 5672;

    @JsonProperty
    public String getHost() {
        return host;
    }

    @JsonProperty
    public void setHost(String host) {
        this.host = host;
    }

    @JsonProperty
    public int getPort() {
        return port;
    }

    @JsonProperty
    public void setPort(int port) {
        this.port = port;
    }

    public MessageQueueClient build(Environment environment) {
        MessageQueueClient client = new MessageQueueClient(getHost(), getPort());
        environment.lifecycle().manage(new Managed() {
            @Override
            public void start() {
            }

            @Override
            public void stop() {
                client.close();
            }
        });
        return client;
    }
}
```

In this example our factory will automatically tie our `MessageQueueClient` connection to the lifecycle of our application's `Environment`.

Your main `Configuration` subclass can then include this as a member field:

```
public class ExampleConfiguration extends Configuration {
    @Valid
    @NotNull
    private MessageQueueFactory messageQueue = new MessageQueueFactory();
```

(continues on next page)

(continued from previous page)

```
    @JsonProperty("messageQueue")
    public MessageQueueFactory getMessageQueueFactory() {
        return messageQueue;
    }

    @JsonProperty("messageQueue")
    public void setMessageQueueFactory(MessageQueueFactory factory) {
        this.messageQueue = factory;
    }
}
```

And your `Application` subclass can then use your factory to directly construct a client for the message queue:

```
public void run(ExampleConfiguration configuration,
                Environment environment) {
    MessageQueueClient messageQueue = configuration.getMessageQueueFactory().
↪build(environment);
}
```

Then, in your application's YAML file, you can use a nested `messageQueue` field:

```
messageQueue:
  host: mq.example.com
  port: 5673
```

The `@NotNull`, `@NotEmpty`, `@Min`, `@Max`, and `@Valid` annotations are part of *Dropwizard Validation* functionality. If your YAML configuration file's `messageQueue.host` field was missing (or was a blank string), Dropwizard would refuse to start and would output an error message describing the issues.

Once your application has parsed the YAML file and constructed its `Configuration` instance, Dropwizard then calls your `Application` subclass to initialize your application's `Environment`.

Note: You can override configuration settings by passing special Java system properties when starting your application. Overrides must start with prefix `dw.`, followed by the path to the configuration value being overridden.

For example, to override the Logging level, you could start your application like this:

```
java -Ddw.logging.level=DEBUG server my-config.json
```

This will work even if the configuration setting in question does not exist in your config file, in which case it will get added.

You can override configuration settings in arrays of objects like this:

```
java -Ddw.server.applicationConnectors[0].port=9090 server my-config.json
```

You can override configuration settings in maps like this:

```
java -Ddw.database.properties.hibernate.hbm2ddl.auto=none server my-config.
json
```

You can also override a configuration setting that is an array of strings by using the ',' character as an array element separator. For example, to override a configuration setting myapp.myserver.hosts that is an array of strings in the configuration, you could start your service like this: `java -Ddw.myapp.myserver.hosts=server1,server2,server3 server my-config.json`

If you need to use the ',' character in one of the values, you can escape it by using '\,' instead.

The array override facility only handles configuration elements that are arrays of simple strings. Also, the setting in question must already exist in your configuration file as an array; this mechanism will not work if the configuration

key being overridden does not exist in your configuration file. If it does not exist or is not an array setting, it will get added as a simple string setting, including the ',' characters as part of the string.

Environment variables

The `dropwizard-configuration` module also provides the capabilities to substitute configuration settings with the value of environment variables using a `SubstitutingSourceProvider` and `EnvironmentVariableSubstitutor`.

```
public class MyApplication extends Application<MyConfiguration> {
    // [...]
    @Override
    public void initialize(Bootstrap<MyConfiguration> bootstrap) {
        // Enable variable substitution with environment variables
        bootstrap.setConfigurationSourceProvider(
                new SubstitutingSourceProvider(bootstrap.
↪getConfigurationSourceProvider(),
                                                new␣
↪EnvironmentVariableSubstitutor(false)
                )
        );

    }

    // [...]
}
```

The configuration settings which should be substituted need to be explicitly written in the configuration file and follow the substitution rules of StrSubstitutor from the Apache Commons Lang library.

```
mySetting: ${DW_MY_SETTING}
defaultSetting: ${DW_DEFAULT_SETTING:-default value}
```

In general `SubstitutingSourceProvider` isn't restricted to substitute environment variables but can be used to replace variables in the configuration source with arbitrary values by passing a custom `StrSubstitutor` implementation.

SSL

SSL support is built into Dropwizard. You will need to provide your own java keystore, which is outside the scope of this document (`keytool` is the command you need, and Jetty's documentation can get you started). There is a test keystore you can use in the Dropwizard example project.

```
server:
  applicationConnectors:
    - type: https
      port: 8443
      keyStorePath: example.keystore
      keyStorePassword: example
      validateCerts: false
```

By default, only secure TLSv1.2 cipher suites are allowed. Older versions of cURL, Java 6 and 7, and other clients may be unable to communicate with the allowed cipher suites, but this was a conscious decision that sacrifices interoperability for security.

Dropwizard allows a workaround by specifying a customized list of cipher suites. If no lists of supported protocols or cipher suites are specified, then the JVM defaults are used. If no lists of excluded protocols or cipher suites are specified, then the defaults are inherited from Jetty.

The following list of excluded cipher suites will allow for TLSv1 and TLSv1.1 clients to negotiate a connection similar to pre-Dropwizard 1.0.

```
server:
  applicationConnectors:
    - type: https
      port: 8443
      excludedCipherSuites:
        - SSL_RSA_WITH_DES_CBC_SHA
        - SSL_DHE_RSA_WITH_DES_CBC_SHA
        - SSL_DHE_DSS_WITH_DES_CBC_SHA
        - SSL_RSA_EXPORT_WITH_RC4_40_MD5
        - SSL_RSA_EXPORT_WITH_DES40_CBC_SHA
        - SSL_DHE_RSA_EXPORT_WITH_DES40_CBC_SHA
        - SSL_DHE_DSS_EXPORT_WITH_DES40_CBC_SHA
```

Since the version 9.4.8 (Dropwizard 1.2.3) Jetty supports native SSL via Google's Conscrypt which uses BoringSSL (Google's fork of OpenSSL) for handling cryptography. You can enable it in Dropwizard by registering the provider in your app:

```
<dependency>
    <groupId>org.conscrypt</groupId>
    <artifactId>conscrypt-openjdk-uber</artifactId>
    <version>${conscrypt.version}</version>
</dependency>
```

```
static {
    Security.addProvider(new OpenSSLProvider());
}
```

and setting the JCE provider in the configuration:

```
server:
  type: simple
  connector:
    type: https
    jceProvider: Conscrypt
```

For HTTP/2 servers you need to add an ALPN Conscrypt provider as a dependency.

```
<dependency>
    <groupId>org.eclipse.jetty</groupId>
    <artifactId>jetty-alpn-conscrypt-server</artifactId>
    <version>${jetty.version}</version>
    <scope>test</scope>
</dependency>
```

2.1.4 Bootstrapping

Before a Dropwizard application can provide the command-line interface, parse a configuration file, or run as a server, it must first go through a bootstrapping phase. This phase corresponds to your `Application` subclass's

`initialize` method. You can add *Bundles*, *Commands*, or register Jackson modules to allow you to include custom types as part of your configuration class.

2.1.5 Environments

A Dropwizard `Environment` consists of all the *Resources*, servlets, filters, *Health Checks*, Jersey providers, *Managed Objects*, *Tasks*, and Jersey properties which your application provides.

Each `Application` subclass implements a `run` method. This is where you should be creating new resource instances, etc., and adding them to the given `Environment` class:

```
@Override
public void run(ExampleConfiguration config,
                Environment environment) {
    // encapsulate complicated setup logic in factories
    final Thingy thingy = config.getThingyFactory().build();

    environment.jersey().register(new ThingyResource(thingy));
    environment.healthChecks().register("thingy", new ThingyHealthCheck(thingy));
}
```

It's important to keep the `run` method clean, so if creating an instance of something is complicated, like the `Thingy` class above, extract that logic into a factory.

2.1.6 Health Checks

A health check is a runtime test which you can use to verify your application's behavior in its production environment. For example, you may want to ensure that your database client is connected to the database:

```
public class DatabaseHealthCheck extends HealthCheck {
    private final Database database;

    public DatabaseHealthCheck(Database database) {
        this.database = database;
    }

    @Override
    protected Result check() throws Exception {
        if (database.isConnected()) {
            return Result.healthy();
        } else {
            return Result.unhealthy("Cannot connect to " + database.getUrl());
        }
    }
}
```

You can then add this health check to your application's environment:

```
environment.healthChecks().register("database", new DatabaseHealthCheck(database));
```

By sending a `GET` request to `/healthcheck` on the admin port you can run these tests and view the results:

```
$ curl http://dw.example.com:8081/healthcheck
{"deadlocks":{"healthy":true},"database":{"healthy":true}}
```

If all health checks report success, a `200 OK` is returned. If any fail, a `500 Internal Server Error` is returned with the error messages and exception stack traces (if an exception was thrown).

All Dropwizard applications ship with the `deadlocks` health check installed by default, which uses Java 1.6's built-in thread deadlock detection to determine if any threads are deadlocked.

2.1.7 Managed Objects

Most applications involve objects which need to be started and stopped: thread pools, database connections, etc. Dropwizard provides the `Managed` interface for this. You can either have the class in question implement the `#start()` and `#stop()` methods, or write a wrapper class which does so. Adding a `Managed` instance to your application's `Environment` ties that object's lifecycle to that of the application's HTTP server. Before the server starts, the `#start()` method is called. After the server has stopped (and after its graceful shutdown period) the `#stop()` method is called.

For example, given a theoretical Riak client which needs to be started and stopped:

```
public class RiakClientManager implements Managed {
    private final RiakClient client;

    public RiakClientManager(RiakClient client) {
        this.client = client;
    }

    @Override
    public void start() throws Exception {
        client.start();
    }

    @Override
    public void stop() throws Exception {
        client.stop();
    }
}
```

```
public class MyApplication extends Application<MyConfiguration>{
    @Override
    public void run(MyApplicationConfiguration configuration, Environment␣
↪environment) {
        RiakClient client = ...;
        RiakClientManager riakClientManager = new RiakClientManager(client);
        environment.lifecycle().manage(riakClientManager);
    }
}
```

If `RiakClientManager#start()` throws an exception–e.g., an error connecting to the server–your application will not start and a full exception will be logged. If `RiakClientManager#stop()` throws an exception, the exception will be logged but your application will still be able to shut down.

It should be noted that `Environment` has built-in factory methods for `ExecutorService` and `ScheduledExecutorService` instances which are managed. See `LifecycleEnvironment#executorService` and `LifecycleEnvironment#scheduledExecutorService` for details.

2.1.8 Bundles

A Dropwizard bundle is a reusable group of functionality, used to define blocks of an application's behavior. For example, `AssetBundle` from the `dropwizard-assets` module provides a simple way to serve static assets from your application's `src/main/resources/assets` directory as files available from `/assets/*` (or any other path) in your application.

Configured Bundles

Some bundles require configuration parameters. These bundles implement `ConfiguredBundle` and will require your application's `Configuration` subclass to implement a specific interface.

For example: given the configured bundle `MyConfiguredBundle` and the interface `MyConfiguredBundleConfig` below. Your application's `Configuration` subclass would need to implement `MyConfiguredBundleConfig`.

```
public class MyConfiguredBundle implements ConfiguredBundle<MyConfiguredBundleConfig>{

    @Override
    public void run(MyConfiguredBundleConfig applicationConfig, Environment␣
↪environment) {
        applicationConfig.getBundleSpecificConfig();
    }

    @Override
    public void initialize(Bootstrap<?> bootstrap) {

    }
}

public interface MyConfiguredBundleConfig{

    String getBundleSpecificConfig();

}
```

Serving Assets

Either your application or your static assets can be served from the root path, but not both. The latter is useful when using Dropwizard to back a Javascript application. To enable it, move your application to a sub-URL.

```
server:
  rootPath: /api/
```

Note: If you use the *Simple* server configuration, then `rootPath` is calculated relatively from `applicationContextPath`. So, your API will be accessible from the path `/application/api/`

Then use an extended `AssetsBundle` constructor to serve resources in the `assets` folder from the root path. `index.htm` is served as the default page.

```
@Override
public void initialize(Bootstrap<HelloWorldConfiguration> bootstrap) {
```

(continues on next page)

(continued from previous page)

```
    bootstrap.addBundle(new AssetsBundle("/assets/", "/"));
}
```

When an `AssetBundle` is added to the application, it is registered as a servlet using a default name of `assets`. If the application needs to have multiple `AssetBundle` instances, the extended constructor should be used to specify a unique name for the `AssetBundle`.

```
@Override
public void initialize(Bootstrap<HelloWorldConfiguration> bootstrap) {
    bootstrap.addBundle(new AssetsBundle("/assets/css", "/css", null, "css"));
    bootstrap.addBundle(new AssetsBundle("/assets/js", "/js", null, "js"));
    bootstrap.addBundle(new AssetsBundle("/assets/fonts", "/fonts", null, "fonts"));
}
```

SSL Reload

By registering the `SslReloadBundle` your application can have new certificate information reloaded at runtime, so a restart is not necessary.

```
@Override
public void initialize(Bootstrap<HelloWorldConfiguration> bootstrap) {
    bootstrap.addBundle(new SslReloadBundle());
}
```

To trigger a reload send a `POST` request to `ssl-reload`

```
curl -k -X POST 'https://localhost:<admin-port>/tasks/ssl-reload'
```

Dropwizard will use the same exact https configuration (keystore location, password, etc) when performing the reload.

Note: If anything is wrong with the new certificate (eg. wrong password in keystore), no new certificates are loaded. So if the application and admin ports use different certificates and one of them is invalid, then none of them are reloaded.

A http 500 error is returned on reload failure, so make sure to trap for this error with whatever tool is used to trigger a certificate reload, and alert the appropriate admin. If the situation is not remedied, next time the app is stopped, it will be unable to start!

2.1.9 Commands

Commands are basic actions which Dropwizard runs based on the arguments provided on the command line. The built-in `server` command, for example, spins up an HTTP server and runs your application. Each `Command` subclass has a name and a set of command line options which Dropwizard will use to parse the given command line arguments.

Below is an example on how to add a command and have Dropwizard recognize it.

```
public class MyCommand extends Command {
    public MyCommand() {
        // The name of our command is "hello" and the description printed is
        // "Prints a greeting"
        super("hello", "Prints a greeting");
    }
```

(continues on next page)

(continued from previous page)

```
    @Override
    public void configure(Subparser subparser) {
        // Add a command line option
        subparser.addArgument("-u", "--user")
                .dest("user")
                .type(String.class)
                .required(true)
                .help("The user of the program");
    }

    @Override
    public void run(Bootstrap<?> bootstrap, Namespace namespace) throws Exception {
        System.out.println("Hello " + namespace.getString("user"));
    }
}
```

Dropwizard recognizes our command once we add it in the `initialize` stage of our application.

```
public class MyApplication extends Application<MyConfiguration>{
    @Override
    public void initialize(Bootstrap<DropwizardConfiguration> bootstrap) {
        bootstrap.addCommand(new MyCommand());
    }
}
```

To invoke the new functionality, run the following:

```
java -jar <jarfile> hello dropwizard
```

Configured Commands

Some commands require access to configuration parameters and should extend the `ConfiguredCommand` class, using your application's `Configuration` class as its type parameter. By default, Dropwizard will treat the last argument on the command line as the path to a YAML configuration file, parse and validate it, and provide your command with an instance of the configuration class.

A `ConfiguredCommand` can have additional command line options specified, while keeping the last argument the path to the YAML configuration.

```
@Override
public void configure(Subparser subparser) {
    super.configure(subparser);

    // Add a command line option
    subparser.addArgument("-u", "--user")
            .dest("user")
            .type(String.class)
            .required(true)
            .help("The user of the program");
}
```

For more advanced customization of the command line (for example, having the configuration file location specified by `-c`), adapt the ConfiguredCommand class as needed.

2.1.10 Tasks

A `Task` is a run-time action your application provides access to on the administrative port via HTTP. All Dropwizard applications start with: the `gc` task, which explicitly triggers the JVM's garbage collection (This is useful, for example, for running full garbage collections during off-peak times or while the given application is out of rotation.); and the `log-level` task, which configures the level of any number of loggers at runtime (akin to Logback's `JmxConfigurator`). The execute method of a `Task` can be annotated with `@Timed`, `@Metered`, and `@ExceptionMetered`. Dropwizard will automatically record runtime information about your tasks. Here's a basic task class:

```
public class TruncateDatabaseTask extends Task {
    private final Database database;

    public TruncateDatabaseTask(Database database) {
        super("truncate");
        this.database = database;
    }

    @Override
    public void execute(ImmutableMultimap<String, String> parameters, PrintWriter␣
↪output) throws Exception {
        this.database.truncate();
    }
}
```

You can then add this task to your application's environment:

```
environment.admin().addTask(new TruncateDatabaseTask(database));
```

Running a task can be done by sending a `POST` request to `/tasks/{task-name}` on the admin port. The task will receive any query parameters as arguments. For example:

```
$ curl -X POST http://dw.example.com:8081/tasks/gc
Running GC...
Done!
```

You can also extend `PostBodyTask` to create a task which uses the body of the post request. Here's an example:

```
public class EchoTask extends PostBodyTask {
    public EchoTask() {
        super("echo");
    }

    @Override
    public void execute(ImmutableMultimap<String, String> parameters, String postBody,
↪ PrintWriter output) throws Exception {
        output.write(postBody);
        output.flush();
    }
}
```

2.1.11 Logging

Dropwizard uses Logback for its logging backend. It provides an slf4j implementation, and even routes all `java.util.logging`, Log4j, and Apache Commons Logging usage through Logback.

slf4j provides the following logging levels:

ERROR Error events that might still allow the application to continue running.

WARN Potentially harmful situations.

INFO Informational messages that highlight the progress of the application at coarse-grained level.

DEBUG Fine-grained informational events that are most useful to debug an application.

TRACE Finer-grained informational events than the `DEBUG` level.

Note: If you don't want to use Logback, you can exclude it from Dropwizard and use an alternative logging configuration:

- Exclude Logback from the *dropwizard-core* artifact

```
<dependency>
    <groupId>io.dropwizard</groupId>
    <artifactId>dropwizard-core</artifactId>
    <version>{$dropwizard.version}</version>
    <exclusions>
        <exclusion>
            <groupId>ch.qos.logback</groupId>
            <artifactId>logback-classic</artifactId>
        </exclusion>
        <exclusion>
            <groupId>ch.qos.logback</groupId>
            <artifactId>logback-access</artifactId>
        </exclusion>
        <exclusion>
            <groupId>org.slf4j</groupId>
            <artifactId>log4j-over-slf4j</artifactId>
        </exclusion>
    </exclusions>
</dependency>
```

- Mark the logging configuration as external in your Dropwizard config

```
server:
  type: simple
  applicationContextPath: /application
  adminContextPath: /admin
  requestLog:
    type: external
logging:
  type: external
```

- Disable bootstrapping Logback in your application

```
public class ExampleApplication extends Application<ExampleConfiguration>
↪{

    @Override
    protected void bootstrapLogging() {
    }
}
```

Log Format

Dropwizard's log format has a few specific goals:

- Be human readable.
- Be machine parsable.
- Be easy for sleepy ops folks to figure out why things are pear-shaped at 3:30AM using standard UNIXy tools like `tail` and `grep`.

The logging output looks like this:

```
TRACE [2010-04-06 06:42:35,271] com.example.dw.Thing: Contemplating doing a thing.
DEBUG [2010-04-06 06:42:35,274] com.example.dw.Thing: About to do a thing.
INFO  [2010-04-06 06:42:35,274] com.example.dw.Thing: Doing a thing
WARN  [2010-04-06 06:42:35,275] com.example.dw.Thing: Doing a thing
ERROR [2010-04-06 06:42:35,275] com.example.dw.Thing: This may get ugly.
! java.lang.RuntimeException: oh noes!
! at com.example.dw.Thing.run(Thing.java:16)
!
```

A few items of note:

- All timestamps are in UTC and ISO 8601 format.
- You can grep for messages of a specific level really easily:

  ```
  tail -f dw.log | grep '^WARN'
  ```

- You can grep for messages from a specific class or package really easily:

  ```
  tail -f dw.log | grep 'com.example.dw.Thing'
  ```

- You can even pull out full exception stack traces, plus the accompanying log message:

  ```
  tail -f dw.log | grep -B 1 '^\!'
  ```

- The *!* prefix does *not* apply to syslog appenders, as stack traces are sent separately from the main message. Instead, *t* is used (this is the default value of the *SyslogAppender* that comes with Logback). This can be configured with the *stackTracePrefix* option when defining your appender.

Configuration

You can specify a default logger level, override the levels of other loggers in your YAML configuration file, and even specify appenders for them. The latter form of configuration is preferable, but the former is also acceptable.

```yaml
# Logging settings.
logging:

  # The default level of all loggers. Can be OFF, ERROR, WARN, INFO, DEBUG, TRACE, or␣
↪ALL.
  level: INFO

  # Logger-specific levels.
  loggers:

    # Overrides the level of com.example.dw.Thing and sets it to DEBUG.
```

(continues on next page)

(continued from previous page)

```
    "com.example.dw.Thing": DEBUG

    # Enables the SQL query log and redirect it to a separate file
    "org.hibernate.SQL":
      level: DEBUG
      # This line stops org.hibernate.SQL (or anything under it) from using the root␣
↪logger
      additive: false
      appenders:
        - type: file
          currentLogFilename: ./logs/example-sql.log
          archivedLogFilenamePattern: ./logs/example-sql-%d.log.gz
          archivedFileCount: 5
```

Console Logging

By default, Dropwizard applications log `INFO` and higher to `STDOUT`. You can configure this by editing the `logging` section of your YAML configuration file:

```
logging:
  appenders:
    - type: console
      threshold: WARN
      target: stderr
```

In the above, we're instead logging only `WARN` and `ERROR` messages to the `STDERR` device.

File Logging

Dropwizard can also log to an automatically rotated set of log files. This is the recommended configuration for your production environment:

```
logging:

  appenders:
    - type: file
      # The file to which current statements will be logged.
      currentLogFilename: ./logs/example.log

      # When the log file rotates, the archived log will be renamed to this and␣
↪gzipped. The
      # %d is replaced with the previous day (yyyy-MM-dd). Custom rolling windows can␣
↪be created
      # by passing a SimpleDateFormat-compatible format as an argument: "%d{yyyy-MM-
↪dd-hh}".
      archivedLogFilenamePattern: ./logs/example-%d.log.gz

      # The number of archived files to keep.
      archivedFileCount: 5

      # The timezone used to format dates. HINT: USE THE DEFAULT, UTC.
      timeZone: UTC
```

Syslog Logging

Finally, Dropwizard can also log statements to syslog.

Note: Because Java doesn't use the native syslog bindings, your syslog server **must** have an open network socket.

```yaml
logging:

  appenders:
    - type: syslog
      # The hostname of the syslog server to which statements will be sent.
      # N.B.: If this is the local host, the local syslog instance will need to be␣
↪configured to
      # listen on an inet socket, not just a Unix socket.
      host: localhost

      # The syslog facility to which statements will be sent.
      facility: local0
```

You can combine any number of different `appenders`, including multiple instances of the same appender with different configurations:

```yaml
logging:

  # Permit DEBUG, INFO, WARN and ERROR messages to be logged by appenders.
  level: DEBUG

  appenders:
    # Log warnings and errors to stderr
    - type: console
      threshold: WARN
      target: stderr

    # Log info, warnings and errors to our apps' main log.
    # Rolled over daily and retained for 5 days.
    - type: file
      threshold: INFO
      currentLogFilename: ./logs/example.log
      archivedLogFilenamePattern: ./logs/example-%d.log.gz
      archivedFileCount: 5

    # Log debug messages, info, warnings and errors to our apps' debug log.
    # Rolled over hourly and retained for 6 hours
    - type: file
      threshold: DEBUG
      currentLogFilename: ./logs/debug.log
      archivedLogFilenamePattern: ./logs/debug-%d{yyyy-MM-dd-hh}.log.gz
      archivedFileCount: 6
```

Logging Configuration via HTTP

Active log levels can be changed during the runtime of a Dropwizard application via HTTP using the `LogConfigurationTask`. For instance, to configure the log level for a single `Logger`:

```
curl -X POST -d "logger=com.example.helloworld&level=INFO" http://localhost:8081/
↪tasks/log-level
```

Logging Filters

Just because a statement has a level of `INFO`, doesn't mean it should be logged with other `INFO` statements. One can create logging filters that will intercept log statements before they are written and decide if they're allowed. Log filters can work on both regular statements and request log statements. The following example will be for request logging as there are many reasons why certain requests may be excluded from the log:

- Only log requests that have large bodies
- Only log requests that are slow
- Only log requests that resulted in a non-2xx status code
- Exclude requests that contain sensitive information in the URL
- Exclude healthcheck requests

The example will demonstrate excluding `/secret` requests from the log.

```
@JsonTypeName("secret-filter-factory")
public class SecretFilterFactory implements FilterFactory<IAccessEvent> {
    @Override
    public Filter<IAccessEvent> build() {
        return new Filter<IAccessEvent>() {
            @Override
            public FilterReply decide(IAccessEvent event) {
                if (event.getRequestURI().equals("/secret")) {
                    return FilterReply.DENY;
                } else {
                    return FilterReply.NEUTRAL;
                }
            }
        };
    }
}
```

Reference `SecretFilterFactory` type in our configuration.

```
server:
  requestLog:
    appenders:
      - type: console
        filterFactories:
          - type: secret-filter-factory
```

The last step is to add our class (in this case `com.example.SecretFilterFactory`) to `META-INF/services/io.dropwizard.logging.filter.FilterFactory` in our resources folder.

2.1.12 Testing Applications

All of Dropwizard's APIs are designed with testability in mind, so even your applications can have unit tests:

```java
public class MyApplicationTest {
    private final Environment environment = mock(Environment.class);
    private final JerseyEnvironment jersey = mock(JerseyEnvironment.class);
    private final MyApplication application = new MyApplication();
    private final MyConfiguration config = new MyConfiguration();

    @Before
    public void setup() throws Exception {
        config.setMyParam("yay");
        when(environment.jersey()).thenReturn(jersey);
    }

    @Test
    public void buildsAThingResource() throws Exception {
        application.run(config, environment);

        verify(jersey).register(isA(ThingResource.class));
    }
}
```

We highly recommend Mockito for all your mocking needs.

2.1.13 Banners

We think applications should print out a big ASCII art banner on startup. Yours should, too. It's fun. Just add a `banner.txt` class to `src/main/resources` and it'll print it out when your application starts:

```
INFO  [2011-12-09 21:56:37,209] io.dropwizard.cli.ServerCommand: Starting hello-world
                                                          dP
                                                          88
  .d8888b. dP.  .dP .d8888b. 88d8b.d8b. 88d888b. 88 .d8888b.
  88ooood8  `8bd8'  88'  `88 88'`88'`88 88'  `88 88 88ooood8
  88.  ...  .d88b.  88.  .88 88  88  88 88.  .88 88 88.  ...
  `88888P' dP'  `dP `88888P8 dP  dP  dP 88Y888P' dP `88888P'
                                        88
                                        dP

INFO  [2011-12-09 21:56:37,214] org.eclipse.jetty.server.Server: jetty-7.6.0
...
```

We could probably make up an argument about why this is a serious devops best practice with high ROI and an Agile Tool, but honestly we just enjoy this.

We recommend you use TAAG for all your ASCII art banner needs.

2.1.14 Resources

Unsurprisingly, most of your day-to-day work with a Dropwizard application will be in the resource classes, which model the resources exposed in your RESTful API. Dropwizard uses Jersey for this, so most of this section is just re-hashing or collecting various bits of Jersey documentation.

Jersey is a framework for mapping various aspects of incoming HTTP requests to POJOs and then mapping various aspects of POJOs to outgoing HTTP responses. Here's a basic resource class:

```
@Path("/{user}/notifications")
@Produces(MediaType.APPLICATION_JSON)
@Consumes(MediaType.APPLICATION_JSON)
public class NotificationsResource {
    private final NotificationStore store;

    public NotificationsResource(NotificationStore store) {
        this.store = store;
    }

    @GET
    public NotificationList fetch(@PathParam("user") LongParam userId,
                                  @QueryParam("count") @DefaultValue("20") IntParam␣
↪count) {
        final List<Notification> notifications = store.fetch(userId.get(), count.
↪get());
        if (notifications != null) {
            return new NotificationList(userId, notifications);
        }
        throw new WebApplicationException(Status.NOT_FOUND);
    }

    @POST
    public Response add(@PathParam("user") LongParam userId,
                        @NotNull @Valid Notification notification) {
        final long id = store.add(userId.get(), notification);
        return Response.created(UriBuilder.fromResource(NotificationResource.class)
                                          .build(userId.get(), id))
                       .build();
    }
}
```

This class provides a resource (a user's list of notifications) which responds to `GET` and `POST` requests to `/{user}/notifications`, providing and consuming `application/json` representations. There's quite a lot of functionality on display here, and this section will explain in detail what's in play and how to use these features in your application.

Paths

Important: Every resource class must have a `@Path` annotation.

The `@Path` annotation isn't just a static string, it's a URI Template. The `{user}` part denotes a named variable, and when the template matches a URI the value of that variable will be accessible via `@PathParam`-annotated method parameters.

For example, an incoming request for `/1001/notifications` would match the URI template, and the value `"1001"` would be available as the path parameter named `user`.

If your application doesn't have a resource class whose `@Path` URI template matches the URI of an incoming request, Jersey will automatically return a `404 Not Found` to the client.

Methods

Methods on a resource class which accept incoming requests are annotated with the HTTP methods they handle: `@GET`, `@POST`, `@PUT`, `@DELETE`, `@HEAD`, `@OPTIONS`, `@PATCH`.

Support for arbitrary new methods can be added via the `@HttpMethod` annotation. They also must be added to the *list of allowed methods*. This means, by default, methods such as `CONNECT` and `TRACE` are blocked, and will return a `405 Method Not Allowed` response.

If a request comes in which matches a resource class's path but has a method which the class doesn't support, Jersey will automatically return a `405 Method Not Allowed` to the client.

The return value of the method (in this case, a `NotificationList` instance) is then mapped to the *negotiated media type* this case, our resource only supports JSON, and so the `NotificationList` is serialized to JSON using Jackson.

Metrics

Every resource method can be annotated with `@Timed`, `@Metered`, and `@ExceptionMetered`. Dropwizard augments Jersey to automatically record runtime information about your resource methods.

- `@Timed` measures the duration of requests to a resource
- `@Metered` measures the rate at which the resource is accessed
- `@ExceptionMetered` measures how often exceptions occur processing the resource

Parameters

The annotated methods on a resource class can accept parameters which are mapped to from aspects of the incoming request. The `*Param` annotations determine which part of the request the data is mapped, and the parameter *type* determines how the data is mapped.

For example:

- A `@PathParam("user")`-annotated `String` takes the raw value from the `user` variable in the matched URI template and passes it into the method as a `String`.
- A `@QueryParam("count")`-annotated `IntParam` parameter takes the first `count` value from the request's query string and passes it as a `String` to `IntParam`'s constructor. `IntParam` (and all other `io.dropwizard.jersey.params.*` classes) parses the string as an `Integer`, returning a `400 Bad Request` if the value is malformed.
- A `@FormParam("name")`-annotated `Set<String>` parameter takes all the `name` values from a posted form and passes them to the method as a set of strings.
- A `*Param`–annotated `NonEmptyStringParam` will interpret empty strings as absent strings, which is useful in cases where the endpoint treats empty strings and absent strings as interchangeable.

What's noteworthy here is that you can actually encapsulate the vast majority of your validation logic using specialized parameter objects. See `AbstractParam` for details.

Request Entities

If you're handling request entities (e.g., an `application/json` object on a `PUT` request), you can model this as a parameter without a `*Param` annotation. In the *example code*, the `add` method provides a good example of this:

```
@POST
public Response add(@PathParam("user") LongParam userId,
                    @NotNull @Valid Notification notification) {
    final long id = store.add(userId.get(), notification);
    return Response.created(UriBuilder.fromResource(NotificationResource.class)
                                      .build(userId.get(), id)
                   .build();
}
```

Jersey maps the request entity to any single, unbound parameter. In this case, because the resource is annotated with `@Consumes(MediaType.APPLICATION_JSON)`, it uses the Dropwizard-provided Jackson support which, in addition to parsing the JSON and mapping it to an instance of `Notification`, also runs that instance through Dropwizard's *Constraining Entities*.

If the deserialized `Notification` isn't valid, Dropwizard returns a `422 Unprocessable Entity` response to the client.

Note: If a request entity parameter is just annotated with `@Valid`, it is still allowed to be `null`, so to ensure that the object is present and validated `@NotNull @Valid` is a powerful combination.

Media Types

Jersey also provides full content negotiation, so if your resource class consumes `application/json` but the client sends a `text/plain` entity, Jersey will automatically reply with a `406 Not Acceptable`. Jersey's even smart enough to use client-provided `q`-values in their `Accept` headers to pick the best response content type based on what both the client and server will support.

Responses

If your clients are expecting custom headers or additional information (or, if you simply desire an additional degree of control over your responses), you can return explicitly-built `Response` objects:

```
return Response.noContent().language(Locale.GERMAN).build();
```

In general, though, we recommend you return actual domain objects if at all possible. It makes *testing resources* much easier.

Error Handling

Almost as important as an application's happy path (receiving expected input and returning expected output) is an application behavior when something goes wrong.

If your resource class unintentionally throws an exception, Dropwizard will log that exception under the `ERROR` level (including stack traces) and return a terse, safe `application/json 500 Internal Server Error` response. The response will contain an ID that can be grepped out the server logs for additional information.

If your resource class needs to return an error to the client (e.g., the requested record doesn't exist), you have two options: throw a subclass of `Exception` or restructure your method to return a `Response`. If at all possible, prefer throwing `Exception` instances to returning `Response` objects, as that will make resource endpoints more self describing and easier to test.

The least instrusive way to map error conditions to a response is to throw a `WebApplicationException`:

```
@GET
@Path("/{collection}")
public Saying reduceCols(@PathParam("collection") String collection) {
    if (!collectionMap.containsKey(collection)) {
        final String msg = String.format("Collection %s does not exist", collection);
        throw new WebApplicationException(msg, Status.NOT_FOUND)
    }

    // ...
}
```

In this example a `GET` request to `/foobar` will return

```
{"code":404,"message":"Collection foobar does not exist"}
```

One can also take exceptions that your resource may throw and map them to appropriate responses. For instance, an endpoint may throw `IllegalArgumentException` and it may be worthy enough of a response to warrant a custom metric to track how often the event occurs. Here's an example of such an `ExceptionMapper`

```
public class IllegalArgumentExceptionMapper implements ExceptionMapper
↪<IllegalArgumentException> {
    private final Meter exceptions;
    public IllegalArgumentExceptionMapper(MetricRegistry metrics) {
        exceptions = metrics.meter(name(getClass(), "exceptions"));
    }

    @Override
    public Response toResponse(IllegalArgumentException e) {
        exceptions.mark();
        return Response.status(Status.BAD_REQUEST)
                .header("X-YOU-SILLY", "true")
                .type(MediaType.APPLICATION_JSON_TYPE)
                .entity(new ErrorMessage(Status.BAD_REQUEST.getStatusCode(),
                        "You passed an illegal argument!"))
                .build();
    }
}
```

and then registering the exception mapper:

```
@Override
public void run(final MyConfiguration conf, final Environment env) {
    env.jersey().register(new IllegalArgumentExceptionMapper(env.metrics()));
    env.jersey().register(new Resource());
}
```

Overriding Default Exception Mappers

To override a specific exception mapper, register your own class that implements the same `ExceptionMapper<T>` as one of the default. For instance, we can customize responses caused by Jackson exceptions:

```
public class JsonProcessingExceptionMapper implements ExceptionMapper
↪<JsonProcessingException> {
    @Override
    public Response toResponse(JsonProcessingException exception) {
```

(continues on next page)

(continued from previous page)

```
        // create the response
    }
}
```

With this method, one doesn't need to know what the default exception mappers are, as they are overridden if the user supplies a conflicting mapper. While not preferential, one can also disable all default exception mappers, by setting `server.registerDefaultExceptionMappers` to `false`. See the class `ExceptionMapperBinder` for a list of the default exception mappers.

URIs

While Jersey doesn't quite have first-class support for hyperlink-driven applications, the provided `UriBuilder` functionality does quite well.

Rather than duplicate resource URIs, it's possible (and recommended!) to initialize a `UriBuilder` with the path from the resource class itself:

```
UriBuilder.fromResource(UserResource.class).build(user.getId());
```

Testing

As with just about everything in Dropwizard, we recommend you design your resources to be testable. Dependencies which aren't request-injected should be passed in via the constructor and assigned to `final` fields.

Testing, then, consists of creating an instance of your resource class and passing it a mock. (Again: Mockito.)

```
public class NotificationsResourceTest {
    private final NotificationStore store = mock(NotificationStore.class);
    private final NotificationsResource resource = new NotificationsResource(store);

    @Test
    public void getsReturnNotifications() {
        final List<Notification> notifications = mock(List.class);
        when(store.fetch(1, 20)).thenReturn(notifications);

        final NotificationList list = resource.fetch(new LongParam("1"), new IntParam(
↪"20"));

        assertThat(list.getUserId(),
                   is(1L));

        assertThat(list.getNotifications(),
                    is(notifications));
    }
}
```

Caching

Adding a `Cache-Control` statement to your resource class is simple with Dropwizard:

```
@GET
@CacheControl(maxAge = 6, maxAgeUnit = TimeUnit.HOURS)
public String getCachableValue() {
```

(continues on next page)

(continued from previous page)

```
    return "yay";
}
```

The `@CacheControl` annotation will take all of the parameters of the `Cache-Control` header.

2.1.15 Representations

Representation classes are classes which, when handled to various Jersey `MessageBodyReader` and `MessageBodyWriter` providers, become the entities in your application's API. Dropwizard heavily favors JSON, but it's possible to map from any POJO to custom formats and back.

Basic JSON

Jackson is awesome at converting regular POJOs to JSON and back. This file:

```
public class Notification {
    private String text;

    public Notification(String text) {
        this.text = text;
    }

    @JsonProperty
    public String getText() {
        return text;
    }

    @JsonProperty
    public void setText(String text) {
        this.text = text;
    }
}
```

gets converted into this JSON:

```
{
    "text": "hey it's the value of the text field"
}
```

If, at some point, you need to change the JSON field name or the Java field without affecting the other, you can add an explicit field name to the `@JsonProperty` annotation.

If you prefer immutable objects rather than JavaBeans, that's also doable:

```
public class Notification {
    private final String text;

    @JsonCreator
    public Notification(@JsonProperty("text") String text) {
        this.text = text;
    }

    @JsonProperty("text")
    public String getText() {
```

(continues on next page)

(continued from previous page)

```
        return text;
    }
}
```

Advanced JSON

Not all JSON representations map nicely to the objects your application deals with, so it's sometimes necessary to use custom serializers and deserializers. Just annotate your object like this:

```
@JsonSerialize(using=FunkySerializer.class)
@JsonDeserialize(using=FunkyDeserializer.class)
public class Funky {
    // ...
}
```

Then make a `FunkySerializer` class which implements `JsonSerializer<Funky>` and a `FunkyDeserializer` class which implements `JsonDeserializer<Funky>`.

Snake Case

A common issue with JSON is the disagreement between `camelCase` and `snake_case` field names. Java and Javascript folks tend to like `camelCase`; Ruby, Python, and Perl folks insist on `snake_case`. To make Dropwizard automatically convert field names to `snake_case` (and back), just annotate the class with `@JsonSnakeCase`:

```
@JsonSnakeCase
public class Person {
    private final String firstName;

    @JsonCreator
    public Person(@JsonProperty String firstName) {
        this.firstName = firstName;
    }

    @JsonProperty
    public String getFirstName() {
        return firstName;
    }
}
```

This gets converted into this JSON:

```
{
    "first_name": "Coda"
}
```

Streaming Output

If your application happens to return lots of information, you may get a big performance and efficiency bump by using streaming output. By returning an object which implements Jersey's `StreamingOutput` interface, your method can stream the response entity in a chunk-encoded output stream. Otherwise, you'll need to fully construct your return value and *then* hand it off to be sent to the client.

HTML Representations

For generating HTML pages, check out Dropwizard's *views support*.

Custom Representations

Sometimes, though, you've got some wacky output format you need to produce or consume and no amount of arguing will make JSON acceptable. That's unfortunate but OK. You can add support for arbitrary input and output formats by creating classes which implement Jersey's `MessageBodyReader<T>` and `MessageBodyWriter<T>` interfaces. (Make sure they're annotated with `@Provider` and `@Produces("text/gibberish")` or `@Consumes("text/gibberish")`.) Once you're done, just add instances of them (or their classes if they depend on Jersey's `@Context` injection) to your application's `Environment` on initialization.

Jersey filters

There might be cases when you want to filter out requests or modify them before they reach your Resources. Jersey has a rich api for filters and interceptors that can be used directly in Dropwizard. You can stop the request from reaching your resources by throwing a `WebApplicationException`. Alternatively, you can use filters to modify inbound requests or outbound responses.

```
@Provider
public class DateNotSpecifiedFilter implements ContainerRequestFilter {
    @Override
    public void filter(ContainerRequestContext requestContext) throws IOException {
        String dateHeader = requestContext.getHeaderString(HttpHeaders.DATE);

        if (dateHeader == null) {
            Exception cause = new IllegalArgumentException("Date Header was not␣
↪specified");
            throw new WebApplicationException(cause, Response.Status.BAD_REQUEST);
        }
    }
}
```

This example filter checks the request for the "Date" header, and denies the request if was missing. Otherwise, the request is passed through.

Filters can be dynamically bound to resource methods using DynamicFeature:

```
@Provider
public class DateRequiredFeature implements DynamicFeature {
    @Override
    public void configure(ResourceInfo resourceInfo, FeatureContext context) {
        if (resourceInfo.getResourceMethod().getAnnotation(DateRequired.class) !=␣
↪null) {
            context.register(DateNotSpecifiedFilter.class);
        }
    }
}
```

The DynamicFeature is invoked by the Jersey runtime when the application is started. In this example, the feature checks for methods that are annotated with `@DateRequired` and registers the `DateNotSpecified` filter on those methods only.

You typically register the feature in your Application class, like so:

```
environment.jersey().register(DateRequiredFeature.class);
```

Servlet filters

Another way to create filters is by creating servlet filters. They offer a way to to register filters that apply both to servlet requests as well as resource requests. Jetty comes with a few bundled filters which may already suit your needs. If you want to create your own filter, this example demonstrates a servlet filter analogous to the previous example:

```
public class DateNotSpecifiedServletFilter implements javax.servlet.Filter {
    // Other methods in interface omitted for brevity

    @Override
    public void doFilter(ServletRequest request, ServletResponse response,␣
↪FilterChain chain) throws IOException, ServletException {
        if (request instanceof HttpServletRequest) {
            String dateHeader = ((HttpServletRequest) request).getHeader(HttpHeaders.
↪DATE);

            if (dateHeader != null) {
                chain.doFilter(request, response); // This signals that the request␣
↪should pass this filter
            } else {
                HttpServletResponse httpResponse = (HttpServletResponse) response;
                httpResponse.setStatus(HttpStatus.BAD_REQUEST_400);
                httpResponse.getWriter().print("Date Header was not specified");
            }
        }
    }
}
```

This servlet filter can then be registered in your Application class by wrapping it in `FilterHolder` and adding it to the application context together with a specification for which paths this filter should active. Here's an example:

```
environment.servlets().addFilter("DateNotSpecifiedServletFilter", new␣
↪DateNotSpecifiedServletFilter())
                       .addMappingForUrlPatterns(EnumSet.of(DispatcherType.REQUEST),␣
↪true, "/*");
```

2.1.16 How it's glued together

When your application starts up, it will spin up a Jetty HTTP server, see `DefaultServerFactory`. This server will have two handlers, one for your application port and the other for your admin port. The admin handler creates and registers the `AdminServlet`. This has a handle to all of the application healthchecks and metrics via the ServletContext.

The application port has an HttpServlet as well, this is composed of `DropwizardResourceConfig`, which is an extension of Jersey's resource configuration that performs scanning to find root resource and provider classes. Ultimately when you call `env.jersey().register(new SomeResource())`, you are adding to the `DropwizardResourceConfig`. This config is a jersey `Application`, so all of your application resources are served from one `Servlet`

`DropwizardResourceConfig` is where the various ResourceMethodDispatchAdapter are registered to enable the following functionality:

- Resource method requests with `@Timed`, `@Metered`, `@ExceptionMetered` are delegated to special dispatchers which decorate the metric telemetry
- Resources that return Guava Optional are unboxed. Present returns underlying type, and non-present 404s
- Resource methods that are annotated with `@CacheControl` are delegated to a special dispatcher that decorates on the cache control headers
- Enables using Jackson to parse request entities into objects and generate response entities from objects, all while performing validation

2.2 Dropwizard Client

The `dropwizard-client` module provides you with two different performant, instrumented HTTP clients so you can integrate your service with other web services: Apache HttpClient and Jersey Client.

2.2.1 Apache HttpClient

The underlying library for `dropwizard-client` is Apache's HttpClient, a full-featured, well-tested HTTP client library.

To create a *managed*, instrumented `HttpClient` instance, your *configuration class* needs an *http client configuration* instance:

```
public class ExampleConfiguration extends Configuration {
    @Valid
    @NotNull
    private HttpClientConfiguration httpClient = new HttpClientConfiguration();

    @JsonProperty("httpClient")
    public HttpClientConfiguration getHttpClientConfiguration() {
        return httpClient;
    }

    @JsonProperty("httpClient")
    public void setHttpClientConfiguration(HttpClientConfiguration httpClient) {
        this.httpClient = httpClient;
    }
}
```

Then, in your application's `run` method, create a new `HttpClientBuilder`:

```
@Override
public void run(ExampleConfiguration config,
                Environment environment) {
    final HttpClient httpClient = new HttpClientBuilder(environment).using(config.
↪getHttpClientConfiguration())
                                                                    .build(getName());
    environment.jersey().register(new ExternalServiceResource(httpClient));
}
```

Metrics

Dropwizard's `HttpClientBuilder` actually gives you an instrumented subclass which tracks the following pieces of data:

`org.apache.http.conn.ClientConnectionManager.available-connections` The number the number idle connections ready to be used to execute requests.

`org.apache.http.conn.ClientConnectionManager.leased-connections` The number of persistent connections currently being used to execute requests.

`org.apache.http.conn.ClientConnectionManager.max-connections` The maximum number of allowed connections.

`org.apache.http.conn.ClientConnectionManager.pending-connections` The number of connection requests being blocked awaiting a free connection

`org.apache.http.client.HttpClient.get-requests` The rate at which `GET` requests are being sent.

`org.apache.http.client.HttpClient.post-requests` The rate at which `POST` requests are being sent.

`org.apache.http.client.HttpClient.head-requests` The rate at which `HEAD` requests are being sent.

`org.apache.http.client.HttpClient.put-requests` The rate at which `PUT` requests are being sent.

`org.apache.http.client.HttpClient.delete-requests` The rate at which `DELETE` requests are being sent.

`org.apache.http.client.HttpClient.options-requests` The rate at which `OPTIONS` requests are being sent.

`org.apache.http.client.HttpClient.trace-requests` The rate at which `TRACE` requests are being sent.

`org.apache.http.client.HttpClient.connect-requests` The rate at which `CONNECT` requests are being sent.

`org.apache.http.client.HttpClient.move-requests` The rate at which `MOVE` requests are being sent.

`org.apache.http.client.HttpClient.patch-requests` The rate at which `PATCH` requests are being sent.

`org.apache.http.client.HttpClient.other-requests` The rate at which requests with none of the above methods are being sent.

Note: The naming strategy for the metrics associated requests is configurable. Specifically, the last part e.g. get-requests. What is displayed is `HttpClientMetricNameStrategies.METHOD_ONLY`, you can also include the host via `HttpClientMetricNameStrategies.HOST_AND_METHOD` or a url without query string via `HttpClientMetricNameStrategies.QUERYLESS_URL_AND_METHOD`

2.2.2 Jersey Client

If HttpClient is too low-level for you, Dropwizard also supports Jersey's Client API. Jersey's `Client` allows you to use all of the server-side media type support that your service uses to, for example, deserialize `application/json` request entities as POJOs.

To create a *managed*, instrumented `JerseyClient` instance, your *configuration class* needs an *jersey client configuration* instance:

```
public class ExampleConfiguration extends Configuration {
    @Valid
    @NotNull
    private JerseyClientConfiguration jerseyClient = new JerseyClientConfiguration();

    @JsonProperty("jerseyClient")
    public JerseyClientConfiguration getJerseyClientConfiguration() {
        return jerseyClient;
    }

    @JsonProperty("jerseyClient")
    public void setJerseyClientConfiguration(JerseyClientConfiguration jerseyClient) {
        this.jerseyClient = jerseyClient;
    }
}
```

Then, in your service's `run` method, create a new `JerseyClientBuilder`:

```
@Override
public void run(ExampleConfiguration config,
                Environment environment) {

    final Client client = new JerseyClientBuilder(environment).using(config.
↪getJerseyClientConfiguration())
                                                              .build(getName());
    environment.jersey().register(new ExternalServiceResource(client));
}
```

Configuration

The Client that Dropwizard creates deviates from the *Jersey Client Configuration* defaults. The default, in Jersey, is for a client to never timeout reading or connecting in a request, while in Dropwizard, the default is 500 milliseconds.

There are a couple of ways to change this behavior. The recommended way is to modify the *YAML configuration*. Alternatively, set the properties on the `JerseyClientConfiguration`, which will take effect for all built clients. On a per client basis, the configuration can be changed by utilizing the `property` method and, in this case, the Jersey Client Properties can be used.

> **Warning:** Do not try to change Jersey properties using Jersey Client Properties through the
>
> `withProperty(String propertyName, Object propertyValue)`
>
> method on the `JerseyClientBuilder`, because by default it's configured by Dropwizard's `HttpClientBuilder`, so the Jersey properties are ignored.

Rx Usage

To increase the ergonomics of asynchronous client requests, Jersey allows creation of rx-clients. You can instruct Dropwizard to create such a client:

```
@Override
public void run(ExampleConfiguration config,
                Environment environment) {

    final RxClient<RxCompletionStageInvoker> client =
        new JerseyClientBuilder(environment)
            .using(config.getJerseyClientConfiguration())
            .buildRx(getName(), RxCompletionStageInvoker.class);
    environment.jersey().register(new ExternalServiceResource(client));
}
```

`RxCompletionStageInvoker.class` is the Java 8 implementation and can be added to the pom:

```
<dependency>
    <groupId>org.glassfish.jersey.ext.rx</groupId>
    <artifactId>jersey-rx-client-java8</artifactId>
</dependency>
```

Alternatively, there are RxJava, Guava, and JSR-166e implementations.

By allowing Dropwizard to create the rx-client, the same thread pool that is utilized by traditional synchronous and asynchronous requests, is used for rx requests.

Proxy Authentication

The client can utilise a forward proxy, supporting both Basic and NTLM authentication schemes. Basic Auth against a proxy is simple:

```
proxy:
     host: '192.168.52.11'
     port: 8080
     scheme : 'https'
     auth:
       username: 'secret'
       password: 'stuff'
     nonProxyHosts:
       - 'localhost'
       - '192.168.52.*'
       - '*.example.com'
```

NTLM Auth is configured by setting the the relevant windows properties.

```
proxy:
     host: '192.168.52.11'
     port: 8080
     scheme : 'https'
     auth:
       username: 'secret'
       password: 'stuff'
       authScheme: 'NTLM'
       realm: 'realm'                    # optional, defaults to ANY_REALM
       hostname: 'workstation'           # optional, defaults to null but may be␣
↪required depending on your AD environment
       domain: 'HYPERCOMPUGLOBALMEGANET' # optional, defaults to null but may be␣
↪required depending on your AD environment
       credentialType: 'NT'
```

(continues on next page)

(continued from previous page)

```
    nonProxyHosts:
      - 'localhost'
      - '192.168.52.*'
      - '*.example.com'
```

2.3 Dropwizard JDBI

The `dropwizard-jdbi` module provides you with managed access to JDBI, a flexible and modular library for interacting with relational databases via SQL.

2.3.1 Configuration

To create a *managed*, instrumented `DBI` instance, your *configuration class* needs a `DataSourceFactory` instance:

```
public class ExampleConfiguration extends Configuration {
    @Valid
    @NotNull
    private DataSourceFactory database = new DataSourceFactory();

    @JsonProperty("database")
    public void setDataSourceFactory(DataSourceFactory factory) {
        this.database = factory;
    }

    @JsonProperty("database")
    public DataSourceFactory getDataSourceFactory() {
        return database;
    }
}
```

Then, in your service's `run` method, create a new `DBIFactory`:

```
@Override
public void run(ExampleConfiguration config, Environment environment) {
    final DBIFactory factory = new DBIFactory();
    final DBI jdbi = factory.build(environment, config.getDataSourceFactory(),
↪"postgresql");
    final UserDAO dao = jdbi.onDemand(UserDAO.class);
    environment.jersey().register(new UserResource(dao));
}
```

This will create a new *managed* connection pool to the database, a *health check* for connectivity to the database, and a new `DBI` instance for you to use.

Your service's configuration file will then look like this:

```
database:
  # the name of your JDBC driver
  driverClass: org.postgresql.Driver

  # the username
  user: pg-user
```

(continues on next page)

(continued from previous page)

```
  # the password
  password: iAMs00perSecrEET

  # the JDBC URL
  url: jdbc:postgresql://db.example.com/db-prod

  # any properties specific to your JDBC driver:
  properties:
    charSet: UTF-8

  # the maximum amount of time to wait on an empty pool before throwing an exception
  maxWaitForConnection: 1s

  # the SQL query to run when validating a connection's liveness
  validationQuery: "/* MyService Health Check */ SELECT 1"

  # the timeout before a connection validation queries fail
  validationQueryTimeout: 3s

  # the minimum number of connections to keep open
  minSize: 8

  # the maximum number of connections to keep open
  maxSize: 32

  # whether or not idle connections should be validated
  checkConnectionWhileIdle: false

  # the amount of time to sleep between runs of the idle connection validation,␣
↪abandoned cleaner and idle pool resizing
  evictionInterval: 10s

  # the minimum amount of time an connection must sit idle in the pool before it is␣
↪eligible for eviction
  minIdleTime: 1 minute
```

2.3.2 Usage

We highly recommend you use JDBI's SQL Objects API, which allows you to write DAO classes as interfaces:

```
public interface MyDAO {
  @SqlUpdate("create table something (id int primary key, name varchar(100))")
  void createSomethingTable();

  @SqlUpdate("insert into something (id, name) values (:id, :name)")
  void insert(@Bind("id") int id, @Bind("name") String name);

  @SqlQuery("select name from something where id = :id")
  String findNameById(@Bind("id") int id);
}

final MyDAO dao = database.onDemand(MyDAO.class);
```

This ensures your DAO classes are trivially mockable, as well as encouraging you to extract mapping code (e.g., `ResultSet` -> domain objects) into testable, reusable classes.

2.3.3 Exception Handling

By adding the `DBIExceptionsBundle` to your *application*, Dropwizard will automatically unwrap any thrown `SQLException` or `DBIException` instances. This is critical for debugging, since otherwise only the common wrapper exception's stack trace is logged.

2.3.4 Prepended Comments

If you're using JDBI's SQL Objects API (and you should be), `dropwizard-jdbi` will automatically prepend the SQL object's class and method name to the SQL query as an SQL comment:

```
/* com.example.service.dao.UserDAO.findByName */
SELECT id, name, email
FROM users
WHERE name = 'Coda';
```

This will allow you to quickly determine the origin of any slow or misbehaving queries.

2.3.5 Library Support

`dropwizard-jdbi` supports a number of popular libraries data types that can be automatically serialized into the appropriate SQL type. Here's a list of what integration `dropwizard-jdbi` provides:

- Guava: support for `Optional<T>` arguments and `ImmutableList<T>` and `ImmutableSet<T>` query results.
- Joda Time: support for `DateTime` arguments and `DateTime` fields in query results
- Java 8: support for `Optional<T>` and kin (`OptionalInt`, etc.) arguments and java.time arguments.

2.4 Dropwizard Migrations

The `dropwizard-migrations` module provides you with a wrapper for Liquibase database refactoring.

2.4.1 Configuration

Like *Dropwizard JDBI*, your *configuration class* needs a `DataSourceFactory` instance:

```
public class ExampleConfiguration extends Configuration {
    @Valid
    @NotNull
    private DataSourceFactory database = new DataSourceFactory();

    @JsonProperty("database")
    public DataSourceFactory getDataSourceFactory() {
        return database;
    }
}
```

2.4.2 Adding The Bundle

Then, in your application's `initialize` method, add a new `MigrationsBundle` subclass:

```
@Override
public void initialize(Bootstrap<ExampleConfiguration> bootstrap) {
    bootstrap.addBundle(new MigrationsBundle<ExampleConfiguration>() {
        @Override
        public DataSourceFactory getDataSourceFactory(ExampleConfiguration␣
↪configuration) {
            return configuration.getDataSourceFactory();
        }
    });
}
```

2.4.3 Defining Migrations

Your database migrations are stored in your Dropwizard project, in `src/main/resources/migrations.xml`. This file will be packaged with your application, allowing you to run migrations using your application's command-line interface. You can change the name of the migrations file used by overriding the `getMigrationsFileName()` method in `MigrationsBundle`.

For example, to create a new `people` table, you might create an initial `migrations.xml` like this:

```
<?xml version="1.0" encoding="UTF-8"?>

<databaseChangeLog
        xmlns="http://www.liquibase.org/xml/ns/dbchangelog"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://www.liquibase.org/xml/ns/dbchangelog
         http://www.liquibase.org/xml/ns/dbchangelog/dbchangelog-3.1.xsd">

    <changeSet id="1" author="codahale">
        <createTable tableName="people">
            <column name="id" type="bigint" autoIncrement="true">
                <constraints primaryKey="true" nullable="false"/>
            </column>
            <column name="fullName" type="varchar(255)">
                <constraints nullable="false"/>
            </column>
            <column name="jobTitle" type="varchar(255)"/>
        </createTable>
    </changeSet>
</databaseChangeLog>
```

For more information on available database refactorings, check the Liquibase documentation.

2.4.4 Checking Your Database's State

To check the state of your database, use the `db status` command:

```
java -jar hello-world.jar db status helloworld.yml
```

2.4.5 Dumping Your Schema

If your database already has an existing schema and you'd like to pre-seed your `migrations.xml` document, you can run the `db dump` command:

```
java -jar hello-world.jar db dump helloworld.yml
```

This will output a Liquibase change log with a changeset capable of recreating your database.

2.4.6 Tagging Your Schema

To tag your schema at a particular point in time (e.g., to make rolling back easier), use the `db tag` command:

```
java -jar hello-world.jar db tag helloworld.yml 2012-10-08-pre-user-move
```

2.4.7 Migrating Your Schema

To apply pending changesets to your database schema, run the `db migrate` command:

```
java -jar hello-world.jar db migrate helloworld.yml
```

Warning: This will potentially make irreversible changes to your database. Always check the pending DDL scripts by using the `--dry-run` flag first. This will output the SQL to be run to stdout.

Note: To apply only a specific number of pending changesets, use the `--count` flag.

2.4.8 Rolling Back Your Schema

To roll back changesets which have already been applied, run the `db rollback` command. You will need to specify either a **tag**, a **date**, or a **number of changesets** to roll back to:

```
java -jar hello-world.jar db rollback helloworld.yml --tag 2012-10-08-pre-user-move
```

Warning: This will potentially make irreversible changes to your database. Always check the pending DDL scripts by using the `--dry-run` flag first. This will output the SQL to be run to stdout.

2.4.9 Testing Migrations

To verify that a set of pending changesets can be fully rolled back, use the `db test` command, which will migrate forward, roll back to the original state, then migrate forward again:

```
java -jar hello-world.jar db test helloworld.yml
```

Warning: Do not run this in production, for obvious reasons.

2.4.10 Preparing A Rollback Script

To prepare a rollback script for pending changesets *before* they have been applied, use the `db prepare-rollback` command:

```
java -jar hello-world.jar db prepare-rollback helloworld.yml
```

This will output a DDL script to stdout capable of rolling back all unapplied changesets.

2.4.11 Generating Documentation

To generate HTML documentation on the current status of the database, use the `db generate-docs` command:

```
java -jar hello-world.jar db generate-docs helloworld.yml ~/db-docs/
```

2.4.12 Dropping All Objects

To drop all objects in the database, use the `db drop-all` command:

```
java -jar hello-world.jar db drop-all --confirm-delete-everything helloworld.yml
```

Warning: You need to specify the `--confirm-delete-everything` flag because this command **deletes everything in the database**. Be sure you want to do that first.

2.4.13 Fast-Forwarding Through A Changeset

To mark a pending changeset as applied (e.g., after having backfilled your `migrations.xml` with `db dump`), use the `db fast-forward` command:

```
java -jar hello-world.jar db fast-forward helloworld.yml
```

This will mark the next pending changeset as applied. You can also use the `--all` flag to mark all pending changesets as applied.

2.4.14 Support For Adding Multiple Migration Bundles

Assuming migrations need to be done for two different databases, you would need to have two different data source factories:

```
public class ExampleConfiguration extends Configuration {
    @Valid
    @NotNull
    private DataSourceFactory database1 = new DataSourceFactory();

    @Valid
```

(continues on next page)

(continued from previous page)

```
    @NotNull
    private DataSourceFactory database2 = new DataSourceFactory();

    @JsonProperty("database1")
    public DataSourceFactory getDb1DataSourceFactory() {
        return database1;
    }

    @JsonProperty("database2")
    public DataSourceFactory getDb2DataSourceFactory() {
        return database2;
    }
}
```

Now multiple migration bundles can be added with unique names like so:

```
@Override
public void initialize(Bootstrap<ExampleConfiguration> bootstrap) {
    bootstrap.addBundle(new MigrationsBundle<ExampleConfiguration>() {
        @Override
        public DataSourceFactory getDataSourceFactory(ExampleConfiguration␣
↪configuration) {
            return configuration.getDb1DataSourceFactory();
        }

        @Override
        public String name() {
            return "db1";
        }
    });

    bootstrap.addBundle(new MigrationsBundle<ExampleConfiguration>() {
        @Override
        public DataSourceFactory getDataSourceFactory(ExampleConfiguration␣
↪configuration) {
            return configuration.getDb2DataSourceFactory();
        }

        @Override
        public String name() {
            return "db2";
        }
    });
}
```

To migrate your schema:

```
java -jar hello-world.jar db1 migrate helloworld.yml
```

and

```
java -jar hello-world.jar db2 migrate helloworld.yml
```

Note: Whenever a name is added to a migration bundle, it becomes the command that needs to be run at the command line. eg: To check the state of your database, use the `status` command:

```
java -jar hello-world.jar db1 status helloworld.yml
```

or

```
java -jar hello-world.jar db2 status helloworld.yml
```

By default the migration bundle uses the "db" command. By overriding you can customize it to provide any name you want and have multiple migration bundles. Wherever the "db" command was being used, this custom name can be used.

There will also be a need to provide different change log migration files as well. This can be done as

```
java -jar hello-world.jar db1 migrate helloworld.yml --migrations <path_to_db1_
↪migrations.xml>
```

```
java -jar hello-world.jar db2 migrate helloworld.yml --migrations <path_to_db2_
↪migrations.xml>
```

2.4.15 More Information

If you are using databases supporting multiple schemas like PostgreSQL, Oracle, or H2, you can use the optional `--catalog` and `--schema` arguments to specify the database catalog and schema used for the Liquibase commands.

For more information on available commands, either use the `db --help` command, or for more detailed help on a specific command, use `db <cmd> --help`.

2.5 Dropwizard Hibernate

The `dropwizard-hibernate` module provides you with managed access to Hibernate, a powerful, industry-standard object-relation mapper (ORM).

2.5.1 Configuration

To create a *managed*, instrumented `SessionFactory` instance, your *configuration class* needs a `DataSourceFactory` instance:

```
public class ExampleConfiguration extends Configuration {
    @Valid
    @NotNull
    private DataSourceFactory database = new DataSourceFactory();

    @JsonProperty("database")
    public DataSourceFactory getDataSourceFactory() {
        return database;
    }
}
```

Then, add a `HibernateBundle` instance to your application class, specifying your entity classes and how to get a `DataSourceFactory` from your configuration subclass:

```
private final HibernateBundle<ExampleConfiguration> hibernate = new HibernateBundle
↪<ExampleConfiguration>(Person.class) {
    @Override
    public DataSourceFactory getDataSourceFactory(ExampleConfiguration configuration)
↪{
        return configuration.getDataSourceFactory();
    }
};

@Override
public void initialize(Bootstrap<ExampleConfiguration> bootstrap) {
    bootstrap.addBundle(hibernate);
}

@Override
public void run(ExampleConfiguration config, Environment environment) {
    final PersonDAO dao = new PersonDAO(hibernate.getSessionFactory());
    environment.jersey().register(new UserResource(dao));
}
```

This will create a new *managed* connection pool to the database, a *health check* for connectivity to the database, and a new `SessionFactory` instance for you to use in your DAO classes.

Your application's configuration file will then look like this:

```
database:
  # the name of your JDBC driver
  driverClass: org.postgresql.Driver

  # the username
  user: pg-user

  # the password
  password: iAMs00perSecrEET

  # the JDBC URL
  url: jdbc:postgresql://db.example.com/db-prod

  # any properties specific to your JDBC driver:
  properties:
    charSet: UTF-8
    hibernate.dialect: org.hibernate.dialect.PostgreSQLDialect

  # the maximum amount of time to wait on an empty pool before throwing an exception
  maxWaitForConnection: 1s

  # the SQL query to run when validating a connection's liveness
  validationQuery: "/* MyApplication Health Check */ SELECT 1"

  # the minimum number of connections to keep open
  minSize: 8

  # the maximum number of connections to keep open
  maxSize: 32

  # whether or not idle connections should be validated
  checkConnectionWhileIdle: false
```

2.5.2 Usage

Data Access Objects

Dropwizard comes with `AbstractDAO`, a minimal template for entity-specific DAO classes. It contains type-safe wrappers for most of `SessionFactory`'s common operations:

```
public class PersonDAO extends AbstractDAO<Person> {
    public PersonDAO(SessionFactory factory) {
        super(factory);
    }

    public Person findById(Long id) {
        return get(id);
    }

    public long create(Person person) {
        return persist(person).getId();
    }

    public List<Person> findAll() {
        return list(namedQuery("com.example.helloworld.core.Person.findAll"));
    }
}
```

Transactional Resource Methods

Dropwizard uses a declarative method of scoping transactional boundaries. Not all resource methods actually require database access, so the `@UnitOfWork` annotation is provided:

```
@GET
@Path("/{id}")
@Timed
@UnitOfWork
public Person findPerson(@PathParam("id") LongParam id) {
    return dao.findById(id.get());
}
```

This will automatically open a session, begin a transaction, call `findById`, commit the transaction, and finally close the session. If an exception is thrown, the transaction is rolled back.

Important: The Hibernate session is closed **before** your resource method's return value (e.g., the `Person` from the database), which means your resource method (or DAO) is responsible for initializing all lazily-loaded collections, etc., before returning. Otherwise, you'll get a `LazyInitializationException` thrown in your template (or `null` values produced by Jackson).

Transactional Resource Methods Outside Jersey Resources

Currently creating transactions with the *@UnitOfWork* annotation works out-of-box only for resources managed by Jersey. If you want to use it outside Jersey resources, e.g. in authenticators, you should instantiate your class with `UnitOfWorkAwareProxyFactory`.

```
SessionDao dao = new SessionDao(hibernateBundle.getSessionFactory());
ExampleAuthenticator exampleAuthenticator = new␣
↪UnitOfWorkAwareProxyFactory(hibernateBundle)
                .create(ExampleAuthenticator.class, SessionDao.class, dao);
```

It will create a proxy of your class, which will open a Hibernate session with a transaction around methods with the `@UnitOfWork` annotation.

2.5.3 Prepended Comments

Dropwizard automatically configures Hibernate to prepend a comment describing the context of all queries:

```
/* load com.example.helloworld.core.Person */
select
    person0_.id as id0_0_,
    person0_.fullName as fullName0_0_,
    person0_.jobTitle as jobTitle0_0_
from people person0_
where person0_.id=?
```

This will allow you to quickly determine the origin of any slow or misbehaving queries.

2.6 Dropwizard Authentication

The `dropwizard-auth` client provides authentication using either HTTP Basic Authentication or OAuth2 bearer tokens.

2.6.1 Authenticators

An authenticator is a strategy class which, given a set of client-provided credentials, possibly returns a principal (i.e., the person or entity on behalf of whom your service will do something).

Authenticators implement the `Authenticator<C, P extends Principal>` interface, which has a single method:

```
public class ExampleAuthenticator implements Authenticator<BasicCredentials, User> {
    @Override
    public Optional<User> authenticate(BasicCredentials credentials) throws␣
↪AuthenticationException {
        if ("secret".equals(credentials.getPassword())) {
            return Optional.of(new User(credentials.getUsername()));
        }
        return Optional.empty();
    }
}
```

This authenticator takes *basic auth credentials* and if the client-provided password is `secret`, authenticates the client as a `User` with the client-provided username.

If the password doesn't match, an absent `Optional` is returned instead, indicating that the credentials are invalid.

Warning: It's important for authentication services not to provide too much information in their errors. The fact that a username or email has an account may be meaningful to an attacker, so the `Authenticator` interface doesn't allow you to distinguish between a bad username and a bad password. You should only throw an `AuthenticationException` if the authenticator is **unable** to check the credentials (e.g., your database is down).

Caching

Because the backing data stores for authenticators may not handle high throughput (an RDBMS or LDAP server, for example), Dropwizard provides a decorator class which provides caching:

```
SimpleAuthenticator simpleAuthenticator = new SimpleAuthenticator();
CachingAuthenticator<BasicCredentials, User> cachingAuthenticator = new␣
↪CachingAuthenticator<>(
                           metricRegistry, simpleAuthenticator,
                           config.getAuthenticationCachePolicy());
```

Dropwizard can parse Guava's `CacheBuilderSpec` from the configuration policy, allowing your configuration file to look like this:

```
authenticationCachePolicy: maximumSize=10000, expireAfterAccess=10m
```

This caches up to 10,000 principals with an LRU policy, evicting stale entries after 10 minutes.

2.6.2 Authorizer

An authorizer is a strategy class which, given a principal and a role, decides if access is granted to the principal.

The authorizer implements the `Authorizer<P extends Principal>` interface, which has a single method:

```
public class ExampleAuthorizer implements Authorizer<User> {
    @Override
    public boolean authorize(User user, String role) {
        return user.getName().equals("good-guy") && role.equals("ADMIN");
    }
}
```

2.6.3 Basic Authentication

The `AuthDynamicFeature` with the `BasicCredentialAuthFilter` and `RolesAllowedDynamicFeature` enables HTTP Basic authentication and authorization; requires an authenticator which takes instances of `BasicCredentials`. If you don't use authorization, then `RolesAllowedDynamicFeature` is not required.

```
@Override
public void run(ExampleConfiguration configuration,
                Environment environment) {
    environment.jersey().register(new AuthDynamicFeature(
            new BasicCredentialAuthFilter.Builder<User>()
                .setAuthenticator(new ExampleAuthenticator())
                .setAuthorizer(new ExampleAuthorizer())
                .setRealm("SUPER SECRET STUFF")
```

(continues on next page)

(continued from previous page)

```
                .buildAuthFilter()));
    environment.jersey().register(RolesAllowedDynamicFeature.class);
    //If you want to use @Auth to inject a custom Principal type into your resource
    environment.jersey().register(new AuthValueFactoryProvider.Binder<>(User.class));
}
```

2.6.4 OAuth2

The `AuthDynamicFeature` with `OAuthCredentialAuthFilter` and `RolesAllowedDynamicFeature` enables OAuth2 bearer-token authentication and authorization; requires an authenticator which takes instances of `String`. If you don't use authorization, then `RolesAllowedDynamicFeature` is not required.

```
@Override
public void run(ExampleConfiguration configuration,
                Environment environment) {
    environment.jersey().register(new AuthDynamicFeature(
        new OAuthCredentialAuthFilter.Builder<User>()
            .setAuthenticator(new ExampleOAuthAuthenticator())
            .setAuthorizer(new ExampleAuthorizer())
            .setPrefix("Bearer")
            .buildAuthFilter()));
    environment.jersey().register(RolesAllowedDynamicFeature.class);
    //If you want to use @Auth to inject a custom Principal type into your resource
    environment.jersey().register(new AuthValueFactoryProvider.Binder<>(User.class));
}
```

2.6.5 Chained Factories

The `ChainedAuthFilter` enables usage of various authentication factories at the same time.

```
@Override
public void run(ExampleConfiguration configuration,
                Environment environment) {
    AuthFilter basicCredentialAuthFilter = new BasicCredentialAuthFilter.Builder<>()
            .setAuthenticator(new ExampleBasicAuthenticator())
            .setAuthorizer(new ExampleAuthorizer())
            .setPrefix("Basic")
            .buildAuthFilter();

    AuthFilter oauthCredentialAuthFilter = new OAuthCredentialAuthFilter.Builder<>()
            .setAuthenticator(new ExampleOAuthAuthenticator())
            .setAuthorizer(new ExampleAuthorizer())
            .setPrefix("Bearer")
            .buildAuthFilter();

    List<AuthFilter> filters = Lists.newArrayList(basicCredentialAuthFilter,␣
↪oauthCredentialAuthFilter);
    environment.jersey().register(new AuthDynamicFeature(new␣
↪ChainedAuthFilter(filters)));
    environment.jersey().register(RolesAllowedDynamicFeature.class);
    //If you want to use @Auth to inject a custom Principal type into your resource
```

(continues on next page)

(continued from previous page)

```
    environment.jersey().register(new AuthValueFactoryProvider.Binder<>(User.class));
}
```

For this to work properly, all chained factories must produce the same type of principal, here `User`.

2.6.6 Protecting Resources

There are two ways to protect a resource. You can mark your resource method with one of the following annotations:

- `@PermitAll`. All authenticated users will have access to the method.
- `@RolesAllowed`. Access will be granted to the users with the specified roles.
- `@DenyAll`. No access will be granted to anyone.

Note: You can use `@RolesAllowed`, `@PermitAll` on the class level. Method annotations take precedence over the class ones.

Alternatively, you can annotate the parameter representing your principal with `@Auth`. Note you must register a jersey provider to make this work.

```
environment.jersey().register(new AuthValueFactoryProvider.Binder<>(User.class));

@RolesAllowed("ADMIN")
@GET
public SecretPlan getSecretPlan(@Auth User user) {
    return dao.findPlanForUser(user);
}
```

You can also access the Principal by adding a parameter to your method `@Context SecurityContext context`. Note this will not automatically register the servlet filter which performs authentication. You will still need to add one of `@PermitAll`, `@RolesAllowed`, or `@DenyAll`. This is not the case with `@Auth`. When that is present, the auth filter is automatically registered to facilitate users upgrading from older versions of Dropwizard

```
@RolesAllowed("ADMIN")
@GET
public SecretPlan getSecretPlan(@Context SecurityContext context) {
    User userPrincipal = (User) context.getUserPrincipal();
    return dao.findPlanForUser(user);
}
```

If there are no provided credentials for the request, or if the credentials are invalid, the provider will return a scheme-appropriate `401 Unauthorized` response without calling your resource method.

Optional protection

Resource methods can be _optionally_ protected by representing the principal as an `Optional`. In such cases, the `Optional` resource method argument will be populated with the principal, if present. Otherwise, the argument will be `Optional.empty`.

For instance, say you have an endpoint that should display a logged-in user's name, but return an anonymous reply for unauthenticated requests. You need to implement a custom filter which injects a security context containing the principal if it exists, without performing authentication.

```
@GET
public String getGreeting(@Auth Optional<User> userOpt) {
    if (userOpt.isPresent()) {
        return "Hello, " + userOpt.get().getName() + "!";
    } else {
        return "Greetings, anonymous visitor!"
    }
}
```

For optionally-protected resources, requests with invalid auth will be treated the same as those with no provided auth credentials. That is to say, requests that _fail_ to meet an authenticator or authorizer's requirements result in an empty principal being passed to the resource method.

2.6.7 Testing Protected Resources

Add this dependency into your `pom.xml` file:

```
<dependencies>
  <dependency>
    <groupId>io.dropwizard</groupId>
    <artifactId>dropwizard-testing</artifactId>
    <version>${dropwizard.version}</version>
  </dependency>
  <dependency>
    <groupId>org.glassfish.jersey.test-framework.providers</groupId>
    <artifactId>jersey-test-framework-provider-grizzly2</artifactId>
    <version>${jersey.version}</version>
    <exclusions>
      <exclusion>
        <groupId>javax.servlet</groupId>
        <artifactId>javax.servlet-api</artifactId>
      </exclusion>
      <exclusion>
        <groupId>junit</groupId>
        <artifactId>junit</artifactId>
      </exclusion>
    </exclusions>
  </dependency>
</dependencies>
```

When you build your `ResourceTestRule`, add the `GrizzlyWebTestContainerFactory` line.

```
@Rule
public ResourceTestRule rule = ResourceTestRule
        .builder()
        .setTestContainerFactory(new GrizzlyWebTestContainerFactory())
        .addProvider(new AuthDynamicFeature(new OAuthCredentialAuthFilter.Builder
↪<User>()
                .setAuthenticator(new MyOAuthAuthenticator())
                .setAuthorizer(new MyAuthorizer())
                .setRealm("SUPER SECRET STUFF")
                .setPrefix("Bearer")
                .buildAuthFilter()))
        .addProvider(RolesAllowedDynamicFeature.class)
        .addProvider(new AuthValueFactoryProvider.Binder<>(User.class))
```

(continues on next page)

(continued from previous page)

```
        .addResource(new ProtectedResource())
        .build();
```

In this example, we are testing the oauth authentication, so we need to set the header manually.

```
@Test
public void testProtected() throws Exception {
    final Response response = rule.target("/protected")
            .request(MediaType.APPLICATION_JSON_TYPE)
            .header("Authorization", "Bearer TOKEN")
            .get();

    assertThat(response.getStatus()).isEqualTo(200);
}
```

2.6.8 Multiple Principals and Authenticators

In some cases you may want to use different authenticators/authentication schemes for different resources. For example you may want Basic authentication for one resource and OAuth for another resource, at the same time using a different *Principal* for each authentication scheme.

For this use case, there is the `PolymorphicAuthDynamicFeature` and the `PolymorphicAuthValueFactoryProvider`. With these two components, we can use different combinations of authentication schemes/authenticators/authorizers/principals. To use this feature, we need to do a few things:

- Register the `PolymorphicAuthDynamicFeature` with a map that maps principal types to authentication filters.
- Register the `PolymorphicAuthValueFactoryProvider` with a set of principal classes that you will be using.
- Annotate your resource method `Principal` parameters with `@Auth`.

As an example, the following code configures both OAuth and Basic authentication, using a different principal for each.

```
final AuthFilter<BasicCredentials, BasicPrincipal> basicFilter
        = new BasicCredentialAuthFilter.Builder<BasicPrincipal>()
                .setAuthenticator(new ExampleAuthenticator())
                .setRealm("SUPER SECRET STUFF")
                .buildAuthFilter());
final AuthFilter<String, OAuthPrincipal> oauthFilter
        = new OAuthCredentialAuthFilter.Builder<OAuthPrincipal>()
                .setAuthenticator(new ExampleOAuthAuthenticator())
                .setPrefix("Bearer")
                .buildAuthFilter());

final PolymorphicAuthDynamicFeature feature = new PolymorphicAuthDynamicFeature<>(
    ImmutableMap.of(
        BasicPrincipal.class, basicFilter,
        OAuthPrincipal.class, oauthFilter));
final AbstractBinder binder = new PolymorphicAuthValueFactoryProvider.Binder<>(
    ImmutableSet.of(BasicPrincipal.class, OAuthPrincipal.class));
```

(continues on next page)

(continued from previous page)

```
environment.jersey().register(feature);
environment.jersey().register(binder);
```

Now we are able to do something like the following

```
@GET
public Response basicAuthResource(@Auth BasicPrincipal principal) {}

@GET
public Response oauthResource(@Auth OAuthPrincipal principal) {}
```

The first resource method will use Basic authentication while the second one will use OAuth.

Note that with the above example, only *authentication* is configured. If you also want *authorization*, the following steps will need to be taken.

- Register the `RolesAllowedDynamicFeature` with the application.
- Make sure you add `Authorizers` when you build your `AuthFilters`.
- Annotate the resource *method* with the authorization annotation. Unlike the note earlier in this document that says authorization annotations are allowed on classes, with this poly feature, currently that is not supported. The annotation MUST go on the resource *method*

So continuing with the previous example you should add the following configurations

```
... = new BasicCredentialAuthFilter.Builder<BasicPrincipal>()
        .setAuthorizer(new ExampleAuthorizer())..  // set authorizer

... = new OAuthCredentialAuthFilter.Builder<OAuthPrincipal>()
        .setAuthorizer(new ExampleAuthorizer())..  // set authorizer

environment.jersey().register(RolesAllowedDynamicFeature.class);
```

Now we can do

```
@GET
@RolesAllowed({ "ADMIN" })
public Response baseAuthResource(@Auth BasicPrincipal principal) {}

@GET
@RolesAllowed({ "ADMIN" })
public Response oauthResource(@Auth OAuthPrincipal principal) {}
```

Note: The polymorphic auth feature *SHOULD NOT* be used with any other `AuthDynamicFeature`. Doing so may have undesired effects.

2.7 Dropwizard Forms

The `dropwizard-forms` module provides you with a support for multi-part forms via Jersey.

2.7.1 Adding The Bundle

Then, in your application's `initialize` method, add a new `MultiPartBundle` subclass:

```
@Override
public void initialize(Bootstrap<ExampleConfiguration> bootstrap) {
    bootstrap.addBundle(new MultiPartBundle());
}
```

2.7.2 Testing

To test resources that utilize multi-part form features, one must add `MultiPartFeature.class` to the `ResourceTestRule` as a provider, and register it on the client like the following:

```
public class MultiPartTest {
    @ClassRule
    public static final ResourceTestRule resource = ResourceTestRule.builder()
            .addProvider(MultiPartFeature.class)
            .addResource(new TestResource())
            .build();

    @Test
    public void testClientMultipart() {
        final FormDataMultiPart multiPart = new FormDataMultiPart()
                .field("test-data", "Hello Multipart");
        final String response = resource.target("/test")
                .register(MultiPartFeature.class)
                .request()
                .post(Entity.entity(multiPart, multiPart.getMediaType()), String.
↪class);
        assertThat(response).isEqualTo("Hello Multipart");
    }

    @Path("test")
    public static class TestResource {
        @POST
        @Consumes(MediaType.MULTIPART_FORM_DATA)
        public String post(@FormDataParam("test-data") String testData) {
            return testData;
        }
    }
}
```

2.7.3 More Information

For additional and more detailed documentation about the Jersey multi-part support, please refer to the documentation in the Jersey User Guide and Javadoc.

2.8 Dropwizard Validation

Dropwizard comes with a host of validation tools out of the box to allow endpoints to return meaningful error messages when constraints are violated. Hibernate Validator is packaged with Dropwizard, so what can be done in Hibernate Validator, can be done with Dropwizard.

2.8.1 Validations

Almost anything can be validated on resource endpoints. To give a quick example, the following endpoint doesn't allow a null or empty `name` query parameter.

```
@GET
public String find(@QueryParam("name") @NotEmpty String arg) {
    // ...
}
```

If a client sends an empty or nonexistent name query param, Dropwizard will respond with a `400 Bad Request` code with the error: `query param name may not be empty`.

Additionally, annotations such as `HeaderParam`, `CookieParam`, `FormParam`, etc, can be constrained with violations giving descriptive errors and 400 status codes.

Constraining Entities

If we're accepting client-provided `Person`, we probably want to ensure that the `name` field of the object isn't `null` or blank in the request. We can do this as follows:

```
public class Person {

    @NotEmpty // ensure that name isn't null or blank
    private final String name;

    @JsonCreator
    public Person(@JsonProperty("name") String name) {
        this.name = name;
    }

    @JsonProperty("name")
    public String getName() {
        return name;
    }
}
```

Then, in our resource class, we can add the `@Valid` annotation to the `Person` annotation:

```
@PUT
public Person replace(@NotNull @Valid Person person) {
    // ...
}
```

If the name field is missing, Dropwizard will return a `422 Unprocessable Entity` response detailing the validation errors: `name may not be empty`

Note: You don't need `@Valid` when the type you are validating can be validated directly (`int`, `String`, `Integer`). If a class has fields that need validating, then instances of the class must be marked `@Valid`. For more information, see the Hibernate Validator documentation on Object graphs and Cascaded validation.

Since our entity is also annotated with `@NotNull`, Dropwizard will also guard against `null` input with a response stating that the body must not be null.

`Optional<T>` Constraints

If an entity, field, or parameter is not required, it can be wrapped in an `Optional<T>`, but the inner value can still be constrained with the `@UnwrapValidatedValue` annotation. If the `Optional` is absent, then the constraints are not applied.

Note: Be careful when using constraints with `*Param` annotations on `Optional<String>` parameters as there is a subtle, but important distinction between null and empty. If a client requests `bar?q=`, `q` will evaluate to `Optional.of("")`. If you want `q` to evaluate to `Optional.absent()` in this situation, change the type to `NonEmptyStringParam`

Note: Param types such as `IntParam` and `NonEmptyStringParam` can also be constrained.

There is a caveat regarding `@UnwrapValidatedValue` and `*Param` types, as there still are some cumbersome situations when constraints need to be applied to the container and the value.

```
@POST
// The @NotNull is supposed to mean that the parameter is required but the Max(3) is␣
↪supposed to
// apply to the contained integer. Currently, this code will fail saying that Max can
↪'t
// be applied on an IntParam
public List<Person> createNum(@QueryParam("num") @UnwrapValidatedValue(false)
                              @NotNull @Max(3) IntParam num) {
    // ...
}

@GET
// Similarly, the underlying validation framework can't unwrap nested types (an␣
↪integer wrapped
// in an IntParam wrapped in an Optional), regardless if the @UnwrapValidatedValue is␣
↪used
public Person retrieve(@QueryParam("num") @Max(3) Optional<IntParam> num) {
    // ...
}
```

To work around these limitations, if the parameter is required check for it in the endpoint and throw an exception, else use `@DefaultValue` or move the `Optional` into the endpoint.

```
@POST
// Workaround to handle required int params and validations
public List<Person> createNum(@QueryParam("num") @Max(3) IntParam num) {
    if (num == null) {
        throw new WebApplicationException("query param num must not be null", 400);
    }
    // ...
}

@GET
// Workaround to handle optional int params and validations with DefaultValue
```

(continues on next page)

(continued from previous page)

```
public Person retrieve(@QueryParam("num") @DefaultValue("0") @Max(3) IntParam num) {
    // ...
}

@GET
// Workaround to handle optional int params and validations with Optional
public Person retrieve2(@QueryParam("num") @Max(3) IntParam num) {
    Optional.fromNullable(num);
    // ...
}
```

Enum Constraints

Given the following enum:

```
public enum Choice {
    OptionA,
    OptionB,
    OptionC
}
```

And the endpoint:

```
@GET
public String getEnum(@NotNull @QueryParam("choice") Choice choice) {
    return choice.toString();
}
```

One can expect Dropwizard not only to ensure that the query parameter exists, but to also provide the client a list of valid options `query param choice must be one of [OptionA, OptionB, OptionC]` when an invalid parameter is provided. The enum that the query parameter is deserialized into is first attempted on the enum's `name()` field and then `toString()`. During the case insensitive comparisons, the query parameter has whitespace removed with dashes and dots normalized to underscores. This logic is also used when deserializing request body's that contain enums.

Return Value Validations

It's reasonable to want to make guarantees to clients regarding the server response. For example, you may want to assert that no response will ever be `null`, and if an endpoint creates a `Person` that the person is valid.

```
@POST
@NotNull
@Valid
public Person create() {
    return new Person(null);
}
```

In this instance, instead of returning someone with a null name, Dropwizard will return an `HTTP 500 Internal Server Error` with the error `server response name may not be empty`, so the client knows the server failed through no fault of their own.

Analogous to an empty request body, an empty entity annotated with `@NotNull` will return `server response may not be null`

Warning: Be careful when using return value constraints when endpoints satisfy all of the following:

- Function name starts with `get`
- No arguments
- The return value has validation constraints

If an endpoint satisfies these conditions, whenever a request is processed by the resource that endpoint will be additionally invoked. To give a concrete example:

```
@Path("/")
public class ValidatedResource {
    private AtomicLong counter = new AtomicLong();

    @GET
    @Path("/foo")
    @NotEmpty
    public String getFoo() {
        counter.getAndIncrement();
        return "";
    }

    @GET
    @Path("/bar")
    public String getBar() {
        return "";
    }
}
```

If a `/foo` is requested then `counter` will have increment by 2, and if `/bar` is requested then `counter` will increment by 1. It is our hope that such endpoints are few, far between, and documented thoroughly.

2.8.2 Limitations

Jersey allows for `BeanParam` to have setters with `*Param` annotations. While nice for simple transformations it does obstruct validation, so clients won't receive as instructive of error messages. The following example shows the behavior:

```
@Path("/root")
@Produces(MediaType.APPLICATION_JSON)
public class Resource {

    @GET
    @Path("params")
    public String getBean(@Valid @BeanParam MyBeanParams params) {
        return params.getField();
    }

    public static class MyBeanParams {
        @NotEmpty
        private String field;

        public String getField() {
            return field;
        }
```

(continues on next page)

(continued from previous page)

```
        @QueryParam("foo")
        public void setField(String field) {
            this.field = Strings.nullToEmpty(field).trim();
        }
    }
}
```

A client submitting the query parameter `foo` as blank will receive the following error message:

```
{"errors":["getBean.arg0.field may not be empty"]}
```

Workarounds include:

- Name `BeanParam` fields the same as the `*Param` annotation values
- Supply validation message on annotation: `@NotEmpty(message = "query param foo must not be empty")`
- Perform transformations and validations on `*Param` inside endpoint

The same kind of limitation applies for *Configuration* objects:

```
public class MyConfiguration extends Configuration {
    @NotNull
    @JsonProperty("foo")
    private String baz;
}
```

Even though the property's name is `foo`, the error when property is null will be:

```
* baz may not be null
```

2.8.3 Annotations

In addition to the annotations defined in Hibernate Validator, Dropwizard contains another set of annotations, which are briefly shown below.

```
public class Person {
    @NotEmpty
    private final String name;

    @NotEmpty
    @OneOf(value = {"m", "f"}, ignoreCase = true, ignoreWhitespace = true)
    // @OneOf forces a value to value within certain values.
    private final String gender;

    @Max(10)
    @Min(0)
    // The integer contained, if present, can attain a min value of 0 and a max of 10.
    private final Optional<Integer> animals;

    @JsonCreator
    public Person(@JsonProperty("name") String name) {
        this.name = name;
    }
```

(continues on next page)

(continued from previous page)

```
    @JsonProperty("name")
    public String getName() {
        return name;
    }

    // Method that must return true for the object to be valid
    @ValidationMethod(message="name may not be Coda")
    @JsonIgnore
    public boolean isNotCoda() {
        return !"Coda".equals(name);
    }
}
```

The reason why Dropwizard defines `@ValidationMethod` is that more complex validations (for example, cross-field comparisons) are often hard to do using declarative annotations. Adding `@ValidationMethod` to any `boolean`-returning method which begins with `is` is a short and simple workaround:

Note: Due to the rather daft JavaBeans conventions, when using `@ValidationMethod`, the method must begin with `is` (e.g., `#isValidPortRange()`. This is a limitation of Hibernate Validator, not Dropwizard.

Validating Grouped Constraints with `@Validated`

The `@Validated` annotation allows for validation groups to be specifically set, instead of the default group. This is useful when different endpoints share the same entity but may have different validation requirements.

Going back to our favorite `Person` class. Let's say in the initial version of our API, `name` has to be non-empty, but realized that business requirements changed and a name can't be longer than 5 letters. Instead of switching out the API from unsuspecting clients, we can accept both versions of the API but at different endpoints.

```
// We're going to create a group of validations for each version of our API
public interface Version1Checks { }

// Our second version will extend Hibernate Validator Default class so that any␣
↪validation
// annotation without an explicit group will also be validated with this version
public interface Version2Checks extends Default { }

public class Person {
    @NotEmpty(groups = Version1Checks.class)
    @Length(max = 5, groups = Version2Checks.class)
    private String name;

    @JsonCreator
    public Person(@JsonProperty("name") String name) {
        this.name = name;
    }

    @JsonProperty
    public String getName() {
        return name;
    }
}
```

(continues on next page)

(continued from previous page)

```
@Path("/person")
@Produces(MediaType.APPLICATION_JSON)
public class PersonResource {

    // For the v1 endpoint, we'll validate with the version1 class, so we'll need to␣
↪change the
    // group of the NotNull annotation from the default of Default.class to␣
↪Version1Checks.class
    @POST
    @Path("/v1")
    public void createPersonV1(
        @NotNull(groups = Version1Checks.class)
        @Valid
        @Validated(Version1Checks.class)
        Person person
    ) {
        // implementation ...
    }

    // For the v2 endpoint, we'll validate with version1 and version2, which␣
↪implicitly
    // adds in the Default.class.
    @POST
    @Path("/v2")
    public void createPersonV2(
        @NotNull
        @Valid
        @Validated({Version1Checks.class, Version2Checks.class})
        Person person
    ) {
        // implementation ...
    }
}
```

Now when clients hit `/person/v1` the `Person` entity will be checked by all the constraints that are a part of the `Version1Checks` group. If `/person/v2` is hit, then all validations are performed.

Warning: If the *Version1Checks* group wasn't set for the *@NotNull* annotation for the v1 endpoint, the annotation would not have had any effect and a null pointer exception would have occurred when a property of a person is accessed. Dropwizard tries to protect against this class of bug by disallowing multiple *@Validated* annotations on an endpoint that contain different groups.

2.8.4 Testing

It is critical to test the constraints so that you can ensure the assumptions about the data hold and see what kinds of error messages clients will receive for bad input. The recommended way for testing annotations is through *Testing Resources*, as Dropwizard does a bit of magic behind the scenes when a constraint violation occurs to set the response's status code and ensure that the error messages are user friendly.

```
@Test
public void personNeedsAName() {
    // Tests what happens when a person with a null name is sent to
```

(continues on next page)

(continued from previous page)

```
    // the endpoint.
    final Response post = resources.target("/person/v1").request()
            .post(Entity.json(new Person(null)));

    // Clients will receive a 422 on bad request entity
    assertThat(post.getStatus()).isEqualTo(422);

    // Check to make sure that errors are correct and human readable
    ValidationErrorMessage msg = post.readEntity(ValidationErrorMessage.class);
    assertThat(msg.getErrors())
            .containsOnly("name may not be empty");
}
```

2.8.5 Extending

While Dropwizard provides good defaults for validation error messages, one can customize the response through an `ExceptionMapper<JerseyViolationException>`:

```
/** Return a generic response depending on if it is a client or server error */
public class MyJerseyViolationExceptionMapper implements ExceptionMapper
↪<JerseyViolationException> {
    @Override
    public Response toResponse(final JerseyViolationException exception) {
        final Set<ConstraintViolation<?>> violations = exception.
↪getConstraintViolations();
        final Invocable invocable = exception.getInvocable();
        final int status = ConstraintMessage.determineStatus(violations, invocable);
        return Response.status(status)
                .type(MediaType.TEXT_PLAIN_TYPE)
                .entity(status >= 500 ? "Server error" : "Client error")
                .build();
    }
}
```

To register `MyJerseyViolationExceptionMapper` and have it override the default:

```
@Override
public void run(final MyConfiguration conf, final Environment env) {
    env.jersey().register(new MyJerseyViolationExceptionMapper());
    env.jersey().register(new Resource());
}
```

Dropwizard calculates the validation error message through `ConstraintMessage.getMessage`.

If you need to validate entities outside of resource endpoints, the validator can be accessed in the `Environment` when the application is first ran.

```
Validator validator = environment.getValidator();
Set<ConstraintViolation> errors = validator.validate(/* instance of class */)
```

2.9 Dropwizard Views

The `dropwizard-views-mustache` & `dropwizard-views-freemarker` modules provide you with simple, fast HTML views using either FreeMarker or Mustache.

To enable views for your *Application*, add the `ViewBundle` in the `initialize` method of your Application class:

```
public void initialize(Bootstrap<MyConfiguration> bootstrap) {
    bootstrap.addBundle(new ViewBundle<MyConfiguration>());
}
```

You can pass configuration through to view renderers by overriding `getViewConfiguration`:

```
public void initialize(Bootstrap<MyConfiguration> bootstrap) {
    bootstrap.addBundle(new ViewBundle<MyConfiguration>() {
        @Override
        public Map<String, Map<String, String>> getViewConfiguration(MyConfiguration␣
↪config) {
            return config.getViewRendererConfiguration();
        }
    });
}
```

The returned map should have, for each extension (such as `.ftl`), a `Map<String, String>` describing how to configure the renderer. Specific keys and their meanings can be found in the FreeMarker and Mustache documentation:

```
views:
  .ftl:
    strict_syntax: yes
```

Then, in your *resource method*, add a `View` class:

```
public class PersonView extends View {
    private final Person person;

    public PersonView(Person person) {
        super("person.ftl");
        this.person = person;
    }

    public Person getPerson() {
        return person;
    }
}
```

`person.ftl` is the path of the template relative to the class name. If this class was `com.example.service.PersonView`, Dropwizard would then look for the file `src/main/resources/com/example/service/person.ftl`.

If your template ends with `.ftl`, it'll be interpreted as a FreeMarker template. If it ends with `.mustache`, it'll be interpreted as a Mustache template.

Tip: Dropwizard Freemarker Views also support localized template files. It picks up the client's locale from their `Accept-Language`, so you can add a French template in `person_fr.ftl` or a Canadian template in `person_en_CA.ftl`.

Your template file might look something like this:

```
<#-- @ftlvariable name="" type="com.example.views.PersonView" -->
<html>
    <body>
        <!-- calls getPerson().getName() and sanitizes it -->
        <h1>Hello, ${person.name?html}!</h1>
    </body>
</html>
```

The `@ftlvariable` lets FreeMarker (and any FreeMarker IDE plugins you may be using) know that the root object is a `com.example.views.PersonView` instance. If you attempt to call a property which doesn't exist on `PersonView` – `getConnectionPool()`, for example – it will flag that line in your IDE.

Once you have your view and template, you can simply return an instance of your `View` subclass:

```
@Path("/people/{id}")
@Produces(MediaType.TEXT_HTML)
public class PersonResource {
    private final PersonDAO dao;

    public PersonResource(PersonDAO dao) {
        this.dao = dao;
    }

    @GET
    public PersonView getPerson(@PathParam("id") String id) {
        return new PersonView(dao.find(id));
    }
}
```

Tip: Jackson can also serialize your views, allowing you to serve both `text/html` and `application/json` with a single representation class.

For more information on how to use FreeMarker, see the FreeMarker documentation.

For more information on how to use Mustache, see the Mustache and Mustache.java documentation.

2.9.1 Template Errors

By default, if there is an error with the template (eg. the template file is not found or there is a compilation error with the template), the user will receive a `500 Internal Server Error` with a generic HTML message. The exact error will logged under error mode.

To customize the behavior, create an exception mapper that will override the default one by looking for `ViewRenderException`:

```
env.jersey().register(new ExtendedExceptionMapper<WebApplicationException>() {
    @Override
    public Response toResponse(WebApplicationException exception) {
        // Return a response here
    }

    @Override
    public boolean isMappable(WebApplicationException e) {
```

(continues on next page)

(continued from previous page)

```
        return ExceptionUtils.indexOfThrowable(e, ViewRenderException.class) != -1;
    }
});
```

As an example, to return a 404 instead of a internal server error when one's mustache templates can't be found:

```
env.jersey().register(new ExtendedExceptionMapper<WebApplicationException>() {
    @Override
    public Response toResponse(WebApplicationException exception) {
        return Response.status(Response.Status.NOT_FOUND).build();
    }

    @Override
    public boolean isMappable(WebApplicationException e) {
        return Throwables.getRootCause(e).getClass() == MustacheNotFoundException.
↪class;
    }
});
```

2.9.2 Caching

By default templates are cached to improve loading time. If you want to disable it during the development mode, set the `cache` property to `false` in the view configuration.

```
views:
  .mustache:
    cache: false
```

2.9.3 Custom Error Pages

To get HTML error pages that fit in with your application, you can use a custom error view. Create a `View` that takes an `ErrorMessage` parameter in its constructor, and hook it up by registering a instance of `ErrorEntityWriter`.

```
env.jersey().register(new ErrorEntityWriter<ErrorMessage,View>(MediaType.TEXT_HTML_
↪TYPE, View.class) {
    @Override
    protected View getRepresentation(ErrorMessage errorMessage) {
        return new ErrorView(errorMessage);
    }
});
```

For validation error messages, you'll need to register another `ErrorEntityWriter` that handles `ValidationErrorMessage` objects.

```
env.jersey().register(new ErrorEntityWriter<ValidationErrorMessage,View>(MediaType.
↪TEXT_HTML_TYPE, View.class) {
    @Override
    protected View getRepresentation(ValidationErrorMessage message) {
        return new ValidationErrorView(message);
    }
});
```

2.10 Dropwizard & Scala

The `dropwizard-scala` module is now maintained and documented elsewhere.

The `metrics-scala` module is maintained here.

2.11 Testing Dropwizard

The `dropwizard-testing` module provides you with some handy classes for testing your representation classes and resource classes. It also provides a JUnit rule for full-stack testing of your entire app.

2.11.1 Testing Representations

While Jackson's JSON support is powerful and fairly easy-to-use, you shouldn't just rely on eyeballing your representation classes to ensure you're producing the API you think you are. By using the helper methods in *FixtureHelpers*, you can add unit tests for serializing and deserializing your representation classes to and from JSON.

Let's assume we have a `Person` class which your API uses as both a request entity (e.g., when writing via a `PUT` request) and a response entity (e.g., when reading via a `GET` request):

```
public class Person {
    private String name;
    private String email;

    private Person() {
        // Jackson deserialization
    }

    public Person(String name, String email) {
        this.name = name;
        this.email = email;
    }

    @JsonProperty
    public String getName() {
        return name;
    }

    @JsonProperty
    public void setName(String name) {
        this.name = name;
    }

    @JsonProperty
    public String getEmail() {
        return email;
    }

    @JsonProperty
    public void setEmail(String email) {
        this.email = email;
    }
```

(continues on next page)

(continued from previous page)

```
    // hashCode
    // equals
    // toString etc.
}
```

Fixtures

First, write out the exact JSON representation of a `Person` in the `src/test/resources/fixtures` directory of your Dropwizard project as `person.json`:

```
{
    "name": "Luther Blissett",
    "email": "lb@example.com"
}
```

Testing Serialization

Next, write a test for serializing a `Person` instance to JSON:

```
import static io.dropwizard.testing.FixtureHelpers.*;
import static org.assertj.core.api.Assertions.assertThat;
import io.dropwizard.jackson.Jackson;
import org.junit.Test;
import com.fasterxml.jackson.databind.ObjectMapper;

public class PersonTest {

    private static final ObjectMapper MAPPER = Jackson.newObjectMapper();

    @Test
    public void serializesToJSON() throws Exception {
        final Person person = new Person("Luther Blissett", "lb@example.com");

        final String expected = MAPPER.writeValueAsString(
                MAPPER.readValue(fixture("fixtures/person.json"), Person.class));

        assertThat(MAPPER.writeValueAsString(person)).isEqualTo(expected);
    }
}
```

This test uses AssertJ assertions and JUnit to test that when a `Person` instance is serialized via Jackson it matches the JSON in the fixture file. (The comparison is done on a normalized JSON string representation, so formatting doesn't affect the results.)

Testing Deserialization

Next, write a test for deserializing a `Person` instance from JSON:

```
import static io.dropwizard.testing.FixtureHelpers.*;
import static org.assertj.core.api.Assertions.assertThat;
import io.dropwizard.jackson.Jackson;
import org.junit.Test;
```

(continues on next page)

(continued from previous page)

```
import com.fasterxml.jackson.databind.ObjectMapper;

public class PersonTest {

    private static final ObjectMapper MAPPER = Jackson.newObjectMapper();

    @Test
    public void deserializesFromJSON() throws Exception {
        final Person person = new Person("Luther Blissett", "lb@example.com");
        assertThat(MAPPER.readValue(fixture("fixtures/person.json"), Person.class))
                .isEqualTo(person);
    }
}
```

This test uses AssertJ assertions and JUnit to test that when a `Person` instance is deserialized via Jackson from the specified JSON fixture it matches the given object.

2.11.2 Testing Resources

While many resource classes can be tested just by calling the methods on the class in a test, some resources lend themselves to a more full-stack approach. For these, use `ResourceTestRule`, which loads a given resource instance in an in-memory Jersey server:

```
import static org.assertj.core.api.Assertions.assertThat;
import static org.mockito.Mockito.*;

public class PersonResourceTest {

    private static final PeopleStore dao = mock(PeopleStore.class);

    @ClassRule
    public static final ResourceTestRule resources = ResourceTestRule.builder()
            .addResource(new PersonResource(dao))
            .build();

    private final Person person = new Person("blah", "blah@example.com");

    @Before
    public void setup() {
        when(dao.fetchPerson(eq("blah"))).thenReturn(person);
    }

    @After
    public void tearDown(){
        // we have to reset the mock after each test because of the
        // @ClassRule, or use a @Rule as mentioned below.
        reset(dao);
    }

    @Test
    public void testGetPerson() {
        assertThat(resources.target("/person/blah").request().get(Person.class))
                .isEqualTo(person);
        verify(dao).fetchPerson("blah");
    }
```

(continues on next page)

(continued from previous page)

```
}
```

Instantiate a `ResourceTestRule` using its `Builder` and add the various resource instances you want to test via `ResourceTestRule.Builder#addResource(Object)`. Use a `@ClassRule` annotation to have the rule wrap the entire test class or the `@Rule` annotation to have the rule wrap each test individually (make sure to remove static final modifier from `resources`).

In your tests, use `#target(String path)`, which initializes a request to talk to and test your instances.

This doesn't require opening a port, but `ResourceTestRule` tests will perform all the serialization, deserialization, and validation that happens inside of the HTTP process.

This also doesn't require a full integration test. In the above *example*, a mocked `PeopleStore` is passed to the `PersonResource` instance to isolate it from the database. Not only does this make the test much faster, but it allows your resource unit tests to test error conditions and edge cases much more easily.

Hint: You can trust `PeopleStore` works because you've got working unit tests for it, right?

Default Exception Mappers

By default, a `ResourceTestRule` will register all the default exception mappers (this behavior is new in 1.0). If `registerDefaultExceptionMappers` in the configuration yaml is planned to be set to `false`, `ResourceTestRule.Builder#setRegisterDefaultExceptionMappers(boolean)` will also need to be set to `false`. Then, all custom exception mappers will need to be registered on the builder, similarly to how they are registered in an `Application` class.

Test Containers

Note that the in-memory Jersey test container does not support all features, such as the `@Context` injection. A different test container can be used via `ResourceTestRule.Builder#setTestContainerFactory(TestContainerFactory)`.

For example, if you want to use the Grizzly HTTP server (which supports `@Context` injections) you need to add the dependency for the Jersey Test Framework providers to your Maven POM and set `GrizzlyWebTestContainerFactory` as `TestContainerFactory` in your test classes.

```
<dependency>
    <groupId>org.glassfish.jersey.test-framework.providers</groupId>
    <artifactId>jersey-test-framework-provider-grizzly2</artifactId>
    <version>${jersey.version}</version>
    <scope>test</scope>
    <exclusions>
        <exclusion>
            <groupId>javax.servlet</groupId>
            <artifactId>javax.servlet-api</artifactId>
        </exclusion>
        <exclusion>
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
        </exclusion>
    </exclusions>
</dependency>
```

```
public class ResourceTestWithGrizzly {
    @ClassRule
    public static final ResourceTestRule RULE = ResourceTestRule.builder()
        .setTestContainerFactory(new GrizzlyWebTestContainerFactory())
        .addResource(new ExampleResource())
        .build();

    @Test
    public void testResource() {
        assertThat(RULE.target("/example").request()
            .get(String.class))
            .isEqualTo("example");
    }
}
```

2.11.3 Testing Client Implementations

To avoid circular dependencies in your projects or to speed up test runs, you can test your HTTP client code by writing a JAX-RS resource as test double and let the `DropwizardClientRule` start and stop a simple Dropwizard application containing your test doubles.

```
public class CustomClientTest {
    @Path("/ping")
    public static class PingResource {
        @GET
        public String ping() {
            return "pong";
        }
    }

    @ClassRule
    public static final DropwizardClientRule dropwizard = new␣
↪DropwizardClientRule(new PingResource());

    @Test
    public void shouldPing() throws IOException {
        final URL url = new URL(dropwizard.baseUri() + "/ping");
        final String response = new BufferedReader(new InputStreamReader(url.
↪openStream())).readLine();
        assertEquals("pong", response);
    }
}
```

Hint: Of course you would use your HTTP client in the `@Test` method and not `java.net.URL#openStream()`.

The `DropwizardClientRule` takes care of:

- Creating a simple default configuration.
- Creating a simplistic application.
- Adding a dummy health check to the application to suppress the startup warning.
- Adding your JAX-RS resources (test doubles) to the Dropwizard application.

- Choosing a free random port number (important for running tests in parallel).
- Starting the Dropwizard application containing the test doubles.
- Stopping the Dropwizard application containing the test doubles.

2.11.4 Integration Testing

It can be useful to start up your entire application and hit it with real HTTP requests during testing. The `dropwizard-testing` module offers helper classes for your easily doing so. The optional `dropwizard-client` module offers more helpers, e.g. a custom JerseyClientBuilder, which is aware of your application's environment.

JUnit

Adding `DropwizardAppRule` to your JUnit test class will start the app prior to any tests running and stop it again when they've completed (roughly equivalent to having used `@BeforeClass` and `@AfterClass`). `DropwizardAppRule` also exposes the app's `Configuration`, `Environment` and the app object itself so that these can be queried by the tests.

If you don't want to use the `dropwizard-client` module or find it excessive for testing, you can get access to a Jersey HTTP client by calling the *client* method on the rule. The returned client is managed by the rule and can be reused across tests.

```
public class LoginAcceptanceTest {

    @ClassRule
    public static final DropwizardAppRule<TestConfiguration> RULE =
            new DropwizardAppRule<TestConfiguration>(MyApp.class, ResourceHelpers.
↪resourceFilePath("my-app-config.yaml"));

    @Test
    public void loginHandlerRedirectsAfterPost() {
        Client client = new JerseyClientBuilder(RULE.getEnvironment()).build("test␣
↪client");

        Response response = client.target(
                String.format("http://localhost:%d/login", RULE.getLocalPort()))
                .request()
                .post(Entity.json(loginForm()));

        assertThat(response.getStatus()).isEqualTo(302);
    }
}
```

Non-JUnit

By creating a DropwizardTestSupport instance in your test you can manually start and stop the app in your tests, you do this by calling its `before` and `after` methods. `DropwizardTestSupport` also exposes the app's `Configuration`, `Environment` and the app object itself so that these can be queried by the tests.

```
public class LoginAcceptanceTest {

    public static final DropwizardTestSupport<TestConfiguration> SUPPORT =
```

(continues on next page)

(continued from previous page)

```
            new DropwizardTestSupport<TestConfiguration>(MyApp.class,
                ResourceHelpers.resourceFilePath("my-app-config.yaml"),
                ConfigOverride.config("server.applicationConnectors[0].port", "0") //␣
↪Optional, if not using a separate testing-specific configuration file, use a␣
↪randomly selected port
            );

    @BeforeClass
    public void beforeClass() {
        SUPPORT.before();
    }

    @AfterClass
    public void afterClass() {
        SUPPORT.after();
    }

    @Test
    public void loginHandlerRedirectsAfterPost() {
        Client client = new JerseyClientBuilder(SUPPORT.getEnvironment()).build("test␣
↪client");

        Response response = client.target(
                 String.format("http://localhost:%d/login", SUPPORT.getLocalPort()))
                .request()
                .post(Entity.json(loginForm()));

        assertThat(response.getStatus()).isEqualTo(302);
    }
}
```

2.11.5 Testing Commands

Commands can and should be tested, as it's important to ensure arguments are interpreted correctly, and the output is as expected.

Below is a test for a command that adds the arguments as numbers and outputs the summation to the console. The test ensures that the result printed to the screen is correct by capturing standard out before the command is ran.

```
public class CommandTest {
    private final PrintStream originalOut = System.out;
    private final PrintStream originalErr = System.err;
    private final InputStream originalIn = System.in;

    private final ByteArrayOutputStream stdOut = new ByteArrayOutputStream();
    private final ByteArrayOutputStream stdErr = new ByteArrayOutputStream();
    private Cli cli;

    @Before
    public void setUp() throws Exception {
        // Setup necessary mock
        final JarLocation location = mock(JarLocation.class);
        when(location.getVersion()).thenReturn(Optional.of("1.0.0"));

        // Add commands you want to test
```

(continues on next page)

(continued from previous page)

```
        final Bootstrap<MyConfiguration> bootstrap = new Bootstrap<>(new␣
↪MyApplication());
        bootstrap.addCommand(new MyAddCommand());

        // Redirect stdout and stderr to our byte streams
        System.setOut(new PrintStream(stdOut));
        System.setErr(new PrintStream(stdErr));

        // Build what'll run the command and interpret arguments
        cli = new Cli(location, bootstrap, stdOut, stdErr);
    }

    @After
    public void teardown() {
        System.setOut(originalOut);
        System.setErr(originalErr);
        System.setIn(originalIn);
    }

    @Test
    public void myAddCanAddThreeNumbersCorrectly() {
        final boolean success = cli.run("add", "2", "3", "6");

        SoftAssertions softly = new SoftAssertions();
        softly.assertThat(success).as("Exit success").isTrue();

        // Assert that 2 + 3 + 6 outputs 11
        softly.assertThat(stdOut.toString()).as("stdout").isEqualTo("11");
        softly.assertThat(stdErr.toString()).as("stderr").isEmpty();
        softly.assertAll();
    }
}
```

2.11.6 Testing Database Interactions

In Dropwizard, the database access is managed via the `@UnitOfWork` annotation used on resource methods. In case you want to test database-layer code independently, a `DAOTestRule` is provided which setups a Hibernate `SessionFactory`.

```
public class DatabaseTest {

    @Rule
    public DAOTestRule database = DAOTestRule.newBuilder().addEntityClass(FooEntity.
↪class).build();

    private FooDAO fooDAO;

    @Before
    public void setUp() {
        fooDAO = new FooDAO(database.getSessionFactory());
    }

    @Test
    public createsFoo() {
        FooEntity fooEntity = new FooEntity("bar");
```

(continues on next page)

(continued from previous page)

```
        long id = database.inTransaction(() -> {
            return fooDAO.save(fooEntity);
        });

        assertThat(fooEntity.getId, notNullValue());
    }

    @Test
    public roundtripsFoo() {
        long id = database.inTransaction(() -> {
            return fooDAO.save(new FooEntity("baz"));
        });

        FooEntity fooEntity = fooDAO.get(id);

        assertThat(fooEntity.getFoo(), equalTo("baz"));
    }
}
```

The `DAOTestRule`

- Creates a simple default Hibernate configuration using an H2 in-memory database
- Provides a `SessionFactory` instance which can be passed to, e.g., a subclass of `AbstractDAO`
- Provides a function for executing database operations within a transaction

2.11.7 Testing Configurations

Configuration objects can be tested for correct deserialization and validation. Using the classes created in *polymorphic configurations* as an example, one can assert the expected widget is deserialized based on the `type` field.

```
public class WidgetFactoryTest {
    private final ObjectMapper objectMapper = Jackson.newObjectMapper();
    private final Validator validator = Validators.newValidator();
    private final YamlConfigurationFactory<WidgetFactory> factory =
            new YamlConfigurationFactory<>(WidgetFactory.class, validator,␣
↪objectMapper, "dw");

    @Test
    public void isDiscoverable() throws Exception {
        // Make sure the types we specified in META-INF gets picked up
        assertThat(new DiscoverableSubtypeResolver().getDiscoveredSubtypes())
                .contains(HammerFactory.class)
                .contains(ChiselFactory.class);
    }

    @Test
    public void testBuildAHammer() throws Exception {
        final File yml = new File(Resources.getResource("yaml/hammer.yml").toURI());
        final WidgetFactory wid = factory.build(yml);
        assertThat(wid).isInstanceOf(HammerFactory.class);
        assertThat(((HammerFactory) wid).createWidget().getWeight()).isEqualTo(10);
    }

    // test for the chisel factory
}
```

2.12 Dropwizard Example, Step by Step

The `dropwizard-example` module provides you with a working Dropwizard Example Application.

- Preconditions
  - Make sure you have Maven installed
  - Make sure `JAVA_HOME` points at JDK 8
  - Make sure you have `curl`
- Preparations to start the Dropwizard Example Application
  - Open a terminal / cmd
  - Navigate to the project folder of the Dropwizard Example Application
  - `mvn clean install`
  - `java -jar target/dropwizard-example-1.0.0.jar db migrate example.yml`
  - The statement above ran the liquibase migration in `/src/main/resources/migrations.xml`, creating the table schema
- Starting the Dropwizard Example Application
  - You can now start the Dropwizard Example Application by running `java -jar target/dropwizard-example-1.0.0.jar server example.yml`
  - Alternatively, you can run the Dropwizard Example Application in your IDE: `com.example.helloworld.HelloWorldApplication server example.yml`
- Working with the Dropwizard Example Application
  - Insert a new person: `curl -H "Content-Type: application/json" -d '{"fullName":"John Doe", "jobTitle" : "Chief Wizard" }' http://localhost:8080/people`
  - Retrieve that person: `curl http://localhost:8080/people/1`
  - View that person in a freemarker template: curl or open in a browser `http://localhost:8080/people/1/view_freemarker`
  - View that person in a mustache template: curl or open in a browser `http://localhost:8080/people/1/view_mustache`

2.13 Dropwizard Configuration Reference

2.13.1 Servers

Tweaking some of the options will require good understanding of how Jetty is working. See the Jetty architecture chapter for reference.

```
server:
  type: default
  maxThreads: 1024
```

All

Name	Default	Description
type	default	• default • simple
maxThreads	1024	The maximum number of threads the thread pool is allowed to grow. Jetty will throw `java.lang.IllegalStateException: Insufficient threads:` in case of too aggressive limit on the thread count.
minThreads	8	The minimum number of threads to keep alive in the thread pool. Note that each Jetty's connector consumes threads from the pool. See *HTTP connector* how the thread counts are calculated.
maxQueuedRequests	1024	The maximum number of requests to queue before blocking the acceptors.
idleThreadTimeout	1 minute	The amount of time a worker thread can be idle before being stopped.
nofileSoftLimit	(none)	The number of open file descriptors before a soft error is issued. Requires Jetty's `libsetuid.so` on `java.library.path`.
nofileHardLimit	(none)	The number of open file descriptors before a hard error is issued. Requires Jetty's `libsetuid.so` on `java.library.path`.
gid	(none)	The group ID to switch to once the connectors have started. Requires Jetty's `libsetuid.so` on `java.library.path`.
uid	(none)	The user ID to switch to once the connectors have started. Requires Jetty's `libsetuid.so` on `java.library.path`.
user	(none)	The username to switch to once the connectors have started. Requires Jetty's `libsetuid.so` on `java.library.path`.
group	(none)	The group to switch to once the connectors have started. Requires Jetty's `libsetuid.so` on `java.library.path`.
umask	(none)	The umask to switch to once the connectors have started. Requires Jetty's `libsetuid.so` on `java.library.path`.
startsAsRoot	(none)	Whether or not the Dropwizard application is started as a root user. Requires Jetty's `libsetuid.so` on `java.library.path`.
shutdownGracePeriod	30 seconds	The maximum time to wait for Jetty, and all Managed instances,

GZip

```
server:
  gzip:
    bufferSize: 8KiB
```

Name	Default	Description
enabled	true	If true, all requests with `gzip` or `deflate` in the `Accept-Encoding` header will have their response entities compressed and requests with `gzip` or `deflate` in the `Content-Encoding` header will have their request entities decompressed.
minimumEntitySize	256 bytes	All response entities under this size are not compressed.
bufferSize	8KiB	The size of the buffer to use when compressing.
excludedUserAgentPatterns	[]	The set of user agent patterns to exclude from compression.
compressedMimeTypes	Jetty's default	The list of mime types to compress. The default is all types apart the commonly known image, video, audio and compressed types.
includedMethods	Jetty's default	The list list of HTTP methods to compress. The default is to compress only GET responses.
deflateCompressionLevel	-1	The compression level used for ZLIB deflation(compression).
gzipCompatibleInflation	true	If true, then ZLIB inflation(decompression) will be performed in the GZIP-compatible mode.
syncFlush	false	The flush mode. Set to true if the application wishes to stream (e.g. SSE) the data, but this may hurt compression performance (as all pending output is flushed).

Request Log

The new request log uses the logback-access library for processing request logs, which allow to use an extended set of logging patterns. See the logback-access-pattern docs for the reference.

```
server:
  requestLog:
    appenders:
      - type: console
```

Name	Default	Description
appenders	console appender	The set of AppenderFactory appenders to which requests will be logged. See *logging* for more info.

Classic Request Log

The classic request log uses the logback-classic library for processing request logs. It produces logs only in the standard NCSA common log format, but allows to use an extended set of appenders.

```
server:
  requestLog:
    type: classic
    timeZone: UTC
    appenders:
      - type: console
```

Name	Default	Description
timeZone	UTC	The time zone to which request timestamps will be converted.
appenders	console appender	The set of AppenderFactory appenders to which requests will be logged. See *logging* for more info.

Server Push

Server push technology allows a server to send additional resources to a client along with the requested resource. It works only for HTTP/2 connections.

```
server:
  serverPush:
    enabled: true
    associatePeriod: '4 seconds'
    maxAssociations: 16
    refererHosts: ['dropwizard.io', 'dropwizard.github.io']
    refererPorts: [8444, 8445]
```

Name	Default	Description
enabled	false	If true, the filter will organize resources as primary resources (those referenced by the `Referer` header) and secondary resources (those that have the `Referer` header). Secondary resources that have been requested within a time window from the request of the primary resource will be associated with the it. The next time a client will request the primary resource, the server will send to the client the secondary resources along with the primary in a single response.
associatePeriod	4 seconds	The time window within which a request for a secondary resource will be associated to a primary resource..
maxAssociations	16	The maximum number of secondary resources that may be associated to a primary resource.
refererHosts	All hosts	The list of referrer hosts for which the server push technology is supported.
refererPorts	All ports	The list of referrer ports for which the server push technology is supported

Simple

Extends the attributes that are available to *all servers*

```
server:
  type: simple
  applicationContextPath: /application
  adminContextPath: /admin
  connector:
    type: http
    port: 8080
```

Name	Default	Description
connector	http connector	HttpConnectorFactory HTTP connector listening on port 8080. The ConnectorFactory connector which will handle both application and admin requests. TODO link to connector below.
applicationContextPath	/application	The context path of the application servlets, including Jersey.
adminContextPath	/admin	The context path of the admin servlets, including metrics and tasks.

Default

Extends the attributes that are available to *all servers*

```yaml
server:
  adminMinThreads: 1
  adminMaxThreads: 64
  adminContextPath: /
  applicationContextPath: /
  applicationConnectors:
    - type: http
      port: 8080
    - type: https
      port: 8443
      keyStorePath: example.keystore
      keyStorePassword: example
      validateCerts: false
  adminConnectors:
    - type: http
      port: 8081
    - type: https
      port: 8444
      keyStorePath: example.keystore
      keyStorePassword: example
      validateCerts: false
```

Name	Default	Description
applicationConnectors	An HTTP connector listening on port 8080.	A set of *connectors* which will handle application requests.
adminConnectors	An HTTP connector listening on port 8081.	An HTTP connector listening on port 8081. A set of *connectors* which will handle admin requests.
adminMinThreads	1	The minimum number of threads to use for admin requests.
adminMaxThreads	64	The maximum number of threads to use for admin requests.
adminContextPath	/	The context path of the admin servlets, including metrics and tasks.
applicationContextPath	/	The context path of the application servlets, including Jersey.

2.13.2 Connectors

HTTP

```yaml
# Extending from the default server configuration
server:
  applicationConnectors:
    - type: http
      port: 8080
      bindHost: 127.0.0.1 # only bind to loopback
      inheritChannel: false
      headerCacheSize: 512 bytes
      outputBufferSize: 32KiB
      maxRequestHeaderSize: 8KiB
      maxResponseHeaderSize: 8KiB
      inputBufferSize: 8KiB
      idleTimeout: 30 seconds
```

(continues on next page)

(continued from previous page)

```
      minBufferPoolSize: 64 bytes
      bufferPoolIncrement: 1KiB
      maxBufferPoolSize: 64KiB
      acceptorThreads: 1
      selectorThreads: 2
      acceptQueueSize: 1024
      reuseAddress: true
      soLingerTime: 345s
      useServerHeader: false
      useDateHeader: true
      useForwardedHeaders: true
      httpCompliance: RFC7230
```

Name	Default	Description
port	8080	The TCP/IP port on which to listen for incoming connections.
bindHost	(none)	The hostname to bind to.
inheritChannel	false	Whether this connector uses a channel inherited from the JVM. Use it with Server::Starter, to launch an instance of Jetty on demand.
headerCacheSize	512 bytes	The size of the header field cache.
outputBufferSize	32KiB	The size of the buffer into which response content is aggregated before being sent to the client. A larger buffer can improve performance by allowing a content producer to run without blocking, however larger buffers consume more memory and may induce some latency before a client starts processing the content.
maxRequestHeaderSize	8KiB	The maximum size of a request header. Larger headers will allow for more and/or larger cookies plus larger form content encoded in a URL. However, larger headers consume more memory and can make a server more vulnerable to denial of service attacks.
maxResponseHeaderSize	8KiB	The maximum size of a response header. Larger headers will allow for more and/or larger cookies and longer HTTP headers (eg for redirection). However, larger headers will also consume more memory.
inputBufferSize	8KiB	The size of the per-connection input buffer.
idleTimeout	30 seconds	The maximum idle time for a connection, which roughly translates to the java.net.Socket#setSoTimeout(int) call, although with NIO implementations other mechanisms may be used to implement the timeout. The max idle time is applied when waiting for a new message to be received on a connection or when waiting for a new message to be sent on a connection. This value is interpreted as the maximum time between some progress being made on the connection. So if a single byte is read or written, then the timeout is reset.
blockingTimeout	(none)	The timeout applied to blocking operations. This timeout is in addition to the *idleTimeout*, and applies to the total operation (as opposed to the idle timeout that applies to the time no data is being sent).
minBufferPoolSize	64 bytes	The minimum size of the buffer

HTTPS

Extends the attributes that are available to the *HTTP connector*

```
# Extending from the default server configuration
server:
  applicationConnectors:
    - type: https
      port: 8443
      ....
      keyStorePath: /path/to/file
      keyStorePassword: changeit
      keyStoreType: JKS
      keyStoreProvider:
      trustStorePath: /path/to/file
      trustStorePassword: changeit
      trustStoreType: JKS
      trustStoreProvider:
      keyManagerPassword: changeit
      needClientAuth: false
      wantClientAuth:
      certAlias: <alias>
      crlPath: /path/to/file
      enableCRLDP: false
      enableOCSP: false
      maxCertPathLength: (unlimited)
      ocspResponderUrl: (none)
      jceProvider: (none)
      validateCerts: false
      validatePeers: false
      supportedProtocols: (JVM default)
      excludedProtocols: [SSL, SSLv2, SSLv2Hello, SSLv3] # (Jetty's default)
      supportedCipherSuites: (JVM default)
      excludedCipherSuites: [.*_(MD5|SHA|SHA1)$] # (Jetty's default)
      allowRenegotiation: true
      endpointIdentificationAlgorithm: (none)
```

Name	Default	Description
keyStorePath	REQUIRED	The path to the Java key store which contains the host certificate and private key.
keyStorePassword	REQUIRED	The password used to access the key store.
keyStoreType	JKS	The type of key store (usually `JKS`, `PKCS12`, `JCEKS`, `Windows-MY`}, or `Windows-ROOT`).
keyStoreProvider	(none)	The JCE provider to use to access the key store.
trustStorePath	(none)	The path to the Java key store which contains the CA certificates used to establish trust.
trustStorePassword	(none)	The password used to access the trust store.
trustStoreType	JKS	The type of trust store (usually `JKS`, `PKCS12`, `JCEKS`, `Windows-MY`, or `Windows-ROOT`).
trustStoreProvider	(none)	The JCE provider to use to access the trust store.
keyManagerPassword	(none)	The password, if any, for the key manager.
needClientAuth	(none)	Whether or not client authentication is required.
wantClientAuth	(none)	Whether or not client authentication is requested.
certAlias	(none)	The alias of the certificate to use.
crlPath	(none)	The path to the file which contains the Certificate Revocation List.
enableCRLDP	false	Whether or not CRL Distribution Points (CRLDP) support is enabled.
enableOCSP	false	Whether or not On-Line Certificate Status Protocol (OCSP) support is enabled.
maxCertPathLength	(unlimited)	The maximum certification path length.
ocspResponderUrl	(none)	The location of the OCSP responder.
jceProvider	(none)	The name of the JCE provider to use for cryptographic support.
validateCerts	false	Whether or not to validate TLS certificates before starting. If enabled, Dropwizard will refuse to start with expired or otherwise invalid certificates. This option will cause unconditional failure in Dropwizard 1.x until a new validation mechanism can be implemented.
validate-	false	Whether or not to validate TLS peer certificates. This option will cause unconditional failure in Dropwizard 1.x until a new validation mechanism can be implemented.

HTTP/2 over TLS

HTTP/2 is a new protocol, intended as a successor of HTTP/1.1. It adds several important features like binary structure, stream multiplexing over a single connection, header compression, and server push. At the same time it remains semantically compatible with HTTP/1.1, which should make the upgrade process more seamless. Checkout HTTP/2 FAQ for the further information.

For an encrypted connection HTTP/2 uses ALPN protocol. It's a TLS extension, that allows a client to negotiate a protocol to use after the handshake is complete. If either side does not support ALPN, then the protocol will be ignored, and an HTTP/1.1 connection over TLS will be used instead.

For this connector to work with ALPN protocol you need to provide alpn-boot library to JVM's bootpath. The correct library version depends on a JVM version. Consult Jetty ALPN guide for the reference.

Note that your JVM also must provide `TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256` cipher. The specification states that HTTP/2 deployments must support it to avoid handshake failures. It's the single supported cipher in HTTP/2 connector by default. In case you want to support more strong ciphers, you should specify them in the `supportedCipherSuites` parameter along with `TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256`.

This connector extends the attributes that are available to the *HTTPS connector*

```
server:
  applicationConnectors:
    - type: h2
      port: 8445
      maxConcurrentStreams: 1024
      initialStreamRecvWindow: 65535
      keyStorePath: /path/to/file # required
      keyStorePassword: changeit
      trustStorePath: /path/to/file # required
      trustStorePassword: changeit
      supportedCipherSuites: # optional
        - TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384
        - TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256
```

Name	Default	Description
maxConcurrentStreams	1024	The maximum number of concurrently open streams allowed on a single HTTP/2 connection. Larger values increase parallelism, but cost a memory commitment.
initialStreamRecvWindow	65535	The initial flow control window size for a new stream. Larger values may allow greater throughput, but also risk head of line blocking if TCP/IP flow control is triggered.

HTTP/2 Plain Text

HTTP/2 promotes using encryption, but doesn't require it. However, most browsers stated that they will not support HTTP/2 without encryption. Currently no browser supports HTTP/2 unencrypted.

The connector should only be used in closed secured networks or during development. It expects from clients an HTTP/1.1 OPTIONS request with `Upgrade : h2c` header to indicate a wish to upgrade to HTTP/2, or a request with the HTTP/2 connection preface. If the client doesn't support HTTP/2, a plain HTTP/1.1 connections will be used instead.

This connector extends the attributes that are available to the *HTTP connector*

```
server:
  applicationConnectors:
```

(continues on next page)

(continued from previous page)

```
    - type: h2c
      port: 8446
      maxConcurrentStreams: 1024
      initialStreamRecvWindow: 65535
```

Name	Default	Description
maxConcurrentStreams	1024	The maximum number of concurrently open streams allowed on a single HTTP/2 connection. Larger values increase parallelism, but cost a memory commitment.
initialStreamRecvWindow	65535	The initial flow control window size for a new stream. Larger values may allow greater throughput, but also risk head of line blocking if TCP/IP flow control is triggered.

2.13.3 Logging

```
logging:
  level: INFO
  loggers:
    "io.dropwizard": INFO
    "org.hibernate.SQL":
      level: DEBUG
      additive: false
      appenders:
        - type: file
          currentLogFilename: /var/log/myapplication-sql.log
          archivedLogFilenamePattern: /var/log/myapplication-sql-%d.log.gz
          archivedFileCount: 5
  appenders:
    - type: console
```

Name	Default	Description
level	Level.INFO	Logback logging level.
additive	true	Logback additive setting.
loggers	(none)	Individual logger configuration (both forms are acceptable).
appenders	(none)	One of console, file or syslog.

Console

```
logging:
  level: INFO
  appenders:
    - type: console
      threshold: ALL
      queueSize: 512
      discardingThreshold: 0
      timeZone: UTC
      target: stdout
      logFormat: "%-5p [%d{ISO8601,UTC}] %c: %m%n%rEx"
      filterFactories:
        - type: URI
```

Name	Default	Description
type	REQUIRED	The appender type. Must be `console`.
threshold	ALL	The lowest level of events to print to the console.
queueSize	256	The maximum capacity of the blocking queue.
discardingThreshold	51	When the blocking queue has only the capacity mentioned in discardingThreshold remaining, it will drop events of level TRACE, DEBUG and INFO, keeping only events of level WARN and ERROR. If no discarding threshold is specified, then a default of queueSize / 5 is used. To keep all events, set discardingThreshold to 0.
timeZone	UTC	The time zone to which event timestamps will be converted. To use the system/default time zone, set it to `system`.
target	stdout	The name of the standard stream to which events will be written. Can be `stdout` or `stderr`.
logFormat	%-5p [%d{ISO8601,UTC}] %c: %m%n%rEx	The Logback pattern with which events will be formatted. See the Logback documentation for details.
filterFactories	(none)	The list of filters to apply to the appender, in order, after the threshold.
neverBlock	false	Prevent the wrapping asynchronous appender from blocking when its underlying queue is full. Set to true to disable blocking.

File

```yaml
logging:
  level: INFO
  appenders:
    - type: file
      currentLogFilename: /var/log/myapplication.log
      threshold: ALL
      queueSize: 512
      discardingThreshold: 0
      archive: true
      archivedLogFilenamePattern: /var/log/myapplication-%d.log
      archivedFileCount: 5
      timeZone: UTC
      logFormat: "%-5p [%d{ISO8601,UTC}] %c: %m%n%rEx"
      bufferSize: 8KB
      filterFactories:
        - type: URI
```

Name	Default	Description
type	REQUIRED	The appender type. Must be `file`.
currentLogFilename	REQUIRED	The filename where current events are logged.
threshold	ALL	The lowest level of events to write to the file.
queueSize	256	The maximum capacity of the blocking queue.
discardingThreshold	51	When the blocking queue has only the capacity mentioned in discardingThreshold remaining, it will drop events of level TRACE, DEBUG and INFO, keeping only events of level WARN and ERROR. If no discarding threshold is specified, then a default of queueSize / 5 is used. To keep all events, set discardingThreshold to 0.
archive	true	Whether or not to archive old events in separate files.
archivedLogFilenamePattern	(none)	Required if `archive` is `true`. The filename pattern for archived files. If `maxFileSize` is specified, rollover is size-based, and the pattern must contain `%i` for an integer index of the archived file. Otherwise rollover is date-based, and the pattern must contain `%d`, which is replaced with the date in `yyyy-MM-dd` form. If the pattern ends with `.gz` or `.zip`, files will be compressed as they are archived.
archivedFileCount	5	The number of archived files to keep. Must be greater than or equal to `0`. Zero is a special value signifying to keep infinite logs (use with caution)
maxFileSize	(unlimited)	The maximum size of the currently active file before a rollover is triggered. The value can be expressed in bytes, kilobytes, megabytes, gigabytes, and terabytes by appending B, K, MB, GB, or TB to the numeric value. Examples include 100MB, 1GB, 1TB. Sizes can also be spelled out, such as 100 megabytes, 1 gigabyte, 1 terabyte.
timeZone	UTC	The time zone to which event timestamps will be converted.
logFormat	%-5p [%d{ISO8601,UTC}] %c: %m%n%rEx	The Logback pattern with which events will be formatted. See the Logback documentation for details.
filterFactories	(none)	The list of filters to apply to the appender, in order, after the threshold.
neverBlock	false	Prevent the wrapping asynchronous appender from blocking when its underlying queue is full. Set to true to disable blocking.
bufferSize	8KB	The buffer size of the underlying FileAppender (setting added in logback 1.1.10). Increasing this from the default of 8KB to 256KB is reported to significantly reduce thread contention.

Syslog

```
logging:
  level: INFO
  appenders:
    - type: syslog
      host: localhost
```

(continues on next page)

(continued from previous page)

```
      port: 514
      facility: local0
      threshold: ALL
      stackTracePrefix: \t
      logFormat: "%-5p [%d{ISO8601,UTC}] %c: %m%n%rEx"
      filterFactories:
        - type: URI
```

Name	Default	Description
host	localhost	The hostname of the syslog server.
port	514	The port on which the syslog server is listening.
facility	local0	The syslog facility to use. Can be either `auth`, `authpriv`, `daemon`, `cron`, `ftp`, `lpr`, `kern`, `mail`, `news`, `syslog`, `user`, `uucp`, `local0`, `local1`, `local2`, `local3`, `local4`, `local5`, `local6`, or `local7`.
threshold	ALL	The lowest level of events to write to the file.
logFormat	%-5p [%d{ISO8601,UTC}] %c: %m%n%rEx	The Logback pattern with which events will be formatted. See the Logback documentation for details.
stackTracePrefix	t	The prefix to use when writing stack trace lines (these are sent to the syslog server separately from the main message)
filterFactories	(none)	The list of filters to apply to the appender, in order, after the threshold.
neverBlock	false	Prevent the wrapping asynchronous appender from blocking when its underlying queue is full. Set to true to disable blocking.

FilterFactories

```
logging:
  level: INFO
  appenders:
    - type: console
      filterFactories:
        - type: URI
```

Name	Default	Description
type	REQUIRED	The filter type.

2.13.4 Metrics

The metrics configuration has two fields; frequency and reporters.

```
metrics:
  frequency: 1 minute
  reporters:
    - type: <type>
```

Name	Default	Description
frequency	1 minute	The frequency to report metrics. Overridable per-reporter.
reporters	(none)	A list of reporters to report metrics.

All Reporters

The following options are available for all metrics reporters.

```
metrics:
  reporters:
    - type: <type>
      durationUnit: milliseconds
      rateUnit: seconds
      excludes: (none)
      includes: (all)
      excludesAttributes: (none)
      includesAttributes: (all)
      useRegexFilters: false
      frequency: 1 minute
```

Name	Default	Description
durationUnit	milliseconds	The unit to report durations as. Overrides per-metric duration units.
rateUnit	seconds	The unit to report rates as. Overrides per-metric rate units.
excludes	(none)	Metrics to exclude from reports, by name. When defined, matching metrics will not be reported.
includes	(all)	Metrics to include in reports, by name. When defined, only these metrics will be reported.
excludesAttributes	(none)	Metric attributes to exclude from reports, by name (e.g. `p98`, `m15_rate`, `stddev`). When defined, matching metrics attributes will not be reported.
includesAttributes	(all)	Metrics attributes to include in reports, by name (e.g. `p98`, `m15_rate`, `stddev`). When defined, only these attributes will be reported.
useRegexFilters	false	Indicates whether the values of the 'includes' and 'excludes' fields should be treated as regular expressions or not.
useSubstringMatching	false	Uses a substring matching strategy to determine whether a metric should be processed.
frequency	(none)	The frequency to report metrics. Overrides the default.

The inclusion and exclusion rules are defined as:

- If **includes** is empty, then all metrics are included;
- If **includes** is not empty, only metrics from this list are included;
- If **excludes** is empty, no metrics are excluded;
- If **excludes** is not empty, then exclusion rules take precedence over inclusion rules. Thus if a name matches the exclusion rules it will not be included in reports even if it also matches the inclusion rules.

When neither **useRegexFilters** nor **useSubstringMatching** are enabled, a default exact matching strategy will be used to determine whether a metric should be processed. In case both **useRegexFilters** and **useSubstringMatching** are set, **useRegexFilters** takes precedence over **useSubstringMatching**.

Formatted Reporters

These options are available only to "formatted" reporters and extend the options available to *all reporters*

```
metrics:
  reporters:
    - type: <type>
      locale: <system default>
```

Name	Default	Description
locale	System default	The Locale for formatting numbers, dates and times.

Console Reporter

Reports metrics periodically to the console.

Extends the attributes that are available to *formatted reporters*

```
metrics:
  reporters:
    - type: console
      timeZone: UTC
      output: stdout
```

Name	Default	Description
timeZone	UTC	The timezone to display dates/times for.
output	stdout	The stream to write to. One of `stdout` or `stderr`.

CSV Reporter

Reports metrics periodically to a CSV file.

Extends the attributes that are available to *formatted reporters*

```
metrics:
  reporters:
    - type: csv
      file: /path/to/file
```

Name	Default	Description
file	No default	The CSV file to write metrics to.

Ganglia Reporter

Reports metrics periodically to Ganglia.

Extends the attributes that are available to *all reporters*

Note: You will need to add `dropwizard-metrics-ganglia` to your POM.

```
metrics:
  reporters:
    - type: ganglia
      host: localhost
      port: 8649
      mode: unicast
      ttl: 1
      uuid: (none)
      spoof: localhost:8649
      tmax: 60
      dmax: 0
```

Name	Default	Description
host	localhost	The hostname (or group) of the Ganglia server(s) to report to.
port	8649	The port of the Ganglia server(s) to report to.
mode	unicast	The UDP addressing mode to announce the metrics with. One of `unicast` or `multicast`.
ttl	1	The time-to-live of the UDP packets for the announced metrics.
uuid	(none)	The UUID to tag announced metrics with.
spoof	(none)	The hostname and port to use instead of this nodes for the announced metrics. In the format `hostname:port`.
tmax	60	The tmax value to announce metrics with.
dmax	0	The dmax value to announce metrics with.

Graphite Reporter

Reports metrics periodically to Graphite.

Extends the attributes that are available to *all reporters*

Note: You will need to add `dropwizard-metrics-graphite` to your POM.

```
metrics:
  reporters:
    - type: graphite
      host: localhost
      port: 8080
      prefix: <prefix>
```

Name	Default	Description
host	localhost	The hostname of the Graphite server to report to.
port	8080	The port of the Graphite server to report to.
prefix	(none)	The prefix for Metric key names to report to Graphite.

SLF4J

Reports metrics periodically by logging via SLF4J.

Extends the attributes that are available to *all reporters*

See BaseReporterFactory and BaseFormattedReporterFactory for more options.

```
metrics:
  reporters:
    - type: log
      logger: metrics
      markerName: <marker name>
```

Name	Default	Description
logger	metrics	The name of the logger to write metrics to.
markerName	(none)	The name of the marker to mark logged metrics with.

2.13.5 Clients

HttpClient

See HttpClientConfiguration for more options.

```
httpClient:
  timeout: 500ms
  connectionTimeout: 500ms
  timeToLive: 1h
  cookiesEnabled: false
  maxConnections: 1024
  maxConnectionsPerRoute: 1024
  keepAlive: 0ms
  retries: 0
  userAgent: <application name> (<client name>)
```

Name	Default	Description
timeout	500 milliseconds	The maximum idle time for a connection, once established.
connectionTimeout	500 milliseconds	The maximum time to wait for a connection to open.
connectionRequestTimeout	500 milliseconds	The maximum time to wait for a connection to be returned from the connection pool.
timeToLive	1 hour	The maximum time a pooled connection can stay idle (not leased to any thread) before it is shut down.
cookiesEnabled	false	Whether or not to enable cookies.
maxConnections	1024	The maximum number of concurrent open connections.
maxConnectionsPerRoute	1024	The maximum number of concurrent open connections per route.
keepAlive	0 milliseconds	The maximum time a connection will be kept alive before it is reconnected. If set to 0, connections will be immediately closed after every request/response.
retries	0	The number of times to retry failed requests. Requests are only retried if they throw an exception other than `InterruptedIOException`, `UnknownHostException`, `ConnectException`, or `SSLException`.
userAgent	`applicationName` (`clientName`)	The User-Agent to send with requests.
validateAfterInactivityPeriod	0 milliseconds	The maximum time before a persistent connection is checked to remain active. If set to 0, no inactivity check will be performed.

Proxy

```
httpClient:
  proxy:
    host: 192.168.52.11
    port: 8080
    scheme : http
    auth:
      username: secret
      password: stuff
      authScheme: NTLM
      realm: realm
      hostname: host
      domain: WINDOWSDOMAIN
      credentialType: NT
    nonProxyHosts:
      - localhost
      - '192.168.52.*'
      - '*.example.com'
```

Name	Default	Description
host	REQUIRED	The proxy server host name or ip address.
port	(scheme default)	The proxy server port. If the port is not set then the scheme default port is used.
scheme	http	The proxy server URI scheme. HTTP and HTTPS schemas are permitted. By default HTTP scheme is used.
auth	(none)	The proxy server `Basic` or `NTLM` authentication schemes. If they are not set then no credentials will be passed to the server.
username	REQUIRED	The username used to connect to the server.
password	REQUIRED	The password used to connect to the server.
authScheme	Basic	The authentication scheme used by the. Allowed options are: `Basic`, `NTLM`
realm	(none)	The realm, used for NTLM authentication.
hostname	(none)	The hostname of the windows workstation, used for NTLM authentication.
domain	(none)	The Windows Domain, used for NTLM authentication.
credentialType	(none)	The Apache HTTP Client Credentials imeplementation used for proxy authentication. Allowed options are: `UsernamePassword` or `NT`
nonProxyHosts	(none)	List of patterns of hosts that should be reached without proxy. The patterns may contain symbol '*' as a wildcard. If a host matches one of the patterns it will be reached through a direct connection.

TLS

```
httpClient:
  tls:
    protocol: TLSv1.2
    verifyHostname: true
    keyStorePath: /path/to/file
    keyStorePassword: changeit
    keyStoreType: JKS
    trustStorePath: /path/to/file
    trustStorePassword: changeit
    trustStoreType: JKS
    trustSelfSignedCertificates: false
    supportedProtocols: TLSv1.1,TLSv1.2
    supportedCipherSuites: TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA256
    certAlias: alias-of-specific-cert
```

Name	Default	Description
protocol	TLSv1.2	The default protocol the client will attempt to use during the SSL Handshake. See here for more information.
verifyHostname	true	Whether to verify the hostname of the server against the hostname presented in the server certificate.
keyStorePath	(none)	The path to the Java key store which contains the client certificate and private key.
keyStorePassword	(none)	The password used to access the key store.
keyStoreType	JKS	The type of key store (usually `JKS`, `PKCS12`, `JCEKS`, `Windows-MY`, or `Windows-ROOT`).
trustStorePath	(none)	The path to the Java key store which contains the CA certificates used to establish trust.
trustStorePassword	(none)	The password used to access the trust store.
trustStoreType	JKS	The type of trust store (usually `JKS`, `PKCS12`, `JCEKS`, `Windows-MY`, or `Windows-ROOT`).
trustSelfSignedCertificates	false	If true, will trust all self-signed certificates regardless of trustStore settings. If false, trust decisions will be handled by the supplied trustStore.
supportedProtocols	(none)	A list of protocols (e.g., `SSLv3`, `TLSv1`) which are supported. All other protocols will be refused.
supportedCipherSuites	(none)	A list of cipher suites (e.g., `TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA256`) which are supported. All other cipher suites will be refused.
certAlias	(none)	The alias of a specific client certificate to present when authenticating. Use this when the specified keystore has multiple certificates to force use of a non-default certficate.

JerseyClient

Extends the attributes that are available to *http clients*

See JerseyClientConfiguration and HttpClientConfiguration for more options.

```
jerseyClient:
  minThreads: 1
  maxThreads: 128
  workQueueSize: 8
  gzipEnabled: true
  gzipEnabledForRequests: true
  chunkedEncodingEnabled: true
```

Name	Default	Description
minThreads	8	The minimum number of threads in the pool used for asynchronous requests.
maxThreads	128	The maximum number of threads in the pool used for asynchronous requests. If asynchronous requests made by jersey client while serving requests, the number must be set according to the *maxThread* setting of the *server*. Otherwise some requests made to dropwizard on heavy load may fail due to congestion on the jersey client's thread pool.
workQueueSize	8	The size of the work queue of the pool used for asynchronous requests. Additional threads will be spawn only if the queue is reached its maximum size.
gzipEnabled	true	Adds an Accept-Encoding: gzip header to all requests, and enables automatic gzip decoding of responses.
gzipEnabledForRequests	true	Adds a Content-Encoding: gzip header to all requests, and enables automatic gzip encoding of requests.
chunkedEncodingEnabled	true	Enables the use of chunked encoding for requests.

2.13.6 Database

```
database:
  driverClass : org.postgresql.Driver
  url: 'jdbc:postgresql://db.example.com/db-prod'
  user: pg-user
  password: iAMs00perSecrEET
```

Name	Default	Description
driverClass	REQUIRED	The full name of the JDBC driver class.
url	REQUIRED	The URL of the server.
user	none	The username used to connect to the server.
password	none	The password used to connect to the server.
removeAbandoned	false	Remove abandoned connections if they exceed removeAbandonedTimeout.
removeAbandonedTimeout	60 seconds	The time before a database connection can be considered abandoned.
abandonWhenPercentageFull	0	Connections that have been abandoned (timed out) won't get closed and repo
alternateUsernamesAllowed	false	Set to true if the call getConnection(username,password) is allowed. This is
commitOnReturn	false	Set to true if you want the connection pool to commit any pending transactio
rollbackOnReturn	false	Set to true if you want the connection pool to rollback any pending transacti
autoCommitByDefault	JDBC driver's default	The default auto-commit state of the connections.
readOnlyByDefault	JDBC driver's default	The default read-only state of the connections.
properties	none	Any additional JDBC driver parameters.
defaultCatalog	none	The default catalog to use for the connections.
defaultTransactionIsolation	JDBC driver's default	The default transaction isolation to use for the connections. Can be one of no
useFairQueue	true	If true, calls to getConnection are handled in a FIFO manner.
initialSize	10	The initial size of the connection pool.
minSize	10	The minimum size of the connection pool.
maxSize	100	The maximum size of the connection pool.
initializationQuery	none	A custom query to be run when a connection is first created.

Name	Default	Description
logAbandonedConnections	false	If true, logs stack traces of abandoned connections.
logValidationErrors	false	If true, logs errors when connections fail validation.
maxConnectionAge	none	If set, connections which have been open for longer than maxConnectionAg
maxWaitForConnection	30 seconds	If a request for a connection is blocked for longer than this period, an except
minIdleTime	1 minute	The minimum amount of time an connection must sit idle in the pool before
validationQuery	SELECT 1	The SQL query that will be used to validate connections from this pool befo
validationQueryTimeout	none	The timeout before a connection validation queries fail.
checkConnectionWhileIdle	true	Set to true if query validation should take place while the connection is idle.
checkConnectionOnBorrow	false	Whether or not connections will be validated before being borrowed from th
checkConnectionOnConnect	false	Whether or not connections will be validated before being added to the pool.
checkConnectionOnReturn	false	Whether or not connections will be validated after being returned to the pool
autoCommentsEnabled	true	Whether or not ORMs should automatically add comments.
evictionInterval	5 seconds	The amount of time to sleep between runs of the idle connection validation,
validationInterval	30 seconds	To avoid excess validation, only run validation once every interval.
validatorClassName	none	Name of a class of a custom validator implementation, which will be used fo
jdbcInterceptors	none	A semicolon separated list of JDBC interceptor classnames.

2.13.7 Polymorphic configuration

The `dropwizard-configuration` module provides you with a polymorphic configuration mechanism, meaning that a particular section of your configuration file can be implemented using one or more configuration classes.

To use this capability for your own configuration classes, create a top-level configuration interface or class that implements `Discoverable` and add the name of that class to `META-INF/services/io.dropwizard.jackson.Discoverable`. Make sure to use Jackson polymorphic deserialization annotations appropriately.

```
@JsonTypeInfo(use = Id.NAME, include = As.PROPERTY, property = "type")
interface WidgetFactory extends Discoverable {
    Widget createWidget();
}
```

Then create subtypes of the top-level type corresponding to each alternative, and add their names to `META-INF/services/WidgetFactory`.

```
@JsonTypeName("hammer")
public class HammerFactory implements WidgetFactory {
    @JsonProperty
    private int weight = 10;

    @Override
    public Hammer createWidget() {
        return new Hammer(weight);
    }
}

@JsonTypeName("chisel")
public class ChiselFactory implements WidgetFactory {
    @JsonProperty
    private float radius = 1;
```

(continues on next page)

(continued from previous page)

```
    @Override
    public Chisel createWidget() {
        return new Chisel(radius);
    }
}
```

Now you can use `WidgetFactory` objects in your application's configuration.

```
public class MyConfiguration extends Configuration {
    @JsonProperty
    @NotNull
    @Valid
    private List<WidgetFactory> widgets;
}
```

```
widgets:
  - type: hammer
    weight: 20
  - type: chisel
    radius: 0.4
```

See *testing configurations* for details on ensuring the configuration will be deserialized correctly.

2.14 Dropwizard Internals

You already read through the whole Dropwizard documentation? Congrats! Then you are ready to have a look into some nitty-gritty details of Dropwizard.

2.14.1 Startup Sequence

`Application<T extends Configuration>` is the "Main" class of a dropwizard Application.

`application.run(args)` is the first method to be called on startup - Here is a simplified code snippet of its implementation:

```
public void run(String... arguments) throws Exception {

  final Bootstrap<T> bootstrap = new Bootstrap<>(this);
  bootstrap.addCommand(new ServerCommand<>(this));
  bootstrap.addCommand(new CheckCommand<>(this));

  initialize(bootstrap); // -- implemented by you; it should call:
        // 1. add bundles (typically being used)
        // 2. add commands (if any)

  // Should be called after `initialize` to give an opportunity to set a custom
↪metric registry
  bootstrap.registerMetrics(); // start tracking some default jvm params...

  // for each cmd, configure parser w/ cmd
  final Cli cli = new Cli(new JarLocation(getClass()), bootstrap, our, err)
  cli.run(arguments);
}
```

`Bootstrap` is the the pre-start (temp) application environment, containing everything required to bootstrap a Dropwizard command. Here is a simplified code snippet to illustrate its structure:

```
Bootstrap(application: Application<T>) {
  this.application = application;
  this.objectMapper = Jackson.newObjectMapper();
  this.bundles = new ArrayList<>();
  this.configuredBundles = new ArrayList<>();
  this.commands = new ArrayList<>();
  this.validatorFactory = Validators.newValidatorFactory();
  this.metricRegistry = new MetricRegistry();
  this.classLoader = Thread.currentThread().getContextClassLoader();
  this.onfigurationFactory = new DefaultConfigurationFactoryFactory<>();
  this.healthCheckRegistry = new HealthCheckRegistry();
}
```

`Environment` is a longer-lived object, holding Dropwizard's Environment (not env. Such as dev or prod). It holds a similar, but somewhat different set of properties than the Bootsrap object - here is a simplified code snippet to illustrate that:

```
Environment (...) {
  // from bootstrap
  this.objectMapper = ...
  this.classLoader = ...
  this.metricRegistry = ...
  this.healthCheckRegistry = ...
  this.validator = bootstrap.getValidatorFactory().getValidator()

  // extra:
  this.bundles = new ArrayList<>();
  this.configuredBundles = new ArrayList<>();

  // sub-environments:
  this.servletEnvironment = ... // -- exposed via the servlets() method
  this.jerseyEnvironment = ... // -- exposed via the jersey() method
  this.adminEnvironment = ... // -- exposed via the admin() method

}
```

A Dropwizard `Bundle` is a reusable group of functionality (sometimes provided by the Dropwizard project itself), used to define blocks of an application's behavior. For example, `AssetBundle` from the dropwizard-assets module provides a simple way to serve static assets from your application's src/main/resources/assets directory as files available from /assets/* (or any other path) in your application.

A `ConfiguredBundle` is a bundle that require a configuration provided by the `Configuration` object (implementing a relevant interface)

Properties such as database connection details should not be stored on the Environment; that is what your Configuration .yml file is for. Each logical environment (dev/test/staging/prod) - would have its own Configuration .yml - reflecting the differences between different "server environments".

Commands

`Command` objects are basic actions, which Dropwizard runs based on the arguments provided on the command line. The built-in `server` command, for example, spins up an HTTP server and runs your application. Each Command subclass has a name and a set of command line options which Dropwizard will use to parse the given command line arguments. The `check` command parses and validates the application's configuration.

If you will check again the first code snippet in this document - you will see creating these two commands, is the first step in the bootstrapping process.

Another important command is `db` - allowing executing various db actions, see *Dropwizard Migrations*

Similar to `ConfiguredBundle`, some commands require access to configuration parameters and should extend the `ConfiguredCommand` class, using your application's `Configuration` class as its type parameter.

The CLI class

Let us begin with a simplified version of the constructor:

```
public Cli(location : JarLocation, bootstrap : Bootstrap<?>,
                   stdOut: OutputStream, stdErr: OutputStream) {
  this.stdout = stdOut; this.stdErr = stdErr;
  this.commands = new TreeMap<>();
  this.parser = buildParser(location);
  this.bootstrap = bootstrap;
  for (command in bootstrap.commands) {
        addCommand(command)
  }
}
```

Cli is the command-line runner for Dropwizard application. Initializing, and then running it - is the last step of the Bootstrapping process.

Run would just handle commandline args (–help, –version) or runs the configured commands. E.g. - When running the `server` command:

```
java -jar target/hello-world-0.0.1-SNAPSHOT.jar server hello-world.yml
```

Just note the two basic commands are built of a parent, and a sub-class:

```
class CheckCommand<T extends Configuration> extends ConfiguredCommand<T>
class ServerCommand<T extends Configuration> extends EnvironmentCommand<T>
```

The order of operations is therefore:

1. Parse cmdline args, determine subcommand.
2. Run `ConfiguredCommand`, which get a parameter with the location of a YAML configuration file - parses and validates it.
3. `CheckCommand.run()` runs next, and does almost nothing: it logs `"Configuration is OK"`
4. Run `EnvironmentCommand`:

a) Create `Environment`

b) Calls `bootstrap.run(cfg, env)` - run bundles with config. & env.

c) Bundles run in FIFO order.

d) Calls `application.run(cfg, env)` – implemented by you

6. Now, `ServerCommand.run()` runs

a) Calls `serverFactory.build(environment)` - to configure Jetty and Jersey, with all relevant Dropwizard modules.

b) Starts Jetty.

2.14.2 Jetty Lifecycle

If you have a component of your app that needs to know when Jetty is going to start, you can implement Managed as described in the dropwizard docs.

If you have a component that needs to be signaled that Jetty has started (this happens after all Managed objects' start() methods are called), you can register with the env's lifecycle like:

```
env.lifecycle().addServerLifecycleListener(new ServerLifecycleListener() {
    @Override
    public void serverStarted(Server server) {
              /// ... do things here ....
    }
});
```

CHAPTER 3

Javadoc

- dropwizard-auth
- dropwizard-client
- dropwizard-configuration
- dropwizard-core
- dropwizard-db
- dropwizard-forms
- dropwizard-hibernate
- dropwizard-jackson
- dropwizard-jdbi
- dropwizard-jersey
- dropwizard-jetty
- dropwizard-lifecycle
- dropwizard-logging
- dropwizard-metrics
- dropwizard-metrics-ganglia
- dropwizard-metrics-graphite
- dropwizard-migrations
- dropwizard-servlets
- dropwizard-spdy
- dropwizard-testing
- dropwizard-util
- dropwizard-validation

- dropwizard-views
- dropwizard-views-freemarker
- dropwizard-views-mustache

CHAPTER 4

About Dropwizard

4.1 Contributors

Dropwizard wouldn't exist without the hard work contributed by numerous individuals.

Many, many thanks to:

- Aaron Ingram
- Adam Jordens
- Adam Marcus
- Aidan
- akumlehn
- Alex Ausch
- Alex Butler
- Alex Heneveld
- Alex Katlein
- Alice Chen
- Al Scott
- Anand Mohan
- Anders Hedström
- Andreas Petersson
- Andreas Stührk
- Andrei Savu
- Andrew Clay Shafer
- anikiej

- Antanas Končius
- Anthony Milbourne
- Anthony Wat
- Arien Kock
- Armando Singer
- Artem Prigoda
- arunh
- Athou
- Bartek Szymański
- Basil James Whitehouse III
- Ben Bader
- Benjamin Bentmann
- Ben Ripkens
- Ben Smith
- Bo Gotthardt
- Børge Nese
- Boyd Meier
- Brandon Beck
- Brett Hoerner
- Brian McCallister
- Brian O'Neill
- Bruce Ritchie
- Burak Dede
- BusComp
- Cagatay Kavukcuoglu
- Camille Fournier
- Carl Lerche
- Carlo Barbara
- Carter Kozak
- Cemalettin Koc
- Chad Selph
- Charlie Greenbacker
- Charlie La Mothe
- cheddar
- Chen Wang
- Chris Micali

- Chris Pimlott
- Chris Tierney
- Christoffer Eide
- Christopher Currie
- Christopher Gray
- Christopher Holmes
- Christoph Kutzinski
- Coda Hale
- Collin Van Dyck
- Csaba Palfi
- Dale Wijnand
- Damian Pawlowski
- Dan Everton
- Dang Nguyen Anh Khoa
- Daniel Correia
- Daniel Temme
- dan mcweeney
- Darren Yin
- David Illsley
- David Martin
- David Morgantini
- David Stendardi
- Dennis Hoersch
- Denny Abraham Cheriyan
- Derek Cicerone
- Devin Breen
- Devin Smith
- Dheerendra Rathor
- Dietrich Featherston
- Dimitris Zavaliadis
- Dmitry Minkovsky
- Dmitry Ustalov
- dom farr
- Dominic Tootell
- Dominik Wagenknecht
- douzzi

- Drew Stephens
- Dylan Scott
- eepstein
- Ellis Pritchard
- Emeka Mosanya
- Erik van Oosten
- Evan Jones
- Evan Meagher
- Fábio Franco Uechi
- Felix
- Flemming Frandsen
- Florian Hirsch
- florinn
- Fredrik Sundberg
- Frode Nerbråten
- Gabe Henkes
- Gary Dusbabek
- Glenn McAllister
- Graham O'Regan
- Greg Bowyer
- Gunnar Ahlberg
- Håkan Jonson
- Henrik Stråth
- Hrvoje Slaviček
- Ian White
- Ilias Bartolini
- ipropper
- Jacek Jackowiak
- Jake Swenson
- James Morris
- James Ward
- Jamie Furnaghan
- Jan Galinski
- Jan Olaf Krems
- Jan-Terje Sørensen
- Jared Stehler

- Jason Clawson
- Jason Dunkelberger
- Jason Toffaletti
- Javier Campanini
- Jeff Klukas
- Jerry-Carter
- Jesse Hodges
- Jilles Oldenbeuving
- Jochen Schalanda
- Joe Lauer
- Joe Schmetzer
- Johan Wirde (@jwirde)
- Jonathan Haber
- Jonathan Halterman
- Jonathan Monette
- Jonathan Ruckwood
- Jonathan Welzel
- Jon Radon
- Jordan Zimmerman
- Joshua Spiewak
- Julien
- Justin Miller
- Justin Plock
- Kashyap Paidimarri
- Kerry Kimbrough
- Kilemensi
- Kirill Vlasov
- Konstantin Yegupov
- Kristian Klette
- Krzysztof Mejka
- kschjeld
- LeekAnarchism
- lehcim
- Lucas
- Lunfu Zhong
- mabuthraa

- maffe
- Malte S. Stretz
- Manabu Matsuzaki
- Marcin Biegan
- Marcus Höjvall
- Marius Volkhart
- markez92
- Mark Reddy
- Mark Wolfe
- Mårten Gustafson
- Martin W. Kirst
- Matt Brown
- Matt Carrier
- Matthew Clarke
- Matt Hurne
- Matt Nelson
- Matt Veitas
- Matt Whipple
- Maximilien Marie
- Max Wenzin
- Michael Chaten
- Michael Fairley
- Michael McCarthy
- Michal Rutkowski
- MikaelAmborn
- Mike Miller
- mnrasul
- Moritz Kammerer
- natnan
- Nick Babcock
- Nick Telford
- Nikhil Bafna
- Nisarg Shah
- Oddmar Sandvik
- Oliver B. Fischer
- Olivier Abdesselam

- Olivier Grégoire
- Ori Schwartz
- Otto Jongerius
- Owen Jacobson
- pandaadb
- Patrick Stegmann
- Patryk Najda
- Paul Samsotha
- Paul Tomlin
- Philip K. Warren
- Philip Potter
- Punyashloka Biswal
- Qinfeng Chen
- Quoc-Viet Nguyen
- Rachel Newstead
- rayokota
- Rémi Alvergnat
- Richard Kettelerij
- Richard Nyström
- Robert Barbey
- Rüdiger zu Dohna
- Ryan Berdeen
- Ryan Kennedy
- Ryan Warren
- saadmufti
- Sam Perman
- Sam Quigley
- Scott D.
- Sean Scanlon
- shartte
- Simon Collins
- smolloy
- Sourav Mitra
- Stan Svec
- Stephen Huenneke
- Steve Agalloco

- Steve Hill
- Stevo Slavić
- Stuart Gunter
- Szymon Pacanowski
- Tatu Saloranta
- Ted Nyman
- Thiago Moretto
- Thomas Darimont
- Tim Bart
- Tim Bartley
- Tom Akehurst
- Tom Crayford
- Tom Lee
- Tom Morris
- Tom Shen
- Tony Gaetani
- Trevor Mack
- Tristan Burch
- twilson-palantir
- Tyrone Cutajar
- Vadim Spivak
- Varun Loiwal
- Vasyl Vavrychuk
- Victor Noël
- Vitor Reis
- Vladimir Ladynev
- Vojtěch Vondra
- vzx
- Wank Sinatra
- William Herbert
- Xavier Shay
- Xiaodong Xie
- Yiwei Gao
- Yun Zhi Lin
- Yurii Savka

4.2 Sponsors

Dropwizard is generously supported by some companies with licenses and free accounts for their products.

4.2.1 JetBrains

JetBrains supports our open source project by sponsoring some All Products Packs within their Free Open Source License program.

4.3 Frequently Asked Questions

What's a Dropwizard? A character in a K.C. Green web comic.

How is Dropwizard licensed? It's licensed under the Apache License v2.

How can I commit to Dropwizard? Go to the GitHub project, fork it, and submit a pull request. We prefer small, single-purpose pull requests over large, multi-purpose ones. We reserve the right to turn down any proposed changes, but in general we're delighted when people want to make our projects better!

4.4 Release Notes

4.4.1 v1.2.8: Jun 25, 2018

- Upgrade to Jetty 9.4.11.v20180605 to address various security issues

4.4.2 v1.1.8: Jun 25, 2018

- Upgrade to Jetty 9.4.11.v20180605 to address various security issues

4.4.3 v1.2.7: Jun 14, 2018

- Upgrade to Jackson 2.9.6 to fix CVE-2018-12022 and CVE-2018-12023 (#2392, #2393)

4.4.4 v1.2.6: May 11, 2018

- Upgrade Jetty to 9.4.10.v20180503 #2346
- Add possibility to disable logging bootstrap for ResourceTestRule #2333

4.4.5 v1.2.5: Apr 4, 2018

- Upgrade to Jackson 2.9.5 (CVE-2018-7489)

4.4.6 v1.2.4: Feb 23, 2018

- Upgrade Jackson to 2.9.4 in 1.2.* to address a CVE #2269

4.4.7 v1.1.7: Feb 23, 2018

- Upgrade to Jackson 2.8.11 to address CVE #2270

4.4.8 v1.2.3: Jan 24, 2018

- Enable auto escaping of strings in Freemarker templates #2251

4.4.9 v1.2.2: Nov 27, 2017

- Don't shut down asynchronous executor in Jersey #2221
- Add possibility to possibility to extend *DropwizardApacheConnector* #2220

4.4.10 v1.2.1: Nov 22, 2017

- Correctly set up SO_LINGER for the HTTP connector #2176
- Support fromString in FuzzyEnumParamConverter #2161
- Upgrade to Hibernate 5.2.12.Final to address HHH-11996, #2206
- Upgrade to Freemaker 2.3.27-incubating

4.4.11 v1.1.6: Nov 2, 2017

- Support fromString in FuzzyEnumParamConverter #2161

4.4.12 v1.1.5: Oct 17, 2017

- Correctly set up SO_LINGER for the HTTP connector #2176

4.4.13 v1.2.0: Oct 6 2017

Complete changelog on GitHub

- Support configuring FileAppender#bufferSize #1951
- Improve error handling of *@FormParam* resources #1982
- Add JDBC interceptors through configuration #2030
- Support Dropwizard applications without logback #1900
- Replace deprecated SizeAndTimeBasedFNATP with SizeAndTimeBasedRollingPolicy #2010
- Decrease allowable tomcat jdbc validation interval to 50ms #2051
- Add support for setting several cipher suites for HTTP/2 #2119
- Remove Dropwizard's Jackson dependency on Logback #2112
- Handle badly formed "Accept-Language" headers #2103
- Use LoadingCache in CachingAuthorizer #2096
- Client NTLM Authentication #2091
- Add optional Jersey filters #1948
- Upgrade to Apache commons-lang3 3.6
- Upgrade to AssertJ 3.8.0
- Upgrade to classmate 1.3.4
- Upgrade to Guava 23.1
- Upgrade to H2 1.4.196
- Upgrade to Hibernate 5.2.11.Final
- Upgrade to Hibernate Validator 5.4.1.Final
- Upgrade to HSQLDB 2.4.0
- Upgrade to Jackson 2.9.1
- Upgrade to Jetty 9.4.7.v20170914
- Upgrade to JMH 1.19
- Upgrade to Joda-Time 2.9.9
- Upgrade to Logback 1.2.3
- Upgrade to Metrics 3.2.5
- Upgrade to Mockito 2.10.0
- Upgrade to Mustache.java 0.9.5
- Upgrade to Objenesis 2.6
- Upgrade to SLF4J 1.7.25
- Upgrade to tomcat-jdbc 8.5.23

4.4.14 v1.1.4: Aug 24 2017

Complete changelog on GitHub

- Upgrade to Jackson 2.8.10 #2120

4.4.15 v1.1.3: Jul 31 2017

Complete changelog on GitHub

- Handle badly formed 'Accept-Language' headers #2097
- Upgrade to Jetty 9.4.6.v20170531 to address CVE-2017-9735 #2113

4.4.16 v1.1.2 June 27 2017

Complete changelog on GitHub

- Updated Jackson to 2.8.9. Fixes a security vulnerability with default typing #2086
- Use the correct *JsonFactory* in JSON configuration parsing #2046
- Support of extending of *DBIFactory* #2067
- Add time zone to Java 8 datetime mappers #2069

4.4.17 v1.0.8 June 27 2017

Complete changelog on GitHub

- Updated Jackson to 2.7.9.1. Fixes a security vulnerability with default typing #2087

4.4.18 v1.1.1 May 19 2017

Complete changelog on GitHub

- Set the console logging context after a reset #1973
- Set logging context for file appenders before setting the buffer size #1975
- Remove javax.el from jersey-bean-validation #1976
- Exclude duplicated JTA 1.1 from dropwizard-hibernate dependencies #1977
- Add missing @UnwrapValidatedValue annotations #1993
- Fix HttpSessionListener.sessionDestroyed is not being called #2032
- Add flag to make ThreadNameFilter optional #2014

4.4.19 v1.1.0: Mar 21 2017

Complete changelog on GitHub

- Upgraded to Hibernate ORM 5.2.7, introducing a series of deprecations and API changes in preparation for Hibernate ORM 6 #1871
- Add runtime certificate reload via admin task #1799

- List available tasks lexically via admin task #1939
- Add support for optional resource protection #1931
- Invalid enum request parameters result in 400 response with possible choices #1734
- Enum request parameters are deserialized in the same fuzzy manner, as the request body #1734
- Request parameter name displayed in response to parse failure #1734
- Add `DurationParam` as a possible request parameter #1734
- Add `SizeParam` as a possible request parameter #1751
- Allow overriding of a default `ExceptionMapper` without re-registering all other defaults #1768
- Allow overriding of default `JsonProvider` #1788
- Finer-grain control of exception behaviour in view renderers #1820
- Default `WebApplicationException` handler preserves exception HTTP headers #1912
- JerseyClientBuilder can create rx-capable client #1721
- Configurable response for empty `Optional` return values #1784
- Add web test container agnostic way of invoking requests in `ResourceTestRule` #1778
- Remove OptionalValidatedValueUnwrapper #1583
- Allow constraints to be applied to type #1586
- Use LoadingCache in CachingAuthenticator #1615
- Switch cert and peer validation to false by default #1855
- Introduce CachingAuthorizer #1639
- Enhance logging of registered endpoints #1804
- Flush loggers on command exit instead of destroying logging #1947
- Add support for neverBlock on AsyncAppenders #1917
- Allow to disable caching of Mustache views #1289
- Add the `httpCompliance` option to the HTTP configuration #1825
- Add the `blockingTimeout` option to the HTTP configuration #1795
- Make `GZipHandler` sync-flush configurable #1685
- Add `min` and `mins` as valid `Duration` abbreviations #1833
- Register Jackson parameter-names modules #1908
- Native Jackson deserialization of enums when Jackson annotations are present #1909
- Add `JsonConfigurationFactory` for first-class support of the JSON configuration #1897
- Support disabled and enabled attributes for metrics #1957
- Support `@UnitOfWork` in sub-resources #1959
- Upgraded to Jackson 2.8.7
- Upgraded to Hibernate Validator 5.3.4.Final
- Upgraded to Hibernate ORM 5.2.8.Final
- Upgraded to Jetty 9.4.2.v20170220

- Upgraded to tomcat-jdbc 8.5.9
- Upgraded to Objenesis 2.5.1
- Upgraded to AssertJ 3.6.2
- Upgraded to classmate 1.3.3
- Upgraded to Metrics 3.2.2 #1970
- Upgraded to Mustache 0.9.4 #1766
- Upgraded to Mockito 2.7.12
- Upgraded to Liquibase 3.5.3
- Upgraded to Logback 1.2.1 #1918
- Upgraded to JDBI 2.78
- Upgraded to Jersey 2.25.1
- Upgraded to javassist 3.21.0-GA
- Upgraded to Guava 21.0
- Upgraded to SLF4J 1.7.24
- Upgraded to H2 1.4.193
- Upgraded to Joda-Time 2.9.7
- Upgraded to commons-lang3 3.5
- Upgraded to Apache HTTP Client 4.5.3

4.4.20 v1.0.7 Mar 20 2017

Complete changelog on GitHub

- Upgrade to Metrics 3.1.4 #1969

4.4.21 v1.0.6 Jan 30 2017

Complete changelog on GitHub

- Switch cert and peer validation to false by default #1855
- Add a JUnit rule for testing database interactions #1905

4.4.22 v1.0.5 Nov 18 2016

Complete changelog on GitHub

- Fix request logs with request parameter in layout pattern #1828

4.4.23 v1.0.4 Nov 14 2016

Complete changelog on GitHub

- Upgraded to Jersey 2.23.2 #1808
- Brought back support for request logging with `logback-classic` #1813

4.4.24 v1.0.3: Oct 28 2016

Complete changelog on GitHub

- Fix support maxFileSize and archivedFileCount #1660
- Upgraded to Jackson 2.7.8 #1755
- Upgraded to Mustache 0.9.4 #1766
- Prefer use of assertj's java8 exception assertions #1753

4.4.25 v1.0.2: Sep 23 2016

Complete changelog on GitHub

- Fix absence of request logs in Dropwizard 1.0.1 #1737

4.4.26 v1.0.1: Sep 21 2016

Complete changelog on GitHub

- Allow use of custom HostnameVerifier on clients #1664
- Allow to configure failing on unknown properties in the Dropwizard configuration #1677
- Fix request attribute-related race condition in Logback request logging #1678
- Log Jetty initialized SSLContext not the Default #1698
- Fix NPE of non-resource sub-resource methods #1718

4.4.27 v1.0.0: Jul 26 2016

Complete changelog on GitHub

- Using Java 8 as baseline
- `dropwizard-java8` bundle merged into mainline #1365
- HTTP/2 and server push support #1349
- `dropwizard-spdy` module is removed in favor of `dropwizard-http2` #1330
- Switching to `logback-access` for HTTP request logging #1415
- Support for validating return values in JAX-RS resources #1251
- Consistent handling null entities in JAX-RS resources #1251
- Support for validating bean members in JAX-RS resources #1572
- Returning an HTTP 500 error for entities that can't be serialized #1347

- Support serialisation of lazy loaded POJOs in Hibernate #1466
- Support fallback to the `toString` method during deserializing enum values from JSON #1340
- Support for setting default headers in Apache HTTP client #1354
- Printing help once on invalid command line arguments #1376
- Support for case insensitive and all single letter `SizeUnit` suffixes #1380
- Added a development profile to the build #1364
- All the default exception mappers in `ResourceTestRule` are registered by default #1387
- Allow DB minSize and initialSize to be zero for lazy connections #1517
- Ability to provide own `RequestLogFactory` #1290
- Support for authentication by polymorphic principals #1392
- Support for configuring Jetty's `inheritedChannel` option #1410
- Support for using `DropwizardAppRule` at the suite level #1411
- Support for adding multiple `MigrationBundles` #1430
- Support for obtaining server context paths in the `Application.run` method #1503
- Support for unlimited log files for file appender #1549
- Support for log file names determined by logging policy #1561
- Default Graphite reporter port changed from 8080 to 2003 #1538
- Upgraded to Apache HTTP Client 4.5.2
- Upgraded to argparse4j 0.7.0
- Upgraded to Guava 19.0
- Upgraded to H2 1.4.192
- Upgraded to Hibernate 5.1.0 #1429
- Upgraded to Hibernate Validator 5.2.4.Final
- Upgraded to HSQLDB 2.3.4
- Upgraded to Jadira Usertype Core 5.0.0.GA
- Upgraded to Jackson 2.7.6
- Upgraded to JDBI 2.73 #1358
- Upgraded to Jersey 2.23.1
- Upgraded to Jetty 9.3.9.v20160517 #1330
- Upgraded to JMH 1.12
- Upgraded to Joda-Time 2.9.4
- Upgraded to Liquibase 3.5.1
- Upgraded to liquibase-slf4j 2.0.0
- Upgraded to Logback 1.1.7
- Upgraded to Mustache 0.9.2
- Upgraded to SLF4J 1.7.21

- Upgraded to tomcat-jdbc 8.5.3
- Upgraded to Objenesis 2.3
- Upgraded to AssertJ 3.4.1
- Upgraded to Mockito 2.0.54-beta

4.4.28 v0.9.2: Jan 20 2016

Complete changelog on GitHub

- Support *@UnitOfWork* annotation outside of Jersey resources #1361

4.4.29 v0.9.1: Nov 3 2015

Complete changelog on GitHub

- Add `ConfigurationSourceProvider` for reading resources from classpath #1314
- Add `@UnwrapValidatedValue` annotation to *BaseReporterFactory.frequency* #1308, #1309
- Fix serialization of default configuration for `DataSourceFactory` by deprecating `PooledDataSourceFactory#getHealthCheckValidationQuery()` and `PooledDataSourceFactory#getHealthCheckValidationTimeout()` #1321, #1322
- Treat `null` values in JAX-RS resource method parameters of type `Optional<T>` as absent value after conversion #1323

4.4.30 v0.9.0: Oct 28 2015

Complete changelog on GitHub

- Various documentation fixes and improvements
- New filter-based authorization & authentication #952, #1023, #1114, #1162, #1241
- Fixed a security bug in `CachingAuthenticator` with caching results of failed authentication attempts #1082
- Correct handling misconfigured context paths in `ServerFactory` #785
- Logging context paths during application startup #994, #1072
- Support for Jersey Bean Validation #842
- Returning descriptive constraint violation messages #1039,
- Trace logging of failed constraint violations #992
- Returning correct HTTP status codes for constraint violations #993
- Fixed possible XSS in constraint violations #892
- Support for including caller data in appenders #995
- Support for defining custom logging factories (e.g. native Logback) #996
- Support for defining the maximum log file size in `FileAppenderFactory`. #1000
- Support for fixed window rolling policy in `FileAppenderFactory` #1218
- Support for individual logger appenders #1092

- Support for disabling logger additivity #1215
- Sorting endpoints in the application startup log #1002
- Dynamic DNS resolution in the Graphite metric reporter #1004
- Support for defining a custom `MetricRegistry` during bootstrap (e.g. with HdrHistogram) #1015
- Support for defining a custom `ObjectMapper` during bootstrap. #1112
- Added facility to plug-in custom DB connection pools (e.g. HikariCP) #1030
- Support for setting a custom DB pool connection validator #1113
- Support for enabling of removing abandoned DB pool connections #1264
- Support for credentials in a DB data source URL #1260
- Support for simultaneous work of several Hibernate bundles #1276
- HTTP(S) proxy support for Dropwizard HTTP client #657
- Support external configuration of TLS properties for Dropwizard HTTP client #1224
- Support for not accepting GZIP-compressed responses in HTTP clients #1270
- Support for setting a custom redirect strategy in HTTP clients #1281
- Apache and Jersey clients are now managed by the application environment #1061
- Support for request-scoped configuration for Jersey client #939
- Respecting Jackson feature for deserializing enums using `toString` #1104
- Support for passing explicit `Configuration` via test rules #1131
- On view template error, return a generic error page instead of template not found #1178
- In some cases an instance of Jersey HTTP client could be abruptly closed during the application lifetime #1232
- Improved build time build by running tests in parallel #1032
- Added JMH benchmarks #990
- Allow customization of Hibernate `SessionFactory` #1182
- Removed javax.el-2.x in favour of javax.el-3.0
- Upgraded to argparse4j 0.6.0
- Upgrade to AssertJ 2.2.0
- Upgraded to JDBI 2.63.1
- Upgraded to Apache HTTP Client 4.5.1
- Upgraded to Dropwizard Metrics 3.1.2
- Upgraded to Freemarker 2.3.23
- Upgraded to H2 1.4.190
- Upgraded to Hibernate 4.3.11.Final
- Upgraded to Jackson 2.6.3
- Upgraded to Jadira Usertype Core 4.0.0.GA
- Upgraded to Jersey 2.22.1
- Upgraded to Jetty 9.2.13.v20150730

- Upgraded to Joda-Time 2.9
- Upgraded to JSR305 annotations 3.0.1
- Upgraded to Hibernate Validator 5.2.2.Final
- Upgraded to Jetty ALPN boot 7.1.3.v20150130
- Upgraded to Jetty SetUID support 1.0.3
- Upgraded to Liquibase 3.4.1
- Upgraded to Logback 1.1.3
- Upgraded to Metrics 3.1.2
- Upgraded to Mockito 1.10.19
- Upgraded to SLF4J 1.7.12
- Upgraded to commons-lang3 3.4
- Upgraded to tomcat-jdbc 8.0.28

4.4.31 v0.8.5: Nov 3 2015

Complete changelog on GitHub

- Treat `null` values in JAX-RS resource method parameters of type `Optional<T>` as absent value after conversion #1323

4.4.32 v0.8.4: Aug 26 2015

- Upgrade to Apache HTTP Client 4.5
- Upgrade to Jersey 2.21
- Fixed user-agent shadowing in Jersey HTTP Client #1198

4.4.33 v0.8.3: Aug 24 2015

Complete changelog on GitHub

- Fixed an issue with closing the HTTP client connection pool after a full GC #1160

4.4.34 v0.8.2: Jul 6 2015

Complete changelog on GitHub

- Support for request-scoped configuration for Jersey client #1137
- Upgraded to Jersey 2.19 #1143

4.4.35 v0.8.1: Apr 7 2015

Complete changelog on GitHub

- Fixed transaction committing lifecycle for `@UnitOfWork` (#850, #915)
- Fixed noisy Logback messages on startup (#902)
- Ability to use providers in TestRule, allows testing of auth & views (#513, #922)
- Custom ExceptionMapper not invoked when Hibernate rollback (#949)
- Support for setting a time bound on DBI and Hibernate health checks
- Default configuration for views
- Ensure that JerseyRequest scoped ClientConfig gets propagated to HttpUriRequest
- More example tests
- Fixed security issue where info is leaked during validation of unauthenticated resources(#768)

4.4.36 v0.8.0: Mar 5 2015

Complete changelog on GitHub

- Migrated `dropwizard-spdy` from NPN to ALPN
- Dropped support for deprecated SPDY/2 in `dropwizard-spdy`
- Upgrade to argparse4j 0.4.4
- Upgrade to commons-lang3 3.3.2
- Upgrade to Guava 18.0
- Upgrade to H2 1.4.185
- Upgrade to Hibernate 4.3.5.Final
- Upgrade to Hibernate Validator 5.1.3.Final
- Upgrade to Jackson 2.5.1
- Upgrade to JDBI 2.59
- Upgrade to Jersey 2.16
- Upgrade to Jetty 9.2.9.v20150224
- Upgrade to Joda-Time 2.7
- Upgrade to Liquibase 3.3.2
- Upgrade to Mustache 0.8.16
- Upgrade to SLF4J 1.7.10
- Upgrade to tomcat-jdbc 8.0.18
- Upgrade to JSR305 annotations 3.0.0
- Upgrade to Junit 4.12
- Upgrade to AssertJ 1.7.1
- Upgrade to Mockito 1.10.17

- Support for range headers
- Ability to use Apache client configuration for Jersey client
- Warning when maximum pool size and unbounded queues are combined
- Fixed connection leak in CloseableLiquibase
- Support ScheduledExecutorService with daemon thread
- Improved DropwizardAppRule
- Better connection pool metrics
- Removed final modifier from Application#run
- Fixed gzip encoding to support Jersey 2.x
- Configuration to toggle regex [in/ex]clusion for Metrics
- Configuration to disable default exception mappers
- Configuration support for disabling chunked encoding
- Documentation fixes and upgrades

4.4.37 v0.7.1: Jun 18 2014

Complete changelog on GitHub

- Added instrumentation to `Task`, using metrics annotations.
- Added ability to blacklist SSL cipher suites.
- Added `@PATCH` annotation for Jersey resource methods to indicate use of the HTTP `PATCH` method.
- Added support for configurable request retry behavior for `HttpClientBuilder` and `JerseyClientBuilder`.
- Added facility to get the admin HTTP port in `DropwizardAppTestRule`.
- Added `ScanningHibernateBundle`, which scans packages for entities, instead of requiring you to add them individually.
- Added facility to invalidate credentials from the `CachingAuthenticator` that match a specified `Predicate`.
- Added a CI build profile for JDK 8 to ensure that Dropwizard builds against the latest version of the JDK.
- Added `--catalog` and `--schema` options to Liquibase.
- Added `stackTracePrefix` configuration option to `SyslogAppenderFactory` to configure the pattern prepended to each line in the stack-trace sent to syslog. Defaults to the TAB character, "t". Note: this is different from the bang prepended to text logs (such as "console", and "file"), as syslog has different conventions for multi-line messages.
- Added ability to validate `Optional` values using validation annotations. Such values require the `@UnwrapValidatedValue` annotation, in addition to the validations you wish to use.
- Added facility to configure the `User-Agent` for `HttpClient`. Configurable via the `userAgent` configuration option.
- Added configurable `AllowedMethodsFilter`. Configure allowed HTTP methods for both the application and admin connectors with `allowedMethods`.
- Added support for specifying a `CredentialProvider` for HTTP clients.

- Fixed silently overriding Servlets or ServletFilters; registering a duplicate will now emit a warning.
- Fixed `SyslogAppenderFactory` failing when the application name contains a PCRE reserved character (e.g. `/` or `$`).
- Fixed regression causing JMX reporting of metrics to not be enabled by default.
- Fixed transitive dependencies on log4j and extraneous sl4j backends bleeding in to projects. Dropwizard will now enforce that only Logback and slf4j-logback are used everywhere.
- Fixed clients disconnecting before the request has been fully received causing a "500 Internal Server Error" to be generated for the request log. Such situations will now correctly generate a "400 Bad Request", as the request is malformed. Clients will never see these responses, but they matter for logging and metrics that were previously considering this situation as a server error.
- Fixed `DiscoverableSubtypeResolver` using the system `ClassLoader`, instead of the local one.
- Fixed regression causing Liquibase `--dump` to fail to dump the database.
- Fixed the CSV metrics reporter failing when the output directory doesn't exist. It will now attempt to create the directory on startup.
- Fixed global frequency for metrics reporters being permanently overridden by the default frequency for individual reporters.
- Fixed tests failing on Windows due to platform-specific line separators.
- Changed `DropwizardAppTestRule` so that it no longer requires a configuration path to operate. When no path is specified, it will now use the applications' default configuration.
- Changed `Bootstrap` so that `getMetricsFactory()` may now be overridden to provide a custom instance to the framework to use.
- Upgraded to Guava 17.0 Note: this addresses a bug with BloomFilters that is incompatible with pre-17.0 BloomFilters.
- Upgraded to Jackson 2.3.3
- Upgraded to Apache HttpClient 4.3.4
- Upgraded to Metrics 3.0.2
- Upgraded to Logback 1.1.2
- Upgraded to h2 1.4.178
- Upgraded to JDBI 2.55
- Upgraded to Hibernate 4.3.5 Final
- Upgraded to Hibernate Validator 5.1.1 Final
- Upgraded to Mustache 0.8.15

4.4.38 v0.7.0: Apr 04 2014

Complete changelog on GitHub

- Upgraded to Java 7.
- Moved to the `io.dropwizard` group ID and namespace.
- Extracted out a number of reusable libraries: `dropwizard-configuration`, `dropwizard-jackson`, `dropwizard-jersey`, `dropwizard-jetty`, `dropwizard-lifecycle`, `dropwizard-logging`, `dropwizard-servlets`, `dropwizard-util`, `dropwizard-validation`.

- Extracted out various elements of `Environment` to separate classes: `JerseyEnvironment`, `LifecycleEnvironment`, etc.
- Extracted out `dropwizard-views-freemarker` and `dropwizard-views-mustache`. `dropwizard-views` just provides infrastructure now.
- Renamed `Service` to `Application`.
- Added `dropwizard-forms`, which provides support for multipart MIME entities.
- Added `dropwizard-spdy`.
- Added `AppenderFactory`, allowing for arbitrary logging appenders for application and request logs.
- Added `ConnectorFactory`, allowing for arbitrary Jetty connectors.
- Added `ServerFactory`, with multi- and single-connector implementations.
- Added `ReporterFactory`, for metrics reporters, with Graphite and Ganglia implementations.
- Added `ConfigurationSourceProvider` to allow loading configuration files from sources other than the filesystem.
- Added setuid support. Configure the user/group to run as and soft/hard open file limits in the `ServerFactory`. To bind to privileged ports (e.g. 80), enable `startsAsRoot` and set `user` and `group`, then start your application as the root user.
- Added builders for managed executors.
- Added a default `check` command, which loads and validates the service configuration.
- Added support for the Jersey HTTP client to `dropwizard-client`.
- Added Jackson Afterburner support.
- Added support for `deflate`-encoded requests and responses.
- Added support for HTTP Sessions. Add the annotated parameter to your resource method: `@Session HttpSession session` to have the session context injected.
- Added support for a "flash" message to be propagated across requests. Add the annotated parameter to your resource method: `@Session Flash message` to have any existing flash message injected.
- Added support for deserializing Java `enums` with fuzzy matching rules (i.e., whitespace stripping, -/_ equivalence, case insensitivity, etc.).
- Added `HibernateBundle#configure(Configuration)` for customization of Hibernate configuration.
- Added support for Joda Time `DateTime` arguments and results when using JDBI.
- Added configuration option to include Exception stack-traces when logging to syslog. Stack traces are now excluded by default.
- Added the application name and PID (if detectable) to the beginning of syslog messages, as is the convention.
- Added `--migrations` command-line option to `migrate` command to supply the migrations file explicitly.
- Validation errors are now returned as `application/json` responses.
- Simplified `AsyncRequestLog`; now standardized on Jetty 9 NCSA format.
- Renamed `DatabaseConfiguration` to `DataSourceFactory`, and `ConfigurationStrategy` to `DatabaseConfiguration`.
- Changed logging to be asynchronous. Messages are now buffered and batched in-memory before being delivered to the configured appender(s).

- Changed handling of runtime configuration errors. Will no longer display an Exception stack-trace and will present a more useful description of the problem, including suggestions when appropriate.
- Changed error handling to depend more heavily on Jersey exception mapping.
- Changed `dropwizard-db` to use `tomcat-jdbc` instead of `tomcat-dbcp`.
- Changed default formatting when logging nested Exceptions to display the root-cause first.
- Replaced `ResourceTest` with `ResourceTestRule`, a JUnit `TestRule`.
- Dropped Scala support.
- Dropped `ManagedSessionFactory`.
- Dropped `ObjectMapperFactory`; use `ObjectMapper` instead.
- Dropped `Validator`; use `javax.validation.Validator` instead.
- Fixed a shutdown bug in `dropwizard-migrations`.
- Fixed formatting of "Caused by" lines not being prefixed when logging nested Exceptions.
- Fixed not all available Jersey endpoints were being logged at startup.
- Upgraded to argparse4j 0.4.3.
- Upgraded to Guava 16.0.1.
- Upgraded to Hibernate Validator 5.0.2.
- Upgraded to Jackson 2.3.1.
- Upgraded to JDBI 2.53.
- Upgraded to Jetty 9.0.7.
- Upgraded to Liquibase 3.1.1.
- Upgraded to Logback 1.1.1.
- Upgraded to Metrics 3.0.1.
- Upgraded to Mustache 0.8.14.
- Upgraded to SLF4J 1.7.6.
- Upgraded to Jersey 1.18.
- Upgraded to Apache HttpClient 4.3.2.
- Upgraded to tomcat-jdbc 7.0.50.
- Upgraded to Hibernate 4.3.1.Final.

4.4.39 v0.6.2: Mar 18 2013

- Added support for non-UTF8 views.
- Fixed an NPE for services in the root package.
- Fixed exception handling in `TaskServlet`.
- Upgraded to Slf4j 1.7.4.
- Upgraded to Jetty 8.1.10.
- Upgraded to Jersey 1.17.1.

- Upgraded to Jackson 2.1.4.
- Upgraded to Logback 1.0.10.
- Upgraded to Hibernate 4.1.9.
- Upgraded to Hibernate Validator 4.3.1.
- Upgraded to tomcat-dbcp 7.0.37.
- Upgraded to Mustache.java 0.8.10.
- Upgraded to Apache HttpClient 4.2.3.
- Upgraded to Jackson 2.1.3.
- Upgraded to argparse4j 0.4.0.
- Upgraded to Guava 14.0.1.
- Upgraded to Joda Time 2.2.
- Added `retries` to `HttpClientConfiguration`.
- Fixed log formatting for extended stack traces, also now using extended stack traces as the default.
- Upgraded to FEST Assert 2.0M10.

4.4.40 v0.6.1: Nov 28 2012

- Fixed incorrect latencies in request logs on Linux.
- Added ability to register multiple `ServerLifecycleListener` instances.

4.4.41 v0.6.0: Nov 26 2012

- Added Hibernate support in `dropwizard-hibernate`.
- Added Liquibase migrations in `dropwizard-migrations`.
- Renamed `http.acceptorThreadCount` to `http.acceptorThreads`.
- Renamed `ssl.keyStorePath` to `ssl.keyStore`.
- Dropped `JerseyClient`. Use Jersey's `Client` class instead.
- Moved JDBI support to `dropwizard-jdbi`.
- Dropped `Database`. Use JDBI's `DBI` class instead.
- Dropped the `Json` class. Use `ObjectMapperFactory` and `ObjectMapper` instead.
- Decoupled JDBI support from tomcat-dbcp.
- Added group support to `Validator`.
- Moved CLI support to argparse4j.
- Fixed testing support for `Optional` resource method parameters.
- Fixed Freemarker support to use its internal encoding map.
- Added property support to `ResourceTest`.
- Fixed JDBI metrics support for raw SQL queries.
- Dropped Hamcrest matchers in favor of FEST assertions in `dropwizard-testing`.

- Split `Environment` into `Bootstrap` and `Environment`, and broke configuration of each into `Service`'s `#initialize(Bootstrap)` and `#run(Configuration, Environment)`.
- Combined `AbstractService` and `Service`.
- Trimmed down `ScalaService`, so be sure to add `ScalaBundle`.
- Added support for using `JerseyClientFactory` without an `Environment`.
- Dropped Jerkson in favor of Jackson's Scala module.
- Added `Optional` support for JDBI.
- Fixed bug in stopping `AsyncRequestLog`.
- Added `UUIDParam`.
- Upgraded to Metrics 2.2.0.
- Upgraded to Jetty 8.1.8.
- Upgraded to Mockito 1.9.5.
- Upgraded to tomcat-dbcp 7.0.33.
- Upgraded to Mustache 0.8.8.
- Upgraded to Jersey 1.15.
- Upgraded to Apache HttpClient 4.2.2.
- Upgraded to JDBI 2.41.
- Upgraded to Logback 1.0.7 and SLF4J 1.7.2.
- Upgraded to Guava 13.0.1.
- Upgraded to Jackson 2.1.1.
- Added support for Joda Time.

Note: Upgrading to 0.6.0 will require changing your code. First, your `Service` subclass will need to implement both `#initialize(Bootstrap<T>)` **and** `#run(T, Environment)`. What used to be in `initialize` should be moved to `run`. Second, your representation classes need to be migrated to Jackson 2. For the most part, this is just changing imports to `com.fasterxml.jackson.annotation.*`, but there are some subtler changes in functionality. Finally, references to 0.5.x's `Json`, `JerseyClient`, or `JDBI` classes should be changed to Jackon's `ObjectMapper`, Jersey's `Client`, and JDBI's `DBI` respectively.

4.4.42 v0.5.1: Aug 06 2012

- Fixed logging of managed objects.
- Fixed default file logging configuration.
- Added FEST-Assert as a `dropwizard-testing` dependency.
- Added support for Mustache templates (`*.mustache`) to `dropwizard-views`.
- Added support for arbitrary view renderers.
- Fixed command-line overrides when no configuration file is present.
- Added support for arbitrary `DnsResolver` implementations in `HttpClientFactory`.
- Upgraded to Guava 13.0 final.

- Fixed task path bugs.
- Upgraded to Metrics 2.1.3.
- Added `JerseyClientConfiguration#compressRequestEntity` for disabling the compression of request entities.
- Added `Environment#scanPackagesForResourcesAndProviders` for automatically detecting Jersey providers and resources.
- Added `Environment#setSessionHandler`.

4.4.43 v0.5.0: Jul 30 2012

- Upgraded to JDBI 2.38.1.
- Upgraded to Jackson 1.9.9.
- Upgraded to Jersey 1.13.
- Upgraded to Guava 13.0-rc2.
- Upgraded to HttpClient 4.2.1.
- Upgraded to tomcat-dbcp 7.0.29.
- Upgraded to Jetty 8.1.5.
- Improved `AssetServlet`:
  - More accurate `Last-Modified-At` timestamps.
  - More general asset specification.
  - Default filename is now configurable.
- Improved `JacksonMessageBodyProvider`:
  - Now based on Jackson's JAX-RS support.
  - Doesn't read or write types annotated with `@JsonIgnoreType`.
- Added `@MinSize`, `@MaxSize`, and `@SizeRange` validations.
- Added `@MinDuration`, `@MaxDuration`, and `@DurationRange` validations.
- Fixed race conditions in Logback initialization routines.
- Fixed `TaskServlet` problems with custom context paths.
- Added `jersey-text-framework-core` as an explicit dependency of `dropwizard-testing`. This helps out some non-Maven build frameworks with bugs in dependency processing.
- Added `addProvider` to `JerseyClientFactory`.
- Fixed `NullPointerException` problems with anonymous health check classes.
- Added support for serializing/deserializing `ByteBuffer` instances as JSON.
- Added `supportedProtocols` to SSL configuration, and disabled SSLv2 by default.
- Added support for `Optional<Integer>` query parameters and others.
- Removed `jersey-freemarker` dependency from `dropwizard-views`.
- Fixed missing thread contexts in logging statements.
- Made the configuration file argument for the `server` command optional.

- Added support for disabling log rotation.
- Added support for arbitrary KeyStore types.
- Added `Log.forThisClass()`.
- Made explicit service names optional.

4.4.44 v0.4.4: Jul 24 2012

- Added support for `@JsonIgnoreType` to `JacksonMessageBodyProvider`.

4.4.45 v0.4.3: Jun 22 2012

- Re-enable immediate flushing for file and console logging appenders.

4.4.46 v0.4.2: Jun 20 2012

- Fixed `JsonProcessingExceptionMapper`. Now returns human-readable error messages for malformed or invalid JSON as a `400 Bad Request`. Also handles problems with JSON generation and object mapping in a developer-friendly way.

4.4.47 v0.4.1: Jun 19 2012

- Fixed type parameter resolution in for subclasses of subclasses of `ConfiguredCommand`.
- Upgraded to Jackson 1.9.7.
- Upgraded to Logback 1.0.6, with asynchronous logging.
- Upgraded to Hibernate Validator 4.3.0.
- Upgraded to JDBI 2.34.
- Upgraded to Jetty 8.1.4.
- Added `logging.console.format`, `logging.file.format`, and `logging.syslog.format` parameters for custom log formats.
- Extended `ResourceTest` to allow for enabling/disabling specific Jersey features.
- Made `Configuration` serializable as JSON.
- Stopped lumping command-line options in a group in `Command`.
- Fixed `java.util.logging` level changes.
- Upgraded to Apache HttpClient 4.2.
- Improved performance of `AssetServlet`.
- Added `withBundle` to `ScalaService` to enable bundle mix-ins.
- Upgraded to SLF4J 1.6.6.
- Enabled configuration-parameterized Jersey containers.
- Upgraded to Jackson Guava 1.9.1, with support for `Optional`.
- Fixed error message in `AssetBundle`.

- Fixed `WebApplicationException``s being thrown by ``JerseyClient`.

4.4.48 v0.4.0: May 1 2012

Complete changelog on GitHub

- Switched logging from Log4j to Logback.
  - Deprecated `Log#fatal` methods.
  - Deprecated Log4j usage.
  - Removed Log4j JSON support.
  - Switched file logging to a time-based rotation system with optional GZIP and ZIP compression.
  - Replaced `logging.file.filenamePattern` with `logging.file.currentLogFilename` and `logging.file.archivedLogFilenamePattern`.
  - Replaced `logging.file.retainedFileCount` with `logging.file.archivedFileCount`.
  - Moved request logging to use a Logback-backed, time-based rotation system with optional GZIP and ZIP compression. `http.requestLog` now has `console`, `file`, and `syslog` sections.
- Fixed validation errors for logging configuration.
- Added `ResourceTest#addProvider(Class<?>)`.
- Added `ETag` and `Last-Modified` support to `AssetServlet`.
- Fixed `off` logging levels conflicting with YAML's helpfulness.
- Improved `Optional` support for some JDBC drivers.
- Added `ResourceTest#getJson()`.
- Upgraded to Jackson 1.9.6.
- Improved syslog logging.
- Fixed template paths for views.
- Upgraded to Guava 12.0.
- Added support for deserializing `CacheBuilderSpec` instances from JSON/YAML.
- Switched `AssetsBundle` and servlet to using cache builder specs.
- Switched `CachingAuthenticator` to using cache builder specs.
- Malformed JSON request entities now produce a `400 Bad Request` instead of a `500 Server Error` response.
- Added `connectionTimeout`, `maxConnectionsPerRoute`, and `keepAlive` to `HttpClientConfiguration`.
- Added support for using Guava's `HostAndPort` in configuration properties.
- Upgraded to tomcat-dbcp 7.0.27.
- Upgraded to JDBI 2.33.2.
- Upgraded to HttpClient 4.1.3.
- Upgraded to Metrics 2.1.2.
- Upgraded to Jetty 8.1.3.

- Added SSL support.

4.4.49 v0.3.1: Mar 15 2012

- Fixed debug logging levels for `Log`.

4.4.50 v0.3.0: Mar 13 2012

Complete changelog on GitHub

- Upgraded to JDBI 2.31.3.
- Upgraded to Jackson 1.9.5.
- Upgraded to Jetty 8.1.2. (Jetty 9 is now the experimental branch. Jetty 8 is just Jetty 7 with Servlet 3.0 support.)
- Dropped `dropwizard-templates` and added `dropwizard-views` instead.
- Added `AbstractParam#getMediaType()`.
- Fixed potential encoding bug in parsing YAML files.
- Fixed a `NullPointerException` when getting logging levels via JMX.
- Dropped support for `@BearerToken` and added `dropwizard-auth` instead.
- Added `@CacheControl` for resource methods.
- Added `AbstractService#getJson()` for full Jackson customization.
- Fixed formatting of configuration file parsing errors.
- `ThreadNameFilter` is now added by default. The thread names Jetty worker threads are set to the method and URI of the HTTP request they are currently processing.
- Added command-line overriding of configuration parameters via system properties. For example, `-Ddw.http.port=8090` will override the configuration file to set `http.port` to `8090`.
- Removed `ManagedCommand`. It was rarely used and confusing.
- If `http.adminPort` is the same as `http.port`, the admin servlet will be hosted under `/admin`. This allows Dropwizard applications to be deployed to environments like Heroku, which require applications to open a single port.
- Added `http.adminUsername` and `http.adminPassword` to allow for Basic HTTP Authentication for the admin servlet.
- Upgraded to Metrics 2.1.1.

4.4.51 v0.2.1: Feb 24 2012

- Added `logging.console.timeZone` and `logging.file.timeZone` to control the time zone of the timestamps in the logs. Defaults to UTC.
- Upgraded to Jetty 7.6.1.
- Upgraded to Jersey 1.12.
- Upgraded to Guava 11.0.2.
- Upgraded to SnakeYAML 1.10.

- Upgraded to tomcat-dbcp 7.0.26.
- Upgraded to Metrics 2.0.3.

4.4.52 v0.2.0: Feb 15 2012

- Switched to using `jackson-datatype-guava` for JSON serialization/deserialization of Guava types.
- Use `InstrumentedQueuedThreadPool` from `metrics-jetty`.
- Upgraded to Jackson 1.9.4.
- Upgraded to Jetty 7.6.0 final.
- Upgraded to tomcat-dbcp 7.0.25.
- Improved fool-proofing for `Service` vs. `ScalaService`.
- Switched to using Jackson for configuration file parsing. SnakeYAML is used to parse YAML configuration files to a JSON intermediary form, then Jackson is used to map that to your `Configuration` subclass and its fields. Configuration files which don't end in `.yaml` or `.yml` are treated as JSON.
- Rewrote `Json` to no longer be a singleton.
- Converted `JsonHelpers` in `dropwizard-testing` to use normalized JSON strings to compare JSON.
- Collapsed `DatabaseConfiguration`. It's no longer a map of connection names to configuration objects.
- Changed `Database` to use the validation query in `DatabaseConfiguration` for its `#ping()` method.
- Changed many `HttpConfiguration` defaults to match Jetty's defaults.
- Upgraded to JDBI 2.31.2.
- Fixed JAR locations in the CLI usage screens.
- Upgraded to Metrics 2.0.2.
- Added support for all servlet listener types.
- Added `Log#setLevel(Level)`.
- Added `Service#getJerseyContainer`, which allows services to fully customize the Jersey container instance.
- Added the `http.contextParameters` configuration parameter.

4.4.53 v0.1.3: Jan 19 2012

- Upgraded to Guava 11.0.1.
- Fixed logging in `ServerCommand`. For the last time.
- Switched to using the instrumented connectors from `metrics-jetty`. This allows for much lower-level metrics about your service, including whether or not your thread pools are overloaded.
- Added FindBugs to the build process.
- Added `ResourceTest` to `dropwizard-testing`, which uses the Jersey Test Framework to provide full testing of resources.
- Upgraded to Jetty 7.6.0.RC4.
- Decoupled URIs and resource paths in `AssetServlet` and `AssetsBundle`.

- Added `rootPath` to `Configuration`. It allows you to serve Jersey assets off a specific path (e.g., `/resources/*` vs `/*`).
- `AssetServlet` now looks for `index.htm` when handling requests for the root URI.
- Upgraded to Metrics 2.0.0-RC0.

4.4.54 v0.1.2: Jan 07 2012

- All Jersey resource methods annotated with `@Timed`, `@Metered`, or `@ExceptionMetered` are now instrumented via `metrics-jersey`.
- Now licensed under Apache License 2.0.
- Upgraded to Jetty 7.6.0.RC3.
- Upgraded to Metrics 2.0.0-BETA19.
- Fixed logging in `ServerCommand`.
- Made `ServerCommand#run()` non-`final`.

4.4.55 v0.1.1: Dec 28 2011

- Fixed `ManagedCommand` to provide access to the `Environment`, among other things.
- Made `JerseyClient`'s thread pool managed.
- Improved ease of use for `Duration` and `Size` configuration parameters.
- Upgraded to Mockito 1.9.0.
- Upgraded to Jetty 7.6.0.RC2.
- Removed single-arg constructors for `ConfiguredCommand`.
- Added `Log`, a simple front-end for logging.

4.4.56 v0.1.0: Dec 21 2011

- Initial release

4.5 Security

No known issues exist

4.6 Documentation TODOs

CHAPTER 5

Other Versions

- 1.3.x
- 1.2.x
- 1.1.x
- 1.0.x
- 0.9.x
- 0.8.x
- 0.7.x
- 0.6.2