Dropwizard Documentation

Release 0.7.0

Coda Hale

June 09, 2016

Contents

1 Getting Started 3

2 User Manual 15

3 About Dropwizard 67

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 less 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-303 reference implementation, provides an easy, declarative framework for validating user input and generating helpful, 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 straight-forward 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 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. (We're assuming you know how to create a new Maven project. If not, you can use this to get started.)

First, add a `dropwizard.version` property to your POM with the current version of Dropwizard (which is 0.7.0):

```
<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 looks 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;
    }

    @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 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);
    }

    @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.core;

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() {
        // 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 Java Bean 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 own 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 will look like this:

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

import com.example.helloworld.core.Saying;
import com.google.common.base.Optional;
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;

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

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;
    }

    @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>1.6</version>
    <configuration>
        <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.ServicesReso
                    <transformer implementation="org.apache.maven.plugins.shade.resource.ManifestReso
                        <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.

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 /Use
[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.Serv
INFO  [2011-12-03 00:38:32,999] com.sun.jersey.server.impl.application.WebApplicationImpl: Initiating
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.Serv
INFO  [2011-12-03 00:38:33,235] org.eclipse.jetty.server.AbstractConnector: Started BlockingChannelCo
INFO  [2011-12-03 00:38:33,238] org.eclipse.jetty.server.AbstractConnector: Started SocketConnector@0
```

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.

But 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

In general, 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*.

Our applications tend to look like this:

- `com.example.myapplication`:
  - `api`: *Representations*.
  - `cli`: *Commands*
  - `client`: *Client* implementation for your application

- `core`: Domain implementation
- `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.

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.

In order 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 ExampleApplicationConfiguration extends Configuration {
    @Valid
    @NotNull
    private MessageQueueFactory messageQueue = new MessageQueueFactory();

    @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's *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.

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). 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
```

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` provides a simple way to serve static assets from your application's `src/main/resources/assets` directory as files available from `/assets/*` in your application.

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

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:
  type: simple
  applicationContextPath: /application/*  # Default value*
```

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) {
    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"));
}
```

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.

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

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. Dropwizard will treat the first 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.

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.) 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 Exce
        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. For example:

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

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.

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 and even override the levels of other loggers in your YAML configuration file:

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

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:

```yaml
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.

```
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:

```
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
```

2.1.12 Testing Applications

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

```
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(any(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,
                        @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 to 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.

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.

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,
                    @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 *Validation*.

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

Note: If your request entity parameter isn't annotated with `@Valid`, it won't be validated.

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

If your resource class unintentionally throws an exception, Dropwizard will log that exception (including stack traces) and return a terse, safe `text/plain 500 Internal Server Error` response.

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.

If you throw a subclass of `WebApplicationException` jersey will map that to a defined response.

If you want more control, you can also declare JerseyProviders in your Environment to map Exceptions to certain responses by calling `JerseyEnvironment#register(Object)` with an implementation of

javax.ws.rs.ext.ExceptionMapper. e.g. Your resource throws an InvalidArgumentException, but the response would be 400, bad request.

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() {
    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() {
        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"
}
```

Validation

Like *Configuration*, you can add validation annotations to fields of your representation classes and validate them. If we're accepting client-provided `Person` objects, we probably want to ensure that the `name` field of the object isn't `null` or blank. 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 Response replace(@Valid Person person) {
    // ...
}
```

If the `name` field is missing, Dropwizard will return a `text/plain 422 Unprocessable Entity` response detailing the validation errors:

```
* name may not be empty
```

Advanced

More complex validations (for example, cross-field comparisons) are often hard to do using declarative annotations. As an emergency maneuver, add the `@ValidationMethod` to any `boolean`-returning method which begins with `is`:

```
@ValidationMethod(message="may not be Coda")
public boolean isNotCoda() {
    return !("Coda".equals(name));
}
```

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

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 provides you with the means to do so. If you want to stop the request from reaching your resources, throw a web-application `WebApplicationException`, if you want to modify the request or let it pass through the filter, return it.

```
public class DateNotSpecifiedFilter implements ContainerRequestFilter {

    @Context ExtendedUriInfo extendedUriInfo;

    @Override
    public ContainerRequest filter(ContainerRequest request) {
        boolean methodNeedsDateHeader = extendedUriInfo.getMatchedMethod().isAnnotationPresent(DateRe
        String dateHeader = request.getHeaderValue(HttpHeaders.DATE);

        if (methodNeedsDateHeader && dateHeader == null) {
            Exception cause = new IllegalArgumentException("Date Header was not specified");
            throw new WebApplicationException(cause, Response.Status.BAD_REQUEST);
        } else {
            return request;
        }
    }
}
```

This example checks the request for the "Date" header, and denies the request if was ommitted and the method this request would call has a certain annotation present. You can then register this filter in your Application class, like so:

```
environment.jersey().getResourceConfig().getContainerRequestFilters().add(new DateNotSpecifiedFilter
```

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 ommited for brevity

    @Override
    public void doFilter(ServletRequest request, ServletResponse response, FilterChain chain) throws
        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
            } 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("DateHeaderServletFilter", new DateHeaderServletFilter())
                      .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, version 4.3 and Jersey Client, version 1.18.

2.2.1 Apache HttpClient, version 4.3

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
    @JsonProperty
    private HttpClientConfiguration httpClient = new HttpClientConfiguration();

    public HttpClientConfiguration getHttpClientConfiguration() {
        return 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.getHttpClientConfi
                                                                     .build();
    environment.addResource(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 execut 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, version 1.18

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
    @JsonProperty
    private JerseyClientConfiguration httpClient = new JerseyClientConfiguration();

    public JerseyClientConfiguration getJerseyClientConfiguration() {
        return httpClient;
    }
}
```

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.getJerseyClientConfigurat
                                                              .build(getName());
    environment.addResource(new ExternalServiceResource(client));
}
```

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
    @JsonProperty
    private DataSourceFactory database = new DataSourceFactory();

    public DataSourceFactory getDataSourceFactory() {
        return database;
    }
}
```

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

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

```
    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. Note the `ClassNotFoundException` is thrown by the `DBIFactory` class when the `build` method is unable to locate the JDBC driver class. This will cause the service to exit displaying the output of the exception.

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

  # 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 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 an
  evictionInterval: 10s

  # the minimum amount of time an connection must sit idle in the pool before it is eligible for evi
  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 Guava Support

`dropwizard-jdbi` supports `Optional<T>` arguments and `ImmutableList<T>` and `ImmutableSet<T>` query results.

2.3.6 Joda Time Support

`dropwizard-jdbi` supports joda-time `DateTime` arguments and `DateTime` fields in query results.

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
    @JsonProperty("database")
    private DataSourceFactory database = new DataSourceFactory();

    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.

For example, to create a new `people` table, I 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-3.0.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 change set 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 change sets 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 change sets, use the `--count` flag.

2.4.8 Rolling Back Your Schema

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

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 Change Set

To mark a pending change set 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 change set as applied. You can also use the `--all` flag to mark all pending change sets as applied.

2.4.14 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
    @JsonProperty("database")
    private DataSourceFactory database = new DataSourceFactory();

    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<ExampleConfigurat
    @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) throws ClassNotFoundException {
    final UserDAO dao = new UserDAO(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
@Timed
```

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

This will automatically open a session, begin a transaction, call `findByPerson`, 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).

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>` interface, which has a single method:

```
public class SimpleAuthenticator 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.absent();
    }
}
```

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 to not 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:

```
CachingAuthenticator.wrap(ldapAuthenticator,
                          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 Basic Authentication

The `BasicAuthProvider` enables HTTP Basic authentication, and requires an authenticator which takes instances of `BasicCredentials`:

```
@Override
public void run(ExampleConfiguration configuration,
                Environment environment) {
    environment.jersey().register(new BasicAuthProvider<User>(new ExampleAuthenticator(),
                                                               "SUPER SECRET STUFF"));
}
```

2.6.3 OAuth2

The `OAuthProvider` enables OAuth2 bearer-token authentication, and requires an authenticator which takes an instance of `String`.

```
@Override
public void run(ExampleConfiguration configuration,
                Environment environment) {
    environment.jersey().register(new OAuthProvider<User>(new ExampleAuthenticator(),
                                                           "SUPER SECRET STUFF"));
}
```

2.6.4 Protecting Resources

To protect a resource, simply include an `@Auth`-annotated principal as one of your resource method parameters:

```
@GET
public SecretPlan getSecretPlan(@Auth User user) {
    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.

If you have a resource which is optionally protected (e.g., you want to display a logged-in user's name but not require login), set the `required` attribute of the annotation to `false`:

```
@GET
public HomepageView getHomepage(@Auth(required = false) User user) {
    return new HomepageView(Optional.fromNullable(user));
}
```

If there is no authenticated principal, `null` is used instead, and your resource method is still called.

2.7 Dropwizard Views

The `dropwizard-views-mustache` & `dropwizard-views-freemarker` modules provides 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());
}
```

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 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.

2.8 Dropwizard & Scala

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

2.9 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.

2.9.1 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 actually 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 {
    @JsonProperty
    private String name;

    @JsonProperty
    private String email;

    private Person() {
        // Jackson deserialization
    }

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

    public String getName() {
        return name;
    }

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

    public String getEmail() {
        return email;
    }

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

    // 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.fest.assertions.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");
        assertThat(MAPPER.writeValueAsString(person))
                .isEqualTo(fixture("fixtures/person.json"));
    }
}
```

This test uses FEST matchers 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 via a normalized JSON string representation, so whitespace 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.fest.assertions.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 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 FEST matchers and JUnit to test that when a `Person` instance is deserialized via Jackson from the specified JSON fixture it matches the given object.

2.9.2 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.fest.assertions.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);
        // 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.client().resource("/person/blah").get(Person.class))
                .isEqualTo(person);
        verify(dao).fetchPerson("blah");
    }
}
```

Instansiate 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 `#client()`, which returns a Jersey `Client` instance 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?

Should you, at some point, grow tired of the near-infinite amount of debug logging produced by `ResourceTestRule` you can use the `java.util.logging` API to silence the `com.sun.jersey` logger.

2.9.3 Integrated Testing

It can be useful to start up your entire app and hit it with real HTTP requests during testing. This can be achieved by adding `DropwizardAppRule` to your JUnit test class, which 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.

```
public class LoginAcceptanceTest {

    @ClassRule
    public static final DropwizardAppRule<TestConfiguration> RULE =
            new DropwizardAppRule<TestConfiguration>(MyApp.class, resourceFilePath("my-app-config.yaml

    @Test
```

```
    public void loginHandlerRedirectsAfterPost() {
        Client client = new Client();

        ClientResponse response = client.resource(
                String.format("http://localhost:%d/login", RULE.getLocalPort()))
                .post(ClientResponse.class, loginForm());

        assertThat(response.getStatus()).isEqualTo(302);
    }
}
```

2.10 Dropwizard Example, Step by Step

The `dropwizard-example` module provides you with a working example of a dropwizard app

- Open a terminal
- Make sure you have maven installed
- Make sure java home points at JDK 7
- Make sure you have curl
- mvn dependency:resolve
- mvn clean compile install
- mvn eclipse:eclipse -DdownloadSources=true
- From eclipse, File –> Import –> Existing Project into workspace
- java -jar ~/git/dropwizard/dropwizard-example/target/dropwizard-example-0.7.0-SNAPSHOT.jar db migrate example.yml
- The above ran the liquibase migration in /src/main/resources/migrations.xml, creating the table schema
- You can now start the app in your IDE by running `java -jar ~/git/dropwizard/dropwizard-example/target/dropwizard-example-0.7.0-SNAPSHOT.jar db migrate example.yml`
- Alternatively you can run this file in your IDE: `com.example.helloworld.HelloWorldApplication server example.yml`
- Insert a new person: `curl -H "Content-Type: application/json" -X POST -d '{"fullName":"Coda Hale", "jobTitle" : "Chief Wizard" }' http://localhost:8080/people`
- Retrieve that person: `curl http://localhost:8080/people/1`
- View the freemarker template: `curl http://localhost:8080/people/1/view_freemarker`
- View the mustache template: `curl http://localhost:8080/people/1/view_mustache`

2.11 Dropwizard Configuration Reference

The `dropwizard-configuration` module provides you with a polymorphic configuration mechanism.

2.11.1 Servers

```
server:
  type: default
  maxThreads: 1024
```

All

Name	Default	Description
type	default	• default • simple
maxThreads	1024	The maximum number of threads to use for requests.
minThreads	8	The minimum number of threads to use for requests.
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, to cleanly shutdown before forcibly terminating them.
allowedMethods	`GET`, `POST`, `PUT`, `DELETE`, `HEAD`, `OPTIONS`, `PATCH`	The set of allowed HTTP methods. Others will be rejected with a 405 Method Not Allowed response.

GZip

```
server:
  gzip:
    bufferSize: 8KiB
```

Name	Default	Description
enabled	true	If true, all requests with gzip in their Accept-Content-Encoding headers will have their response entities encoded with gzip.
minimumEntitySize	256 bytes	All response entities under this size are not compressed.
bufferSize	8KiB	The size of the buffer to use when compressing.
excludedUserAgents	[]	The set of user agents to exclude from compression.
compressedMimeTypes	[]	If specified, the set of mime types to compress.

Request Log

```
server:
  requestLog:
    timeZone: UTC
```

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. *TODO* See logging/appender refs for more info

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*

```
server:
  adminMinThreads: 1
  adminMaxThreads: 64
  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.

2.11.2 Connectors

HTTP

```
# Extending from the default server configuration
server:
  applicationConnectors:
    - type: http
      port: 8080
      bindHost: 127.0.0.1 # only bind to loopback
      headerCacheSize: 512 bytes
      outputBufferSize: 32KiB
      maxRequestHeaderSize: 8KiB
      maxResponseHeaderSize: 8KiB
      inputBufferSize: 8KiB
      idleTimeout: 30 seconds
      minBufferPoolSize: 64 bytes
      bufferPoolIncrement: 1KiB
      maxBufferPoolSize: 64KiB
      acceptorThreads: 1
      selectorThreads: 2
      acceptQueueSize: 1024
```

```
      reuseAddress: true
      soLingerTime: 345s
      useServerHeader: false
      useDateHeader: true
      useForwardedHeaders: true
```

Name	Default	Description
port	8080	The TCP/IP port on which to listen for incoming connections.
bindHost	(none)	The hostname to bind to.
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.
minBufferPoolSize	64 bytes	The minimum size of the buffer pool.
bufferPoolIncrement	1KiB	The increment by which the buffer pool should be increased.
maxBufferPoolSize	64KiB	The maximum size of the buffer pool.
acceptorThreads	# of CPUs/2	The number of worker threads dedicated to accepting connections.
selectorThreads	# of CPUs	The number of worker threads dedicated to sending and receiving data.
acceptQueueSize	(OS default)	The size of the TCP/IP accept queue for the listening socket.
reuseAddress	true	Whether or not `SO_REUSEADDR` is enabled on the listening socket.
soLingerTime	(disabled)	Enable/disable `SO_LINGER` with the specified linger time.
useServerHeader	false	Whether or not to add the `Server` header to each response.
useDateHeader	true	Whether or not to add the `Date` header to each response.
useForwardedHeaders	true	Whether or not to look at `X-Forwarded-*` headers added by proxies. See `ForwardedRequestCustomize` for details.

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: true
      validatePeers: true
      supportedProtocols: SSLv3
      supportedCipherSuites: TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA256
      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	true	Whether or not to validate TLS certificates before starting. If enabled, Dropwizard will refuse to start with expired or otherwise invalid certificates.
validatePeers	true	Whether or not to validate TLS peer certificates.
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
excludedCipherSuites	(none)	A list of cipher suites (e.g., `TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA256`) which are excluded. These cipher suites will be refused and exclusion takes higher precedence than inclusion, such that if a cipher suite is listed in `supportedCipherSuites` and `excludedCipherSuitse`, the cipher suite will be excluded. To verify that the proper cipher suites are being whitelisted and blacklisted, it is recommended to use the tool sslyze.
allowRenegotiation	true	Whether or not TLS renegotiation is allowed.
endpointIdentificationAlgorithm	(none)	Which endpoint identification algorithm, if any, to use during the TLS handshake.

SPDY

Extends the attributes that are available to the *HTTPS connector*

```
server:
  applicationConnectors:
    - type: spdy3
      port: 8445
      keyStorePath: example.keystore
      keyStorePassword: example
      validateCerts: false
```

Name	Default	Description
pushStrategy	(none)	The push strategy to use for server-initiated SPDY pushes.

2.11.3 Logging

```
logging:
  level: INFO
  loggers:
    io.dropwizard: INFO
  appenders:
    - type: console
```

Name	Default	Description
level	Level.INFO	Logback logging level
loggers	(none)	
appenders	(none)	one of console, file or syslog

Console

```
logging:
  level: INFO
  appenders:
    - type: console
      threshold: ALL
      timeZone: UTC
      target: stdout
      logFormat: # TODO
```

Name	Default	Description
type	REQUIRED	The appender type. Must be `console`.
threshold	ALL	The lowest level of events to print to the console.
timeZone	UTC	The time zone to which event timestamps will be converted.
target	stdout	The name of the standard stream to which events will be written. Can be `stdout` or `stderr`.
logFormat	default	The Logback pattern with which events will be formatted. See the Logback documentation for details.

File

```
logging:
  level: INFO
  appenders:
    - type: file
      currentLogFilename: /var/log/myapplication.log
      threshold: ALL
      archive: true
      archivedLogFilenamePattern: /var/log/myapplication-%d.log
      archivedFileCount: 5
      timeZone: UTC
      logFormat: # TODO
```

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.
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. `%d` is replaced with the date in `yyyy-MM-dd` form, and the fact that it ends with `.gz` indicates the file will be gzipped as it's archived. Likewise, filename patterns which end in `.zip` will be filled as they are archived.
archivedFileCount	5	The number of archived files to keep. Must be between `1` and `50`.
timeZone	UTC	The time zone to which event timestamps will be converted.
logFormat	default	The Logback pattern with which events will be formatted. See the Logback documentation for details.

Syslog

```
logging:
  level: INFO
  appenders:
    - type: syslog
      host: localhost
      port: 514
      facility: local0
      threshold: ALL
      stackTracePrefix: \t
      logFormat: # TODO
```

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	default	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)

2.11.4 Metrics

The metrics configuration has two fields; frequency and reporters.

```
metrics:
  frequency: 1 second
  reporters:
    - type: <type>
```

Name	Default	Description
frequency	1 second	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)
      frequency: 1 second
```

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.
frequency	(none)	The frequency to report metrics. Overrides the default.

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*

```
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 annouce 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*

```
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.

2.11.5 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.
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.

JerseyClient

Extends the attributes that are available to *http clients*

See JerseyClientConfiguration and HttpClientConfiguration for more options.

```
jerseyClient:
  minThreads: 1
  maxThreads: 128
  gzipEnabled: true
  gzipEnabledForRequests: true
```

Name	Default	Description
minThreads	1	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.
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.

CHAPTER 3

About Dropwizard

3.1 Contributors

Many, many thanks to:

- Adam Jordens
- Adam Marcus
- Alex Heneveld
- Anders Hedström
- Andrei Savu
- Andrew Clay Shafer.
- Armando Singer
- Arun Horne
- Brandon Beck
- Brian McCallister
- Brian O'Neill
- Bruce Ritchie
- Cagatay Kavukcuoglu
- Cameron Fieber
- Cemalettin Koc
- Chris Gray
- Chris Tierney
- Christopher Currie
- Christopher Elkins
- Collin VanDyck
- Dale Wijnand
- Dan Everton
- David Morgantini

- David Stendardi
- Derek Stainer
- Eric Tschetter
- Fredrik Sundberg
- Hal Hildebrand
- Ian Eure
- James Ward
- Jared Stehler
- Jochen Schalanda
- Joshua Spiewak
- Justin Rudd
- Mark Wolfe
- Michael Fairley
- Mårten Gustafson
- Nick Telford
- Ori Schwartz
- Sam Perman
- Sam Quigley
- Scott Askew
- Sebastian Hartte
- Tatu Saloranta
- Ted Nyman
- Tom Crayford
- Tom Morris
- Vidit Drolia
- Xavier Shay

3.2 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!

3.3 Release Notes

3.3.1 v0.7.1

- 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 configurble 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 connnectors 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 permenantly 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 5.3.5 Final
- Upgraded to Hibernate Validator 5.1.1 Final
- Upgraded to Mustache 0.8.15

3.3.2 v0.7.0: Apr 04 2014

- 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 `startAsRoot` 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-file` 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.

3.3.3 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.

3.3.4 v0.6.1: Nov 28 2012

- Fixed incorrect latencies in request logs on Linux.
- Added ability to register multiple `ServerLifecycleListener` instances.

3.3.5 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.

3.3.6 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`.

3.3.7 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.

3.3.8 v0.4.4: Jul 24 2012

- Added support for `@JsonIgnoreType` to `JacksonMessageBodyProvider`.

3.3.9 v0.4.3: Jun 22 2012

- Re-enable immediate flushing for file and console logging appenders.

3.3.10 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.

3.3.11 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`.

3.3.12 v0.4.0: May 1 2012

- 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.

3.3.13 v0.3.1: Mar 15 2012

- Fixed debug logging levels for `Log`.

3.3.14 v0.3.0: Mar 13 2012

- 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.

3.3.15 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.

3.3.16 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.

3.3.17 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.

3.3.18 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`.

3.3.19 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.

3.3.20 v0.1.0: Dec 21 2011

- Initial release

3.4 Documentation TODOs