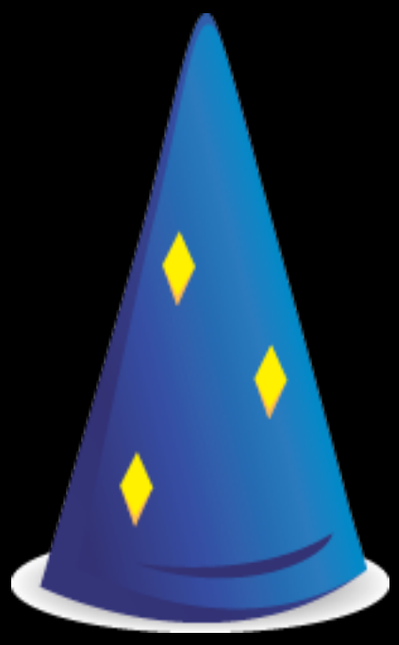

Dropwizard

Make features not WAR

Ryan Kennedy

Infrastructure Engineering at Yammer

@rckenned

The road to the renaissance

We'll be taking the long road to the Java renaissance by way of Service Oriented Architectures. So bear with me for a few slides.

Inevitably every major service decomposes

I'm going to lead into this talk with a discussion around service decomposition. Yammer, Twitter, Netflix. These are all major online services which have, over time, decomposed themselves from monolithic systems to Service Oriented Architectures. For the most part, they've done this because…

Monoliths don't scale technically or organizationally

It's an inevitability of every major online service to decompose because monoliths don't scale. Technically they are a burden as you increase traffic and try to figure out how to scale individual pieces of your monolith. Organizationally you suffer from too many cooks in the kitchen.

Inevitably you turn to Service Oriented Architectures

You begin breaking the monolith apart into components that scale individually. You begin breaking the monolith apart into codebases that scale organizationally. The way many successful companies do this is by decomposing the monolith into a series of separate services.

Why?

Why is it that a service oriented architecture scales technically and organizationally? There's a few reasons.

Loose Coupling

Service oriented architectures allow your system to be loosely coupled. Codebases become smaller and more focused in their concerns. Compared to the monolith which is larger and broader in its concerns. Integrations are done at the API level rather than the code level. You can push updates to the internals of a service without telling anyone. You can upgrade libraries without telling anyone.

Distributed Execution

Loose coupling allows you to have distributed execution. This means teams can work independently, coordinating just long enough to agree on the communication between components. Lower coordination costs lead to more productive teams. Yammer has been successful in large part due to eliminating coordination costs, both at the technical level and the organizational level.

Reusability

If you keep the services general enough, they can be reused. For example, at Yammer we have an image resizing service that we built to be able to dynamically scale user and group avatar images. When it came time to add a new feature allowing users to see an inline preview of large image attachments, we were able to reuse that service for free.

Independent provisioning

Since your application is broken into distinct services, you can scale by adding resources where they are needed. At Yammer we have one service that does search indexing and another that performs user queries. We can scale our indexing horsepower independently of our query horsepower, which is good because we index on every change but users search relatively infrequently. If that changed, we could scale queries up independently of scaling indexing.

However…
Service Oriented Architectures bring transaction costs

SOA is nice, but it does come at a cost. If it was pure profit everyone would do it all the time. What are some of those costs?

Provisioning

In the monolithic world you add a feature and re-deploy your application. In an SOA world you're adding more machines much more frequently. Every time you add a new service you need to find hardware to run it on. That may involve buying and racking hardware. You have to install and configuring an OS.

Deployment

Once you have hardware and an OS, you have to get your application on to it. In a monolithic world you have your deploy script. Sometimes you add or remove hosts from that script. In a SOA world you add a whole new service. That needs different software and configurations installed onto a different set of machines.

Monitoring and Logging

Once you've deployed your software, you have to monitor it. You need to collect logs from it. Every time you add a service that's another set of metrics to collect. Another set of logs to rotate and archive.

Alerting

Most importantly, you have to know when your software isn't working so you need alerting of some sort. Each service added brings a new set of things to alert on. This service is a clustered key-value store with single master replication so you want to know when writes are failing because you can't elect a new master node.

How do we get there?

So how do we mere mortals get from our easy monolithic world to the SOA world where we can take advantage of all of the benefits of SOA while not being killed by the transactional costs?

There are a lot of hot technologies these days

There are a lot of hot technologies these days for building services. The Ruby camp loves Rails and Sinatra. Python has Django and Pylons. And we can't forget the loudest camp of all these days…nodejs.

Java is generally not hot these days

Thanks to verbose APIs like servlets. Thanks to the EJB boogeyman. Thanks to dealing with countless security issues in Tomcat. I can walk into any coffee shop in SF these days and find someone looking for a Rails/Python/nodejs job. Finding someone looking for Java work in this part of town is not as easy.

But we can change that

I wouldn't be here in the Java Renaissance track raising the issue if I wasn't also here with a solution. There's nothing inherently wrong with Java or the base tools it provides. We just need to bring some up-to-date tools to the table and assemble them in a way that makes building services easier and more effective.

Yammer deals with the transaction costs through dropwizard

I joined Yammer 2 years ago. At the time we were a Rails monolith and a JVM service for doing browser push notifications. Today we're a Rails monolith and close to 20 JVM services, thanks in large part to Dropwizard.

What is dropwizard?

So what is this magical concoction that works so well for Yammer? What's in this mystical snake oil that I'm trying to sell to all of you?

Production ready services

Most importantly, Dropwizard is a framework for producing production ready services. At the end of the day, features in production are the thing that produces value for the company. So it's vital that Dropwizard make this happen.

HTTP and friends

In particular, we leverage HTTP to the fullest. We build RESTful JSON services. This provides us with a very loose coupling. If you want to talk to a service, whip out an HTTP library. If your language doesn't have an HTTP library, it's probably a good idea to rethink your choice of language. HTTP is also well supported as infrastructure. You're not going to have a hard time finding a caching load balancer, for instance.

Java

I'm probably/maybe preaching to most of the choir in this room, but Java is a good thing. The JVM is fast, which is always nice. The language is capable with more than 40k libraries in Maven Central, some of which are bound to be usable. And it's incredibly transparent. You can answer a tremendous number of questions regarding what it's doing internally through the available tools. The same can not be said of much of the competition.

Saves us time

Batteries INCLUDED

Dropwizard includes everything you need to ship to production. Configuration, logging, metrics, health checks, sane HTTP clients with connection pool management, authentication, views and more. This means you save time. You'll spend more time shipping to production than you will figuring out how to fulfill some obligation you have to ship to production because DW takes care of most/all of the obligations.

We did this so we (and you) don't have to

Ultimately, we did this so we (and now you, since we open sourced it) wouldn't have to do it again. A capable engineer with some dropwizard experience can have a production ready Hello, World in 15 minutes. For dropwizard-ready organizations where you already have a deployer in place, you can have that same application in production in another minute.

How do I dropwizard?

So by now you're all wondering if I have a Square reader with me, ready to take your money. First, this stuff is free so put your cards away. There is a tip jar in my bag, however. Now that we're past cost, let's talk about the mechanics of a dropwizard application so you can see just how easy this is.

Hello, World!

in dropwizard 0.6.0-SNAPSHOT

This is a Hello, World application built on the current mainline of development, 0.6.0-SNAPSHOT. This application is actually included in the dropwizard github repository if you want to check it out later.

```xml
<project>
    <modelVersion/>
    <artifactId>dropwizard-example</artifactId>

    <dependencies>
        <dependency>
            <groupId>com.yammer.dropwizard</groupId>
            <artifactId>dropwizard-core</artifactId>
            <version>0.6.0-SNAPSHOT</version>
        </dependency>
    </dependencies>

    <build>
        <plugins>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-shade-plugin</artifactId>
                <version>1.6</version>
                <configuration>
                    …
                </configuration>
                <executions>
                    <execution>
                        <phase>package</phase>
                        <goals>
                            <goal>shade</goal>
                        </goals>
                        <configuration>
                            …
                        </configuration>
                    </execution>
                </executions>
            </plugin>
        </plugins>
    </build>
</project>
```

First you'll want a Maven POM file. Yeah, I said it. Maven. Deal with it. Presenters don't let attendees use Ant. Dropwizard distributes via a Maven repository. It will also get you up and running in minutes if you use a good Maven-aware IDE. We big believers in IntelliJ IDEA. Also, you're going to want to use the Maven shade plugin for reasons I'll discuss shortly.

```yaml
template: Hello, %s!

defaultName: Stranger

http:
  port: 8080
  adminPort: 8081

  requestLog:
    console:
      enabled: false

    file:
      enabled: true
      currentLogFilename: ./logs/requests.log
      archivedLogFilenamePattern: ./logs/requests-%d.log.gz
      archivedFileCount: 5

logging:
  level: INFO

  console:
    enabled: false

  file:
    enabled: true
    threshold: ALL
    currentLogFilename: ./logs/example.log
    archivedLogFilenamePattern: ./logs/example-%d.log.gz
    archivedFileCount: 5
```

You're going to want a YAML configuration file for your service (JSON is also an option, same structure, different syntax). Dropwizard provides standard configurations for HTTP servers and application logging. In addition, you can add your own configurations. In this case, we see the greeting template and a default name configured in our YAML file. The application responds on port 8080. Metrics responds on port 8081.

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

    @NotEmpty
    private String defaultName = "Stranger";


    public String getTemplate() {
        return template;
    }

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

    public String getDefaultName() {
        return defaultName;
    }

    public void setDefaultName(String defaultName) {
        this.defaultName = defaultName;
    }

    public Template buildTemplate() {
        return new Template(template, defaultName);
    }
}
```

You're going to need a configuration class. The YAML file will get mapped to your class. It should extend the dropwizard Configuration base class, which specifies things like the HTTP server and logging settings. In addition, you get some nice validations for your configuration properties. Unlike most application servers, dropwizard will refuse to start if your configuration doesn't validate.

```java
public class HelloWorldService extends Service<HelloWorldConfiguration> {
    public static void main(String[] args) throws Exception {
        new HelloWorldService().run(args);
    }

    @Override
    public void initialize(Bootstrap<HelloWorldConfiguration> bootstrap) {
        bootstrap.setName("hello-world");
    }

    @Override
    public void run(HelloWorldConfiguration configuration,
                    Environment environment) throws ClassNotFoundException
        final Template template = configuration.buildTemplate();

        environment.addResource(new HelloWorldResource(template));
    }
}
```

Dropwizard runs just like any other Java application, out of main(). Build your dependencies by hand and then hand them to your resource class constructor. Then hand your resource to the dropwizard environment. We've dropped any use of fancy dependency injection frameworks. This is nice because it works nicely in any IDE without any special application server support. Just run the main() in a debugger.

```java
public class Saying {
    @JsonProperty
    private long id;

    @JsonProperty
    @Length(max = 3)
    private String content;

    private Saying() {
        // Jackson deserialization
    }

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

    public long getId() {
        return id;
    }

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

Dropwizard uses JAX-RS (Jersey) under the hood, so you can accept/return Java objects and they'll be handled by a provider. Dropwizard bundles Jackson to process JSON, so use Jackson annotations to describe how to serialize/deserialize the object. You can also use hibernate validator annotations (@Length above) to validate inbound objects. If you don't know JAX-RS, read the JSR sometime. It's a really nice API and JSR 311 is actually readable.

```java
@Path("/hello-world")
@Produces(MediaType.APPLICATION_JSON)
public class HelloWorldResource {
    private static final Logger LOGGER = LoggerFactory.getLogger(HelloWorldResource.class);

    private final Template template;
    private final AtomicLong counter;

    public HelloWorldResource(Template template) {
        this.template = template;
        this.counter = new AtomicLong();
    }

    @GET
    @Timed(name = "get-requests")
    @CacheControl(maxAge = 1, maxAgeUnit = TimeUnit.DAYS)
    public Saying sayHello(@QueryParam("name") Optional<String> name) {
        return new Saying(counter.incrementAndGet(), template.render(name));
    }

    @POST
    public void receiveHello(@Valid Saying saying) {
        LOGGER.info("Received a saying: {}", saying);
    }
}
```

Then you'll want a resource class to model the REST resource. JAX-RS rules apply again here, so use @Path and @GET/@PUT/@POST/@DELETE annotations. Dropwizard supports @Timed annotations to tie back into the Metrics library. In this case, @Timed resource methods get their own timer in the metrics registry showing you how often the method is called (m1/m5/m15 rates) and how long it takes to process requests (p50/p75/p95/p99/p999).

```java
public class TemplateHealthCheck extends HealthCheck {
    private final Template template;

    public TemplateHealthCheck(Template template) {
        super("template");
        this.template = template;
    }

    @Override
    protected Result check() throws Exception {
        template.render(Optional.of("woo"));
        template.render(Optional.<String>absent());
        return Result.healthy();
    }
}
```

Add a healthcheck because, you know, it's nice when your early warning systems tell you things are broken instead of your users telling you via your customer support team.

SHIP IT

That's all…seriously. Seven files…maybe 200 lines of stuff. And you're ready for production. You all used Maven to build your stuff and you set your build up to use the Maven shade plugin, which means your entire application was built as a single JAR file. So ship that and your configuration YAML file to your production host, restart the JVM process and call it a day. You're live.

Optimized for Mean Time To Production

Everything in dropwizard is optimized for mean time to production because, ultimately, code in production is what matters. That's how you make money. You don't make any money while you're figuring out how to set up request logging, how to wire jersey into jetty, how to wire jackson into jersey. In addition, because you ship faster you get information faster. Faster information now means shipping better in the future.

Some dropwizard words of wisdom

We didn't magically wake up one morning and envision this dropwizard thing and how we'd use it. It came from months of living without dropwizard and months more of living with earlier dropwizards. We've iterated on some principles that have worked pretty well for us, so let me share some of these with you today.

Extract, don’t preempt

We didn’t build dropwizard from the start. We built three services using similar (but not identical) principles and then extracted the good parts of each into what became dropwizard 0.0.1. Ultimately, your code needs to advance the business. Make sure you’re doing that first.

Invent as little as possible

Most companies don’t have extensive R&D budgets. Before you go off building your own ORM to support your home grown dependency injection system, look for good off the shelf solutions. In the end, dropwizard is a bunch of glue binding solid, useful libraries into a thoughtful pattern. There’s nothing particularly novel about dropwizard and yet it’s incredibly powerful as a productivity tool for us.

Focus on the business

Value shipping over elegance.

Downloads and further information

Where can you get dropwizard and information about dropwizard?

http://search.maven.org/

com.yammer.dropwizard:
dropwizard-core

Builds of dropwizard are available on Maven central. This includes snapshot and release versions.

https://github.com/codahale/dropwizard

Dropwizard is hosted on github in Coda's personal repository. You'll find the example application and a number of bundles for extending dropwizard in the repository.

dropwizard-user forum on Google Groups

There's a dropwizard-user mailing list on Google Groups. This is a good place to drop in and ask questions about using, extending or modifying dropwizard.

http://dropwizard.codahale.com/

Some really nice documentation is also available online, including a pretty nice getting started guide.

Questions?