RESTful Groovy

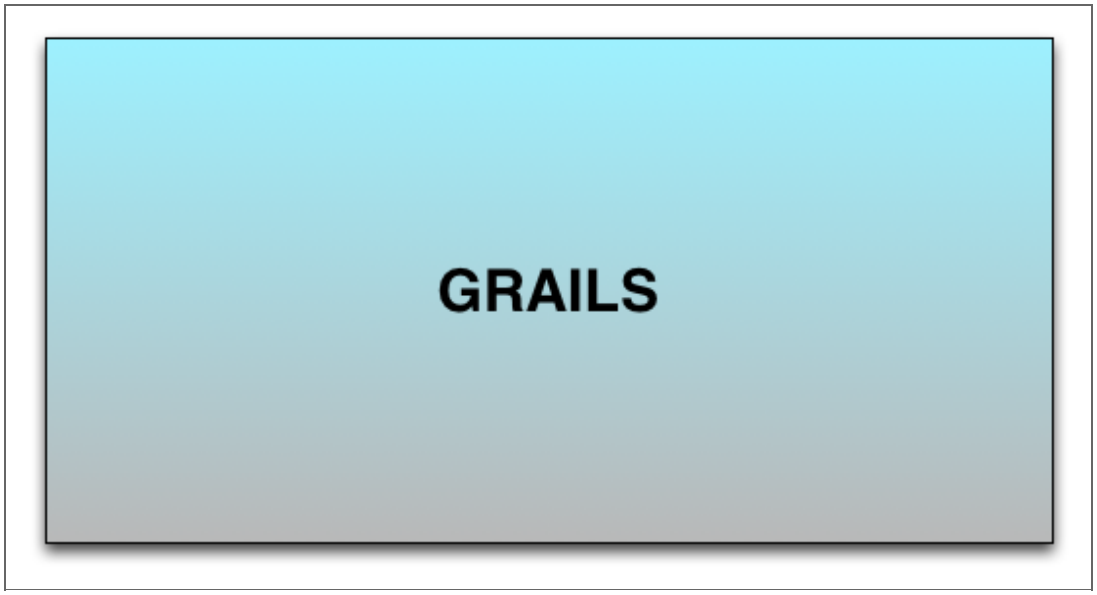
Options for building a Services Oriented Architecture in Groovy

Kyle Boon

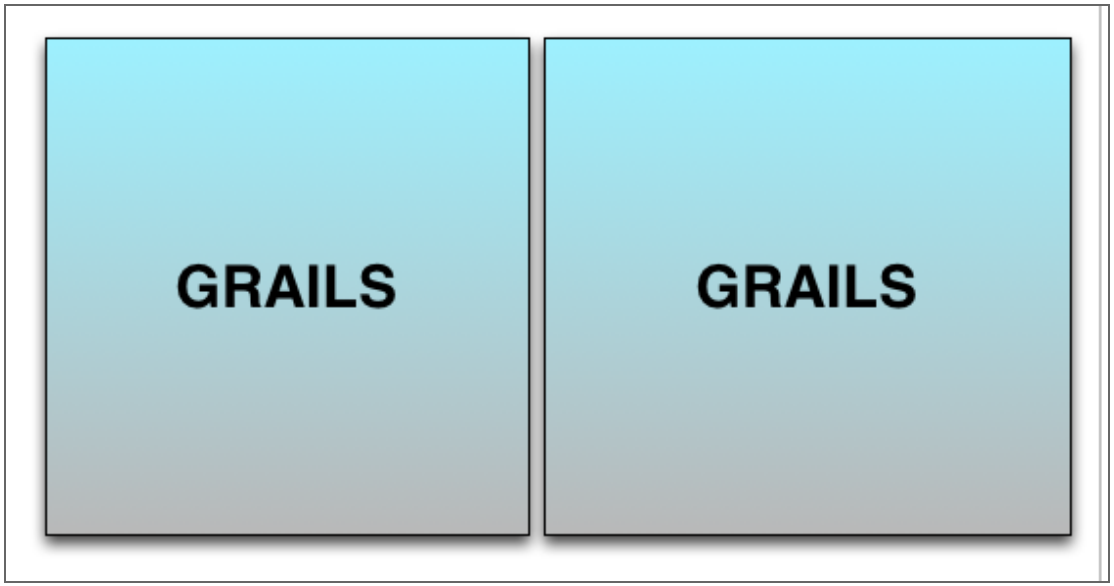
Kyle Boon

Technical Lead @ Bloomhealth
kyle.f.boon@gmail.com
@kyleboon
http://www.kyleboon.org

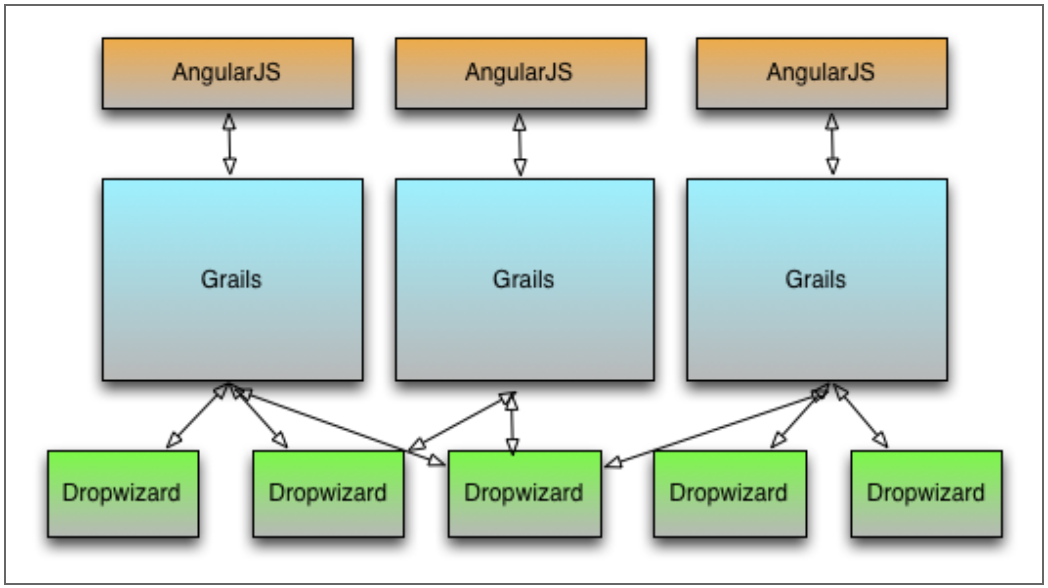
Architecture 2011



Architecture 2012



Architecture 2013



Goals

- Separate components
- Allow for different SLAs for each component
- Be able to combine components in different ways
- Increase developer productivity and ownership

Complete Tool Chain

- Groovy
- Grails for web applications
- Dropwizard for JSON web services
- Gradle for builds
- Swagger for Service Discovery
- Spock for testing
- Gatling for Performace/Load Testing
- Redis for Caching
- RabbitMQ for messaging
- Chef for infrastructure

What is Dropwizard

Dropwizard is a heavily opinionated framework for building web services on the JVM. It is mostly glue around mature java libraries like Jetty, Jersey, Jackson and Guava.

> *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 HTTP+JSON web service in the shortest time possible.*

Who Created it?

@coda
As I've said before, the only reason Dropwizard exists at all is to provide opinions on what a service should be.

It uses fat JARs because I think they work better.

It embeds Jetty because I think that works better.

It uses Jackson because I think that works better.

It uses Jersey because I think that works better.

It has a single YAML configuration file because I think that works better.

It wraps Logback because I think that works better.

How does it work?

Deployed as a fat jar and starts jetty. No need for a container.

```
public static void main(String[] args) throws Exception {
        new ContactsService().run(args)
}
```

Start the server from the command line by running:

```
java -jar contact_dropwizard-shadow-0.1.0-SNAPSHOT.jar
server start dev_config.yml
```

The Service

Services are a collection of bundles, commands, healthchecks, tasks and resources. The service class defines all of the abilities of your application.

```
@Override
public void initialize(Bootstrap bootstrap) {
        bootstrap.name = 'configuration_service'

        bootstrap.addBundle migrationsBundle
        bootstrap.addBundle hibernateBundle
        bootstrap.addBundle(new AssetsBundle('/swagger-ui-1.1.0/', '/swagger'))
        bootstrap.addCommand(new migrationsCommand())
}

@Override
public void run(ContactsConfiguration configuration,
        Environment environment) throws ClassNotFoundException {

        ContactDAO contactDAO = new ContactDAO(hibernateBundle.sessionFactory)
        environment.addResource(new ContactResource(contactDAO))
}
```

The Resource

Resources model what is exposed via your RESTful API. Dropwizard uses Jersey for this so these classes are mostly jersey annotations.

```
@Path('/contacts')
@Produces(MediaType.APPLICATION_JSON)
class ContactResource {
        private final ContactDAO contactDAO

        public ContactResource(ContactDAO contactDAO) {
                this.contactDAO = contactDAO
        }


        @Timed(name = 'createContact')
        @POST
        @UnitOfWork
        public Contact createContact(@Valid Contact contact) {
                return contactDAO.saveOrUpdate(contact)
        }
}
```

The Representation

Your POGOs will be turned into JSON via Jackson. Hibernate Validator lets you specify validation rules.

```
@ToString
@EqualsAndHashCode
class Contact {
        @JsonProperty
        Long id

        @NotNull
        @NotEmpty
        @JsonProperty
        String firstName

        @NotNull
        @NotEmpty
        @JsonProperty
        String lastName

        @Transient
        @JsonIgnore
        def someCrap = false
}
```

Metrics

Yammer Metrics is built in and provides metrics over an administration port (8081). Resources can be annotated with `@Timed` or `@Metered`, and `@ExceptionMetered`.

```
"post-requests" : {
        "type" : "timer",
        "duration" : {
        "unit" : "milliseconds",
        "min" : 0.0,
        "max" : 0.0,
        "mean" : 0.0,
        "std_dev" : 0.0,
        "median" : 0.0,
        "p75" : 0.0,
        "p95" : 0.0,
        "p98" : 0.0,
        "p99" : 0.0,
        "p999" : 0.0
    },
    "rate" : {
        "unit" : "seconds",
        "count" : 0,
        "mean" : 0.0,
        "m1" : 0.0,
        "m5" : 0.0,
        "m15" : 0.0
      }
    }
```

Health Checks

Health Checks are a method to make sure the infrastructure your service depends on are all running. They are accessible on the administration port.

```
public class RedisHealthCheck extends HealthCheck {
    private static final String ECHO = 'checking'
    private final JedisPool jedisPool

    public RedisHealthCheck(JedisPool jedisPool) {
        super('redis')
        this.jedisPool = jedisPool
    }

    @Override
    protected Result check() throws Exception {
        if ( ECHO.equalsIgnoreCase(jedisPool.resource.echo(ECHO)) ) {
            return Result.healthy()
        }

        return Result.unhealthy('ManagedJedisPool connectivity is down!')
    }
}
```

Bundles

Bundles are reusable blocks of behaviour designed to be reused across services. Assets, Hibernate and Liquibase are all implemented as Dropwizard Bundles.

```
@Slf4j
abstract class RedisBundle implements ConfiguredBundle {
    ManagedJedisPool jedisPool

    abstract RedisConfiguration getRedisConfiguration(T configuration)

    @Override
    public final void initialize(Bootstrap bootstrap) {
        log.info('Initializing Redis Bundle')
    }

    @Override
    public final void run(T configuration, Environment environment) throws Exception {
        jedisPool = new ManagedJedisPool(
                new JedisPoolConfig(),
                getRedisConfiguration(configuration).host,
                getRedisConfiguration(configuration).port,
                getRedisConfiguration(configuration).timeout,
                getRedisConfiguration(configuration).db)

        environment.manage(jedisPool)
        environment.addHealthCheck(new RedisHealthCheck(jedisPool))
    }
}
```

Commands

Commands add options to the command line interface of your service. For example the server starts based on the 'server' command. Migrations run based on the 'db migrate' command. You might add your own command for running functional tests or seeding the database.

Tasks

Tasks are run time actions available over the administration port. Dropwizard ships with a garbage collection task. You might want to right a task to clean a cache by key.

```
class FlushRedisTask extends Task {
    ManagedJedisPool managedJedisPool

    protected FlushRedisTask(String name, ManagedJedisPool managedJedisPool) {
        super(name)
        this.managedJedisPool = managedJedisPool
    }

    @Override
    void execute(Map parameters, PrintWriter output) throws Exception {
        Jedis jedis = managedJedisPool.resource
        jedis.flushDB()
    }
}
```

Other Stuff

- Configuration
- Logging
- Hibernate/JDBI
- Clients
- Authentication
- Views

Grails 2.3

- New REST APIs, making it easy to build REST APIs in Grails
- Scaffolding plugin that can generate REST controllers and Async controllers

Declaring the resource as part of the domain

```
package grails.example

import grails.rest.*

@Resource(uri='/contacts', formats=['json', 'xml'])
class Contact {
        String firstName
        String lastName
        String jobTitle
        String phoneNumber
        Address address

    static constraints = {
    }
}
```

Declaring the resource in UrlMappings.groovy

```
"/contacts"(resources:'contact')
```

or

```
"/contacts"(resources:'contact', exclude:['delete'])
```

or

```
"/contacts"(resources:'contact', include:['index', 'show'])
```

Default URL patterns

HTTP Method	URI	Grails Action
GET	/contacts	index
GET	/contacts/create	create
POST	/contacts	save
GET	/contacts/${id}	show
GET	/contacts/${id}/edit	edit
PUT	/contacts/${id}	update
DELETE	/contacts/${id}	delete

Nested Resources

```
"/contacts"(resources:'contact') {
  "/addresses"(resources:"address")
}
```

Default Nested Resrouce URL patterns

HTTP Method	URI	Grails Action
GET	/contacts/{$id}/addresses	index
GET	/contacts/{$id}/addresses/create	create
POST	/contacts/{$id}/addresses	save
GET	/contacts/{$id}/addresses/${id}	show
GET	/contacts/{$id}/addresses/${id}/edit	edit
PUT	/contacts/{$id}/addresses/${id}	update
DELETE	/contacts/{id}/addresses/${id}	delete

Versioning APIs

```
"/contacts/v1"(resources:"contact", namespace:'v1')
"/contacts/v2"(resources:"contacts", namespace:'v2')
```

or

```
"/books"(version:'1.0', resources:"book", namespace:'v1')
"/books"(version:'2.0', resources:"book", namespace:'v2')
```

More options

- Using the RestController super class
- Implementing a REST controller from scratch
- Using a generated REST controller form the scaffold plugin

Hypermedia as the Engine of Application State

Allows the idea of a 'uniform interface' for REST. Services are self documenting and describing in the sense that each resource gives the clients the links necessary to change the state of that resource.

JSON+HAL example

```
HTTP/1.1 200 OK
Server: Apache-Coyote/1.1
Content-Type: application/vnd.books.org.book+json;charset=ISO-8859-1
{
  "_links": {
    "self": {
      "href": "http://localhost:8080/myapp/books/1",
      "hreflang": "en",
      "type": "application/vnd.books.org.book+json"
    }
    "publisher": {
        "href": "http://localhost:8080/myapp/books/1/publisher",
        "hreflang": "en"
    }
  },
  "title": ""The Stand""
}
```

Custom MIME types

```
import grails.rest.render.hal.*
beans = {
    halContactRenderer(HalJsonRenderer, grails.example.Contact)
}


grails.mime.types = [
    all:      "*/*",
    book:     "application/vnd.books.org.book+json",
    bookList: "application/vnd.books.org.booklist+json",
    …
]
```

```
import grails.rest.render.hal.*
import org.codehaus.groovy.grails.web.mime.*
beans = {
    halBookRenderer(HalJsonRenderer, rest.test.Book, new MimeType("application/vnd.books.org.book+json", [v:"1.0"]))
    halBookListRenderer(HalJsonCollectionRenderer, rest.test.Book, new MimeType("application/vnd.books.org.booklist+j
}
```

Non REST stuff in 2.3

- Changed resolver from ivy to maven
- Declarative transaction declaration
- Forked execution mode for speedier test
- Hibernate 4

Metrics gathering

No metrics collection by default for grails applications

- Yammer Metrics Plugin
- New Relic
- Custom code

Dropwizard Main Benefits

- Dead simple deployments
- Built in metrics collection
- Smaller than grails
- No Spring
- More choices made for you

Grails Main Benefits

- Great Community
- Large Library of Plugins
- Less opiniated
- More Conventions
- Supports HATEOAS and XML

Conclusion

There is no spoon.

References

- **Dropwizard User Guide**
- **Dropwizard User Group**
- **https://github.com/codahale/dropwizard**
- **Presentation about Dropwizard @ Yammer**
- **Presentation about Dropwizard @ Simple**
- **Coda Hale and Metrics**
- **Coda Hale and the Programming Ape**
- Grails 2.3 REST improvements

Code we've Looked at

- **Presentation and Example Code**
- **Swagger**
- **Enhanced Groovy Doc**
- **swagger-jaxrs-doclet**

Thanks

A special thanks to the people who actually figured all this stuff out. Including but not limited to:

- Chad Small
- Sairam Rekapalli
- Ryley Gahagan
- Charlie Knudsen
- John Engelman

Kyle Boon

Technical Lead @ Bloomhealth
kyle.f.boon@gmail.com
@kyleboon
http://www.kyleboon.org