RESTful Web Services

Rest

REST APIs allow applications to talk to each other over HTTP

Usually support JSON or XML formats (or both)

HTTP methods GET, PUT, POST or DELETE

- and sometimes PATCH

Domain Classes as REST resources

```
import grails.rest.*


@Resource(uri='/books')
class Book {
   String title


   static constraints = {
        title blank:false
    }
}
```

Default Response

Default Response is xml, to change the default, set `formats` attribute of `@Resource`

```
@Resource(uri='/books', formats=['json', 'xml'])
```

Specify all the formats the resource should expose. All formats available are in

- `Config.groovy -> grails.mime.types`

@Resource

The transformation will automatically register the necessary RESTful URL mapping and create a controller called BookController.

Enables all of the HTTP method verbs on the resource

- Make `@Resource` read only

```
@Resource(uri='/books', readOnly=true)
```

Removing some of the magic

Specify the URL mapping in UrlMappings.groovy

```
"/books"(resources:'book')
```

Results in the following URLs:

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

Url Mappings

Exclude delete and update actions

```
"/books"(resources:'book', excludes:['delete', 'update'])
```

Include only the index and show actions

```
"/books"(resources:'book', includes:['index', 'show'])
```

Single Resource - there is only 1 possible in the system

```
"/book"(resource:'book')
```

Nested Mappings

You can also nest regular URL mappings within a resource mapping

```
"/books"(resources: "book") {
    "/publisher"(controller:"publisher", method:"GET")
}
```

Maps `/books/1/publisher` to PublisherController.index

Linking to REST resources

```
<g:link resource="${book}">My Link</g:link>
```

Currently g:link does not work with DELETE action and most browsers do not support sending the DELETE method directly

The best way to accomplish this is to use a form submit:

```
<form action="/book/2" method="post">
    <input type="hidden" name="_method" value="DELETE"/>
 </form>
```

Workshop

Make the User domain a REST resource

Verify its working by going to

`http://localhost:8080/questionApp/users.json`

Versioning REST resource

Using the URI to version APIs

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

That will match the following controllers:

```
package myapp.v1

class BookController {
    static namespace = 'v1'
}

package myapp.v2

class BookController {
    static namespace = 'v2'
}
```

URI Versioning

discouraged in favour of Hypermedia

This approach has the disadvantage of requiring two different URI namespaces for your API.

Accept-Version header

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

Clients pass which version they need

```
> curl -i -H "Accept-Version: 1.0" -X GET
                        http://localhost:8080/myapp/books
```

Still need 2 namespaces

Hypermedia Versioning

Create a custom Mime Type for your resource that includes a version parameter

```
grails.mime.types = [
    all: '*/*',
    book: "application/vnd.books.org.book+json;v=1.0",
    bookv2: "application/vnd.books.org.book+json;v=2.0",
    …
]
```

Hypermedia Versioning

Register the new Mime types in

`conf/spring/resources.groovy`

```
import grails.rest.render.json.*
import org.codehaus.groovy.grails.web.mime.*

beans = {
    bookRendererV1(JsonRenderer, myapp.v1.Book, new
      MimeType("application/vnd.books.org.book+json", [v:"
1.0"]))

    bookRendererV2(JsonRenderer, myapp.v2.Book, new
      MimeType("application/vnd.books.org.book+json", [v:"
2.0"]))
}
```

Calling Hypermedia Version

Using the Accept header you can specify the version using the Mime Type

```
curl -i -H
    "Accept:application/vnd.books.org.book+json;v=1.0"
    -X GET http://localhost:8080/myapp/books
```

Rest Controller Super Class

Going beyond @Resource

```
class BookController extends RestfulController {
    static responseFormats = ['json', 'xml']
    BookController() {
        super(Book)
    }

    @Override
    protected Book queryForResource(Serializable id) {
        Book.where {
            id == id && author.id = params.authorId
        }.find()
    }
}
```

Roll Your Own REST Controller

Don’t have to use RestfulController

Create your own. For examples look at the scaffolding that is created by

```
grails generate-controller
```

The big win is the respond method

Modifying the Response Renderers

Custom Renderers allow you to include/exclude properties from a domain

Register custom renderers in

`conf/spring/resources.groovy`

```
beans = {
    bookRenderer(XmlRenderer, Book) {
        includes = ['title']
        excludes = ['isbn']
    }
}
```

Modifying the Response Converters

Grails' Converters have a registered `ObjectMarshaller`

Register custom marshaller in `BootStrap.init`

```
JSON.registerObjectMarshaller(Book) {
                        def        map        =        [:]
                      map['title']       =       it.title
                      map['auth']        =       it.author
    return map
}
```

Why Renderers and Converters?

Renderers use Converters under the covers

- as JSON, as XML

When implementing a custom renderer you could use Jackson, Gson, or any Java library

Converters are tied to Grails' own marshalling implementation

Workshop

Modify the response for the User domain to only show the username, firstName, lastName and email

You can do either a renderer or marshaller

Workshop Solution

Converter

```
JSON.registerObjectMarshaller(User) {
  def map= [:]
  map['username'] = it.username
  map['firstName'] = it.firstName
  map['lastName'] = it.lastName
  map['email'] = it.email
  return map
}
```

Workshop Solution

Renderer

```
import grails.rest.render.xml.*

beans = {
  userRenderer(XmlRenderer, com.opi.User) {
    includes = ['firstName', 'lastName', 'username', 'email']
  }
}
```