DATUM ACADEMY

MBDS course:
« Native Mobile Programming »
Part 1
Grails Framework and REST API

Gregory Galli

Freelance Teacher and Projet Manager at University of Nice Sophia Antipolis (UCA) - France

www.mbds-fr.org

DATUM ACADEMY

Module 1:
Back-end development

Groovy – Grails – Project summary

eMBDS

DATUM ACADEMY

Groovy Language

Overview

eMBDS

Context

- Groovy, an alternative to Java
- Grails Framework
  - Philosophy
  - Advantages
  - Architecture

Groovy

- Created in 2003
- Object oriented language for the Java platform
- Inspirations
- Superset of Java
  - Inherit Java strong points
  - Enhance Java
  - Can use Java libraries

Groovy

- Syntax
- Functionalities
  - Closures
  - Easy XML / JSON manipulation
  - Flexible and complete collection system
  - Dynamic typing (def)
  - Native support for regular expression
  - Native support for markup languages
  - String interpolation
  - Safe navigation operator (?.)
  - Can be executed as a script
  - And so on …

Groovy – Dynamic typing

- Can use static or dynamic typing
  - “def” keyword
- Type checking at runtime
- Pros
  - More flexibility
  - Quicker coding
  - A variable can have his type changed
- Cons
  - Need to be more rigorous
  - Can lead to unexpected behaviours

Groovy – Assertion

- Use of “assert”(assertions) to check some conditions are met

```
// Define a variable
def subject = "Mooc"

// Confirm the variable value
assert subject == "Mooc"

// Produce traces on error
assert subject == "Mook"
```

```
// Prints out
"""
Caught: Assertion failed:

assert subject == "Mook"
       |       |
       'Mooc'  false
"""
```

Groovy – Closures

- Anonymous
- Can take arguments
- Return a value
- Can be assigned to a variable
- Can be stored into a variable
- Out of the surrounding scope
- Declaration

```
{ [ closureParams -> ] statements }
```

- Optional parameters
  - Comma separated
- Offers flexibility for simple things

Groovy – Closures – Declaration

- Often used with implicit parameter “it”

```
// Prints out provided parameter, classic declaration
def closureA = { def var -> println var }

//  Equivalents
def closureB = { println it } // implicit parameter
def closureC = { it -> println it } // named parameter, no type defined
def closureD = { def it -> println it } // named parameter, dynamic typed variable
def closureE = { String it -> println it } // named parameter, typed variable
```

Groovy – Closures – Call

- Different ways to call a Closure

```
// No param closure
def closure = { "mooc" }

// Can be called like this
assert closure() == "mooc"
assert closure.call() == "mooc"

// Closure with param
def closureWithParam = { it }

// Can be called this way
assert closureWithParam("closure") == "closure"
assert closureWithParam.call("closure") == "closure"
```

Groovy – Closures – Examples

```groovy
// Return true if the provided parameter is odd
def isOdd = { int i -> i % 2 != 0 }
assert isOdd(3) == true
assert isOdd(2) == false

// Return true if the provided parameter is even
def isEven = { int i -> i % 2 == 0 }
assert isEven(3) == false
assert isEven(2) == true

// Return the surface considering the parameters
def rectSurface = { int w, h -> w * h}
assert rectSurface(2, 3) == 6
```

Groovy – Collections

- Various collection types
  - Lists
  - Sets
  - Maps
  - Ranges
- A lot of available methods
- Simple syntax
  - Declaration
  - Usage

Groovy – Collections – List

- Ordered
  - 0-based index
- Can hold duplicates
- Can hold various types
- Declaration & manipulation

Groovy – Collections – List

- List Declaration

```
// Empty List
List emptyList = []

// Intergers List
List integerList = [1, 2, 3, 4]

// Mixed List
List mixedList = ['Mooc', 101, [1, 2], 1.01]
```

Groovy – Collections – List

- Manipulation

```
// Declare new List
List list = ['Paris', 'London', 'Berlin']

// Checking List data
assert list.size() == 3
assert list.isEmpty() == false
assert list.contains('Berlin') == true
```

Groovy – Collections – List

- Manipulation

```
// Declare new List
List list = ['Paris', 'London', 'Berlin']

// Editing current List
list.add('Rome')
assert list == ['Paris', 'London', 'Berlin', 'Rome']
assert list.pop() == 'Paris' // would be the last item < groovy 2.5
assert list == ['London', 'Berlin', 'Rome']

// This method is creating a new List
assert list.reverse() == ['Berlin', 'London', 'Paris']
```

Groovy – Collections – Set

- Not ordered
- Can not hold duplicates
- Can hold various types
- Declaration & manipulation

Groovy – Collections – Set

- Declaration

```
// Empty Set
Set emptySet = []

// Set from List
def list = ['Paris', 'London', 'Berlin']
def listToSet = list.toSet()
def listAsSet = list as Set
```

Groovy – Collections – Map

- Associative Array
- Keys are strings by default
- Not ordered
- Very handy to handle structured data

Groovy – Collections – Map

- Declaration

```
// Empty Map
def emptyMap = [:]

// Classic Map
Map map = [france: 'Paris', uk: 'London', germany: 'Berlin', italy: 'Rome']
```

Groovy – Collections – Range

- Quick way to declare list of sequential values
- Inclusive by default

```
// Classic Range
def inclusiveRange = 11..15
assert inclusiveRange == [11, 12, 13, 14, 15]

// Exclusive notation
def exclusiveRange = 11..<15
assert exclusiveRange == [11, 12, 13, 14]
```

Groovy – Collections – Generics

- Handy Methods for Collections / Iterables
  - plus : Combine collections
  - minus : Subtract collections
  - each / eachWithIndex : Iterate over Collection item
  - collect : Iterate over elements to transform them
  - find / findAll / findIndexOf : Filter & search
  - flatten : Self explanatory
  - split : Divide a collection

Groovy – Collections – Plus / Minus

- Combine / Subtract collections
- Create a new collection

```
// Declare new List
List list = ['Paris', 'London', 'Berlin']

// These are creating new Lists
assert list.minus('Paris') == ['London', 'Berlin']
assert list.plus('Rome') == ['Paris', 'London', 'Berlin', 'Rome']
assert list.size() == 3
```

Groovy – Collections – Each

- Iterate over collection items

```
List list = ['Paris', 'London', 'Berlin']

// Iterate and print value
list.each {
    println it
}

// Prints out
London
Berlin
Rome
```

Groovy – Collections – Each

- Works with all collection type

```
// Declare new Map
Map map = [france: 'Paris', uk: 'London', germany: 'Berlin', italy: 'Rome']

// Iterate and print
map.each {
    println "The capital of $it.key is $it.value"
}

// Strict equivalent
map.each { key, value ->
    println "The capital of $key is $value"
}

// Prints out
The capital of france is Paris
The capital of uk is London
The capital of germany is Berlin
The capital of italy is Rome
```

Groovy – Collections – EachWithIndex

- With index if you need to keep track

```
// With index
map.eachWithIndex { key, value, index ->
    println "[$index] The capital of $key is $value"
}

// Prints out
"""
[0] The capital of france is Paris
[1] The capital of uk is London
[2] The capital of germany is Berlin
[3] The capital of italy is Rome
"""
```

Groovy – Collections – Collect

- Iterate over items and transform them

```
// Collect to edit
def newList = map.collect {
    key, value ->
        [ key.toUpperCase(), value ]
}

assert newList == [
    ['FRANCE', 'Paris'],
    ['UK', 'London'],
    ['GERMANY', 'Berlin'],
    ['ITALY', 'Rome']]
```

Groovy – Collections – Collect

- Can add the collection you want to build as a parameter

```
def newOtherList = []
map.collect(newOtherList) {
    key, value ->
        [ key.toUpperCase(), value ]
}

assert newOtherList == [
    ['FRANCE', 'Paris'],
    ['UK', 'London'],
    ['GERMANY', 'Berlin'],
    ['ITALY', 'Rome']]
```

Groovy – Collections – Find

```
// Declare new list
def intList = [1, 2, 3, 4, 5, 6, 11, 15]

// Returns the first matching element
assert intList.find{ it  > 5 } == 6

// Returns all matching elements
assert intList.findAll{ it  > 5 } == [6, 11, 15]
// FindAll will always return a list even if no matching element
assert intList.findAll{ it > 20 } == []
```

Groovy – Collections – Other

```groovy
// Declare new list
def multiDimList = [1, [2, 3], 4, [5, 6, [7, 8], 9]]

// Flatten any depth collection
def flatList = multiDimList.flatten()
assert flatList == [1, 2, 3, 4, 5, 6, 7, 8, 9]

// Split returns a list with two batches
// First batch contains the matching elements
// Second batch contains the rejected values
assert flatList.split { it < 6} == [[1, 2, 3, 4, 5], [6, 7, 8, 9]]
```

Groovy – Syntax

DATUM ACADEMY

- May be surprising, Can omit :
  - Brackets before methods call
  - Dots between successive calls

```
// Declare methods
def turn(def direction) {
    println direction
    return this
}

def then(def direction) {
    println direction
}
```

```
// Declare variables
def left = "west"
def right = "east"

// Equivalent to turn(left).then(right)
turn left then right

// Prints out
"""
west
east
"""
```

DATUM ACADEMY

Groovy on Rails : Grails

Overview

eMBDS

Grails – Context

- Created in 2005 (Graeme Rocher)
- Open source agile full stack development framework
- Based on the Groovy language (Grails → Groovy on Rails (agile))
- Built on top of Spring Boot
- Seamless integration with Java
- Built for the JVM

Grails – Concepts & Advantages

- DRY : Don’t repeat yourself
- Convention over configuration
- Model driven architecture
- Easy prototyping (scaffolding)
- Plugins
  - Spring security core

Grails – Concepts & Advantages

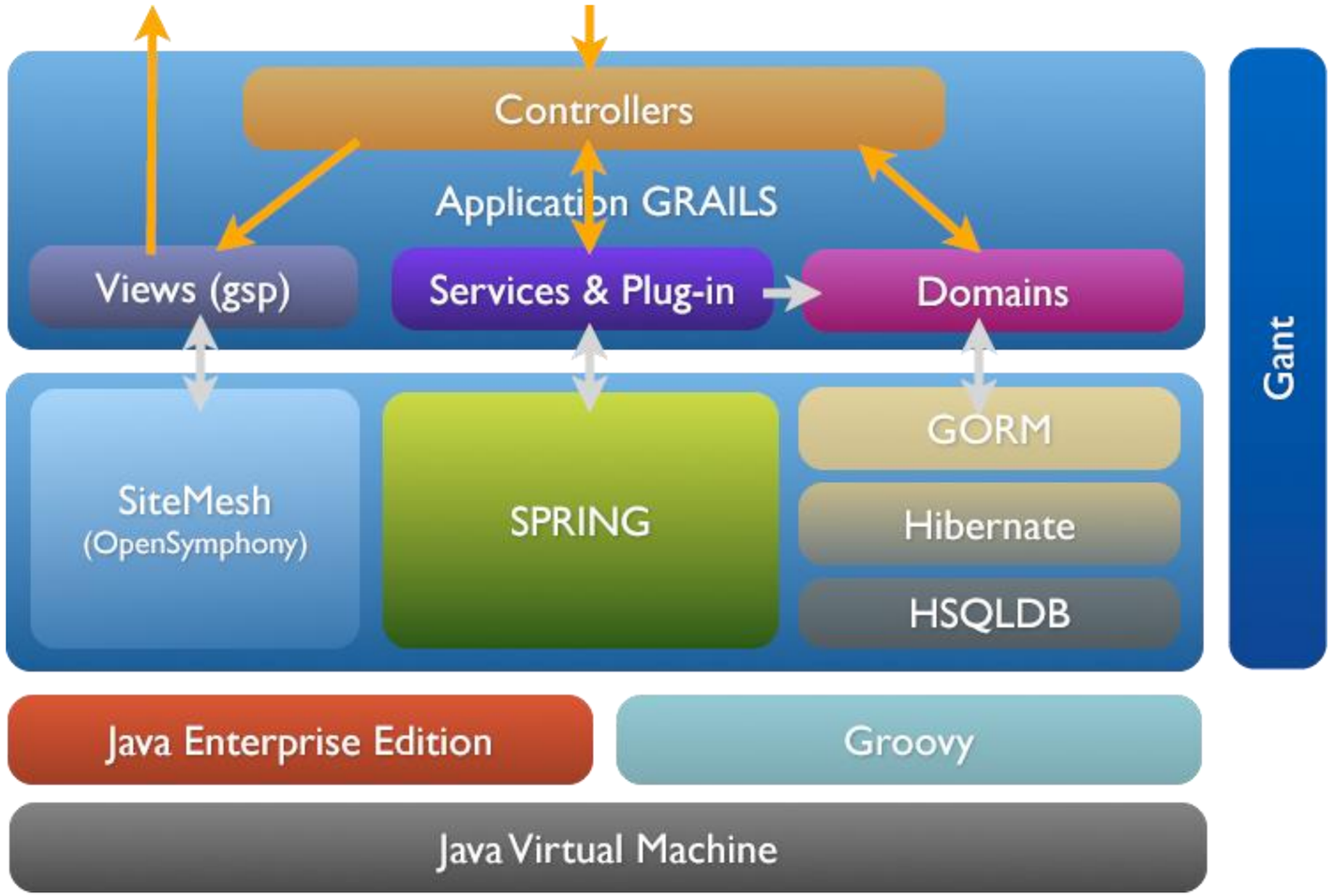
DATUM ACADEMY

- Vast and helping community
- View technologies – Mainly for HTML / JSON rendering
- Asynchronous capabilities – Promises / Events – RxJava
- Domain-specific languages (DSLs)
  - Validation
  - Querying
  - Rendering

Grails – Concepts & Advantages

- GORM
  - Hibernate (SQL)
  - MongoDB
  - Cassandra
- Embedded Application Server
  - Tomcat (Plugin)
- In-memory database for development stage (H2)
- Many IDE available
  - Intellij IDEA – Ultimate edition (best in my opinion, free student licence)
  - Eclipse
  - Sublime

Grails – Architecture



Grails – Used by

Grails – Plugins

- Default
  - Application Server : Tomcat / Glassfish
  - Database : H2 / Hibernate
  - Web resources handling : Assets
- Useful
  - Security : Spring Security Core / REST
  - Payment Service Provider : Paypal / Stripe / …
  - Mail handling
  - Cron Job handling
- Check before using
- Restrict usage to prevent performance issues

DATUM ACADEMY

Grails

Modelling, Hibernate, GORM & Querying

eMBDS

Grails – Modelling

- Main focus when building a Grails application
  - Base attributes
  - Constraints
  - Relations
- Scalability
- Mappings
- Using GORM DSL

Grails – Modelling

- Attributes defined with any base type

```
class BaseTypes {
    byte[]      grails_byte_array
    String      grails_string
    byte        grails_byte
    Character   grails_character
    Integer     grails_integer
    Double      grails_double
    Float       grails_float
    Long        grails_long
    Date        grails_date
    Boolean     grails_Boolean
}
```

- Complete list : Hibernate documentation BasicTypes

Grails – Modelling

- Attributes defined with any base type

SHOW COLUMNS FROM BASE_TYPES;

FIELD	TYPE	NULL	KEY	DEFAULT
ID	BIGINT(19)	NO	PRI	(NEXT VALUE FOR PUBLIC.SYSTEM_SEQUENCE_05AFABDC_454A_4743_B392_3ED0262E2B70)
VERSION	BIGINT(19)	NO		NULL
GRAILS_BOOLEAN	BOOLEAN(1)	NO		NULL
GRAILS_BYTE_ARRAY	VARBINARY(255)	NO		NULL
GRAILS_FLOAT	DOUBLE(17)	NO		NULL
GRAILS_DATE	TIMESTAMP(26)	NO		NULL
GRAILS_BYTE	TINYINT(3)	NO		NULL
GRAILS_LONG	BIGINT(19)	NO		NULL
GRAILS_CHARACTER	CHAR(255)	NO		NULL
GRAILS_DOUBLE	DOUBLE(17)	NO		NULL
GRAILS_STRING	VARCHAR(255)	NO		NULL
GRAILS_INTEGER	INTEGER(10)	NO		NULL

Grails – Modelling

- Relations
  - One-to-One
  - One-to-Many
  - Many-to-Many

```
class User {
    String username

    // One-to-One
    Address address

    // One-to-Many (Message) and Many-to-Many (Group)
    static belongsTo = [UserGroup]
    static hasMany = [messages: Message, groups: UserGroup]
}
```

Grails – Modelling

- Unidirectional / Bidirectional
  - Strong impact on performance
  - Bidirectional when needed only
- Properties added by GORM
  - `id` : Long – Primary key – Auto increment
  - `version` : Long – Used to maintain data consistency

SHOW COLUMNS FROM BASE_TYPES;

FIELD	TYPE	NULL	KEY	DEFAULT
ID	BIGINT(19)	NO	PRI	(NEXT VALUE FOR PUBLIC.SYSTEM_SEQUENCE_05AFABDC_454A_4743_B392_3ED0262E2B70)
VERSION	BIGINT(19)	NO		NULL

Grails – Hibernate

- Open source persistence Framework
- Usable in web or classic application environment
- Replace a classic DAL (Data Access Layer)
- Provide high level object method access
- Usable with most relational DBMS
- Used in Grails environment through GORM

Grails – GORM

- Grails Object Relational Mapping
- Data Access Framework
  - Quick data access code
- GORM for …
  - Hibernate (SQL)
  - MongoDB (Document oriented)
  - Cassandra (NoSQL)
  - …
- Illusion of an object oriented database
- Matching the object with a relational database

Grails – Hibernate / GORM – Pros

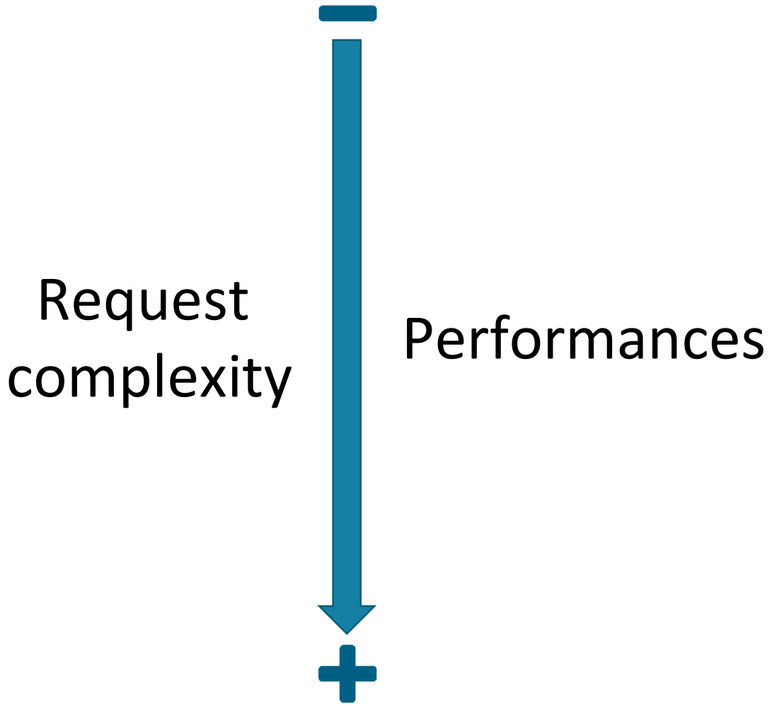
- Accessing high level object instead of tables
- Easy and quick setup and usage
  - No need to setup tables or tables properties
  - Dynamic finders for prototyping or simple request
- Transaction handling
- Same syntax whatever the DBMS
  - Seamless & painless transition if DBMS migration

Grails – Hibernate / GORM – Cons

- Can be tough in complex projects
- Adding a new layer hinders performances
- Abstraction
  - Does not mean you should not understand what’s behind
  - Without a global understanding, will encounter blocking points

Grails – GORM – Querying

- Basic CRUD
- Dynamic Finders
- Where Queries
- Criteria Queries
- Hibernate Query Language (HQL)



Conclusion

- Covered the basics
  - Groovy
  - Grails
- Next steps, go deeper into Grails
  - Project structure
  - Modelling
  - Data handling
  - Security
- Dive into back office development

DATUM ACADEMY

Module 2:
Back-end development

Project Structure – Querying – Specifics – Scaffolding – Security

eMBDS

DATUM ACADEMY

Grails – Project Structure

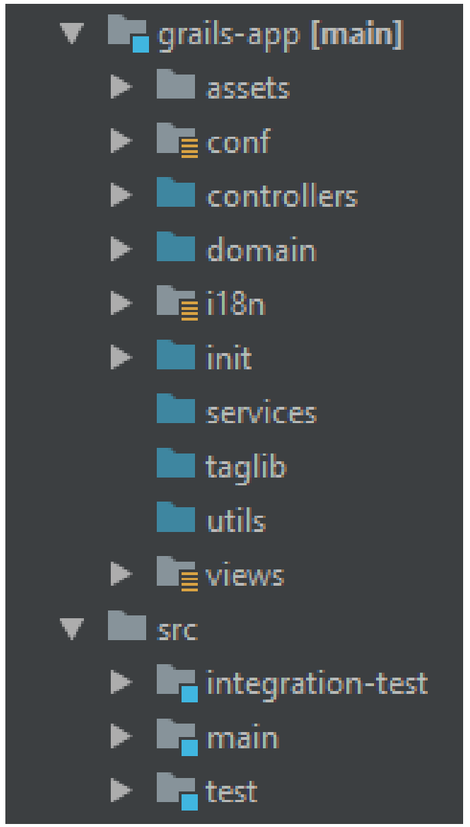
Assets, Configuration, Plugins & Tools

eMBDS

Grails – Project Structure

- Convention over configuration
  - File names and location is key

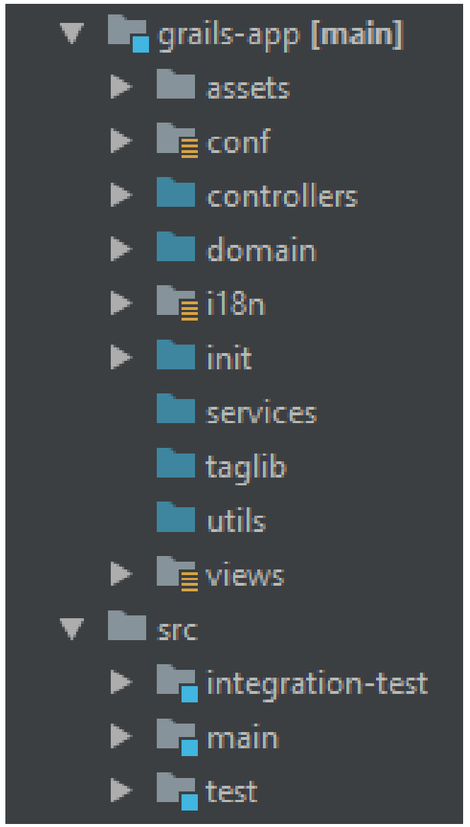
- `grails-app` : top level directory containing groovy sources
  - `assets` : front-end ressources, managed by Asset Pipeline Plugin
  - `conf` : configurations sources
  - `controllers` : web controllers
  - `domain` : domain classes
  - `i18n` : internationalization files
  - `init` : contains `BootStrap.groovy` file (init data)
  - `services` : service layer
  - `taglib` : tag libraries
  - `utils` : Grails utilities
  - `views` : contains Groovy Server Pages and JSON Views
- `src / …`



Grails – Project Structure

- Convention over configuration
  - File names and location is key

- `grails-app` : top level directory containing groovy sources
  - [...]
- `src / integration-test` : self explanatory
- `src / main` : groovy sources
- `src / test` : unit tests



Grails – Asset Pipeline Plugin

- Managing and processing static assets
- Available since Grails 2.4
- Process and minify CSS and Javascript files
- Built-in taglibs for proper ressources usage
- Can be extended to compile custom statics
  - Coffeescript
  - LESS
  - SASS
  - …

Grails – Asset Pipeline Plugin

- Default manifest files with directives
- Default structure
  - `grails-app/assets/javascript`
  - `grails-app/assets/images`
  - `grails-app/assets/stylesheets`

application.js

```
1   // This is a manifest file that'll be compiled into application.js.
2   //
3   // Any JavaScript file within this directory can be referenced here using a relative path.
4   //
5   // You're free to add application-wide JavaScript to this file, but it's generally better
6   // to create separate JavaScript files as needed.
7   //
8   //= require jquery-2.2.0.min
9   //= require bootstrap
10  //= require_tree .
11  //= require_self
```

Grails – Asset Pipeline Plugin – Serving Assets

- Served by application server during development & tests
- Should be externalized for production
  - Dedicated external web server
  - CDN

```
# grails-app/conf/application.yml
environments:
    # for specific environment
    production:
        grails:
            # specify
            assets:
                # the base url
                url: http://cdn.example.com/
                # the base path
                storagePath: /var/www/dedicated/web/server
```

Grails – Asset Pipeline Plugin – Taglibs

- Use Taglibs to create proper references

```
<head>
    %{-- Javascript / CSS inclusion in pages --}%
    <asset:javascript src="application.js"/>
    <asset:stylesheet src="application.css"/>
</head>
<body>
    %{-- Simple reference to asset --}%
    <asset:image src="logo.png"/>
</body>
```

Grails – Configuration

- Not mandatory
- Mainly useful for overrides
- Build configuration → `build.gradle`
- Runtime configuration → `grails-app/conf/application.yml`
- Can be externalized

```yaml
# file : grails-app/conf/application.yml
server:
    # Change the application server port this way
    port: 8082
    # Change the context path like this
    contextPath: /myProjectName
```

Grails – Data Sources

- Provide configuration for database link
  - Default settings are completely fine to start with
- Environment specific
- Default database : H2
- Swap database
  - Provide new driver (dependency / manual JAR)
  - Change the driverClassName
  - Change the username & password if needed
  - Edit the JDBC url
  - Restart project
  - Done !

Grails – URL Mappings

- Defined in `grails-app/controllers/package/UrlMappings.groovy`

- Mapping URL
  - Controllers
  - Actions
  - REST Resources

- No real convention
  - defined and editable

```
class UrlMappings {
    static mappings = {
        "/$controller/$action?/$id?(.$format)?"{
            constraints {
                // apply constraints here
            }
        }

        "/"(view:"/index")
        "500"(view:'/error')
        "404"(view:'/notFound')
    }
}
```

Grails – URL Mappings

- View redirection, static views

```
"/"(view:"/index")
"500"(view:'/error')
"404"(view:'/notFound‘)
```

- Redirect on controller / action

```
// default action : index
"/"(controller: "main")
// equivalent to
"/"(controller: "main", action: "index")
```

- Redirect on REST Resources

```
"/users"(resources: "user")
```

Grails – URL Mappings

- Can use
  - Embedded variables
  - Optional variables
  - Regular expressions

```
// matching url will be map declared variables
// will match :          /promo/product/107
"/$category/product/$id"(controller: "product")

// use ? to make variables optionals
// will match :          /promo/product/107
// will also match :     /promo/product
"/$category/product/$id?"(controller: "product")
```

Grails – URL Mappings

- Can use
  - Embedded variables
  - Optional variables
  - Regular expressions

```
// use regular expressions to constraint variables values
// will match :          /promo/product/107
// wont match :          /promo/product/abc
"/$category/product/$id"
{
    controller = "product"
    constraints{
        id(matches: /\d+/)
    }
}
```

Grails – Bootstrap

- Located under « init » directory
- Contains 2 closures
  - “init” closure called at server start
  - “destroy” closure called when
    - The servlet instance has been taken out
    - A timeout period has passed
- Mainly used to setup project essential data
  - Define main Users (admin), Roles (security)
- Useful in development phase
  - Wipe database data on restart
  - Reset base data with Bootstrap at launch
  - Always save with options
    - Flush : to trigger the persistence operations immediately
    - FailOnError : to stop the application server start if something goes wrong

Grails – Plugins

- Defined in `build.gradle`
- Huge plugin base
- Use sparingly
- Default plugins
  - Hibernate
  - Assets
  - Tomcat
  - …
- Useful
  - Spring Security Core
  - Spring Security REST
  - Connectors
  - …

DATUM ACADEMY

Grails – Modelling

Focus on Modelling, Inheritance, Associations, Constraints & Mapping

eMBDS

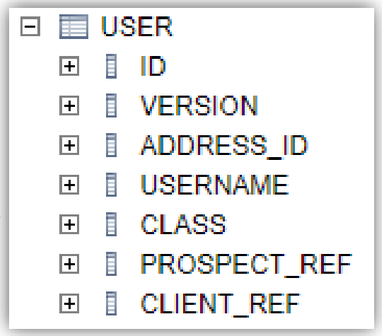
Grails – Modelling – Inheritance

- Avoid as much as possible
  - Performance issues
- Default behaviour
  - Gather all properties in the same table
  - Can’t force properties to be “not null” at database level

```
class Client extends User {
    String clientRef
}
```

```
class User {
    String username
}
```

```
class Prospect extends User {
    String prospectRef
}
```



Grails – Modelling – Inheritance

```
new User(username:"user", address: new Address()).save()
new Client(username:"client", clientRef: "clientRef", address: new Address()).save()
new Prospect(username:"prospect", prospectRef: "prospectRef", address: new Address()).save()
```

SELECT * FROM USER;

ID	VERSION	ADDRESS_ID	USERNAME	CLASS	CLIENT_REF	PROSPECT_REF
1	0	1	user	grailsproject.User	*null*	*null*
2	0	2	client	grailsproject.Client	clientRef	*null*
3	0	3	prospect	grailsproject.Prospect	*null*	prospectRef

Grails – Modelling – Inheritance

SELECT * FROM USER;

ID	VERSION	ADDRESS_ID	USERNAME	CLASS	CLIENT_REF	PROSPECT_REF
1	0	1	user	grailsproject.User	*null*	*null*
2	0	2	client	grailsproject.Client	clientRef	*null*
3	0	3	prospect	grailsproject.Prospect	*null*	prospectRef

- Non matching properties set to `null`
- `Class` field added to keep track
- Default behavior : `table-per-hierarchy`
- Override : `table-per-subclass` with `mapping` DSL

Grails – Modelling – Inheritance

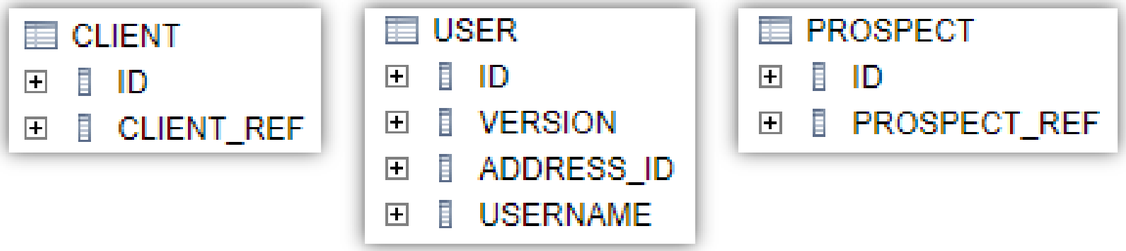
```
class Client extends User {
    String clientRef
}
```

```
class User {
    String username

    static mapping = {
        tablePerHierarchy false
    }

}
```

```
class Prospect extends User {
    String prospectRef
}
```



Grails – Modelling – Inheritance

SELECT * FROM PROSPECT;

ID	PROSPECT_REF
3	prospectRef

SELECT * FROM USER;

ID	VERSION	ADDRESS_ID	USERNAME
1	0	1	user
2	0	2	client
3	0	3	prospect

SELECT * FROM CLIENT;

ID	CLIENT_REF
2	clientRef

- No useless fields
  - Saves disk space
- Each request requires table joins
  - Computing cost is higher
  - Disk access cost is higher
- No systematic better choice → decide according to the situation

Grails – Modelling – Inheritance

- Seamless when getting objects

```
// Retrieve all Users an render as JSON
render User.list() as JSON
```



```
[
  - {
        id: 1,
      - address: {
            id: 1
        },
        username: "user"
    },
  - {
        id: 2,
      - address: {
            id: 2
        },
        clientRef: "clientRef",
        username: "client"
    },
  - {
        id: 3,
        prospectRef: "prospectRef",
      - address: {
            id: 3
        },
        username: "prospect"
    }
]
```

Grails – Modelling – Associations

- Define how domain class interact
- Unidirectional by default
- Available setups
  - `Many-to-one`
  - `One-to-One`
  - `One-to-Many`
  - `Many-to-Many`
- Complex cases
  - Multiple properties of the same type
  - Self referential properties
  - Deal with GORM incorrect guesses

Grails – Associations – Many-to-One

- Simplest case
- Unidirectional
- Multiple `Books` may reference a same `Author`
- `Book` references a single `Author` instance

```
// Book.groovy
class Book {
    String title
    Author author
}

// Author.groovy
class Author {
    String name
}
```

Grails – Associations – Many-to-One

- Persist new object with Many-to-one association

```
// Persist a new Book with a new Author
new Book(title: "title", author: new Author(name: "author's name")).save()

// Persist a new Book with an existing Author
new Book(title: "title", author: Author.get(1)).save()

// Persist a new Author, assign it to new Books
def authorInstance = new Author(name: "new author").save()
new Book(title: "title", author: authorInstance).save()
new Book(title: "other title", author: authorInstance).save()
```

Grails – Associations – One-to-One

- « Kind of » One-to-one
- Address belongs to User
  - Behave like a One-to-one
- « Belongs to » implies
  - Strong dependency from Address to User
  - User cascade saves **and deletes**
  - Address can’t exist on its own

```groovy
// User.groovy
class User {
    String username
    Address address
}

// Address.groovy
class Address {
    static belongsTo = [user: User]
}
```

Grails – Associations – One-to-One

```groovy
// User.groovy
class User {
    String username
    Address address
}

// Address.groovy
class Address {
    static belongsTo = [user: User]
}
```

```groovy
// Save user AND address instance
def userInstance = new User(username: "username")
userInstance.address = new Address()
userInstance.save()

// Delete user AND associated address
userInstance.delete()
```

Grails – Associations – One-to-One

- Proper `One-to-one`
- Foreign key in the weak side of the association (`Address`)
- `hasOne` always bidirectional
- Without « `belongsTo` »
  - `Cascade save`
- With « `belongsTo` »
  - `Cascade delete`

```groovy
// User.groovy
class User {
    String username
    static hasOne = [address: Address]
}

// Address.groovy
class Address {
    // Simple reference
    // won't cascade deletes
    User user
}
```

Grails – Associations – One-to-One

- Persist new object with One-to-one association

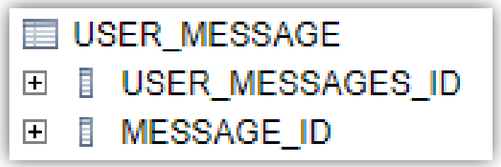
```
// Persist a new User with a new Address
// No need to explicitly define Address > User attribute
new User(username: "username", address: new Address()).save()

// User won't be created, Address is mandatory
new User(username: "username").save()

// Address holds reference to User, cannot create Address on its own
// Address won't be created
new Address().save()
```

Grails – Associations – One-to-Many

- User has many instances of Message
- Will create a join table




```groovy
// User.groovy
class User {
    String username
    static hasMany = [messages: Message]
}

// Message.groovy
class Message {
    String content
}
```

- Default names for table and columns customizable with mappedBy
- Default cascading behaviour
  - Cascade saves and updates
  - Casdade deletes if Message belongs to User
- User's property : « messages » will be a « Set »
  - Can be overridden

Grails – Associations – One-to-Many

```
// Instantiate new User
def userInstance = new User(username: "username")

// Add new Message to userInstance
userInstance.addToMessages(new Message(content: "message content"))

// Adding multiple Messages
["message","other message", "another message", "..."].each {
    userInstance.addToMessages(new Message(content: it))
}

// Persist userInstance will save User and associated Message
userInstance.save()

// Will delete userInstance without deleting Message (no belongsTo)
userInstance.delete()
```

Grails – Associations – One-to-Many

- Fetching strategy
  - Default : Lazy
  - Can be overridden
- Eager fetching
  - Need to be very carefull
  - Can introduce cyclic queries
  - Only when needed

```groovy
// User.groovy
class User {
    String username
    static hasMany = [messages: Message]
}

// Message.groovy
class Message {
    String content
}
```

- Collection won’t be full loaded on first access

Grails – Associations – One-to-Many

- Fetching a user will fetch messages references
- When iterating over the message Set GORM will fetch the message
- « messages » is a Set
  - Unsorted
  - Can be overridden using a List instead of a Set

```
// Fetch a User from ID (1)
def userInstance = User.get(1)
// Only messages references at this step

// Iterate over the User messages
// will trigger the fetching of the messages
userInstance.messages.each{
    // Print the message content in the console
    println it.content
}
```

Grails – Associations – One-to-Many

- Deleting the « Many » side of the association
  - Declare the « Many » side « belongsTo » the « One » side
  - Declare an explicit cascading behaviour

```
// Instantiate new User
def userInstance = new User(username: "username")

// Adding multiple Messages
["message","other message", "another message", "..."].each {
    userInstance.addToMessages(new Message(content: it))
}

// Persist userInstance will save User and associated Message
userInstance.save()

// Will delete userInstance as well as linked messages
userInstance.delete(flush:true)
```

Grails – Associations – One-to-Many

- Deleting the « Many » side of the association
  - Declare the « Many » side « belongsTo » the « One » side (1)
  - Declare an explicit cascading behaviour (2)

- Example case 1.

```
class User {
    String username
    static hasMany = [messages: Message]
}
```

```
class Message {
    String content
    static belongsTo = [user: User]
}
```

Grails – Associations – One-to-Many

- Deleting the « Many » side of the association
  - Declare the « Many » side « belongsTo » the « One » side (1)
  - Declare an explicit cascading behaviour (2)

- Example case 2.

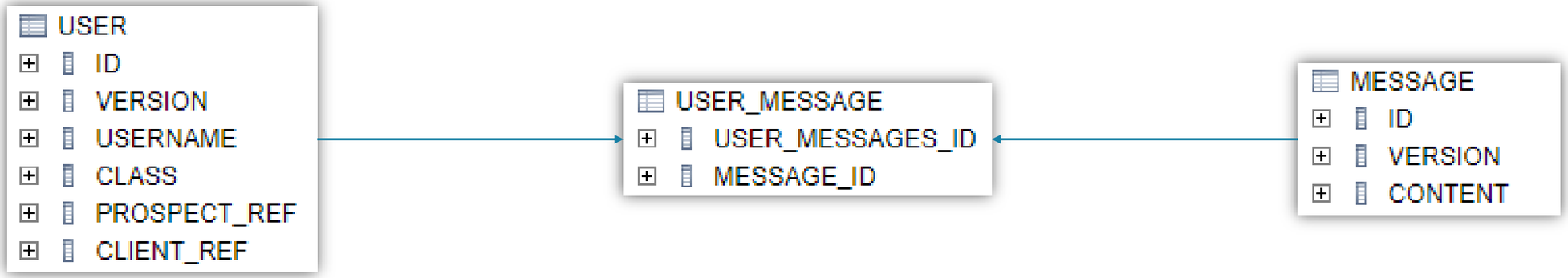
```
class User {
    String username
    static hasMany = [messages: Message]
    static mapping = {
        messages cascade: 'all-delete-orphan'
    }
}
```

```
class Message {
    String content
}
```

Grails – Associations – One-to-Many

- « belongsTo » on the « Many » side of the association
  - Change the database structure

- Without « belongsTo »
  - Association table



Grails – Associations – One-to-Many

- « `belongsTo` » on the « `Many` » side of the association
  - Change the database structure

- With « `belongsTo` »
  - No association table
  - « `Many` » side references the « `One` » side

Grails – Associations – Many-to-Many

- Inconsistent with ORMs
- Defined with « `hasMany` » on both sides
- Must declare the owned side of the relationship using « `belongsTo` »

```groovy
// Book.groovy
class Book {
    String title

    static hasMany = [authors: Author]
}
```

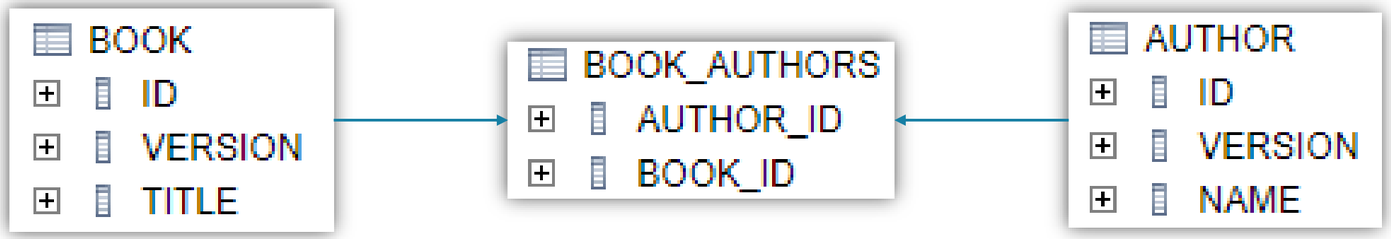
```groovy
// Author.groovy
class Author {
    String name

    static hasMany = [books: Book]
    // Author is the owned side
    static belongsTo = Book
}
```

Grails – Associations – Many-to-Many

- Handled like a « `One-to-Many` » at database level

- Owning side takes responsibility for the persisting
  - Cannot cascade `saves` from the owned side



Grails – Associations – Many-to-Many

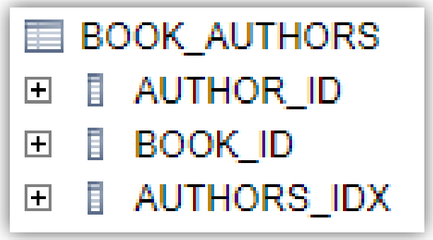
```
// Will save the Book as well as the Authors (Owning side)
new Book(title: "Art and Science of Big Data")
    .addToAuthors(new Author(name: "Serge Miranda"))
    .addToAuthors(new Author(name: "Robin Girard"))
    .addToAuthors(new Author(name: "Benjamin Renaut"))
    .save()

// Will only save the Author (Owned side)
new Author(name: "Serge Miranda")
    .addToBooks(new Book(title:  "Art and Science of Big Data"))
    .addToBooks(new Book(title:  "Relational databases"))
    .save()
```

Grails – Associations – Sets & Lists

- At code level a « `One-to-Many` » is a `Set`
  - Not ordered
  - Cannot hold duplicates
  - Can be costly if collection is really large
  - Can be handled manually (Spring Security)

- Can override `Set` to `List`
  - Ordered
  - Keep unicity constraint from `Set`
  - Automatic index handling
  - Add a new column to keep the index



Grails – Modelling – Constraints

- Declare constraints within domain class
  - Database level constraints
  - Code level constraints
- Constraints property
  - property name
  - named parameters
- Default : Properties are « `nullable: false` »
- Makes development safer

Grails – Modelling – Constraints

Constraint	Description	Example
blank	Validates that a String value is not blank	`login(blank:false)`
creditCard	Validates that a String value is a valid credit card number	`cardNumber(creditCard: true)`
email	Validates that a String value is a valid email address.	`homeEmail(email: true)`
inList	Validates that a value is within a range or collection of constrained values.	`name(inList: ["Joe", "Fred", "Bob"])`
matches	Validates that a String value matches a given regular expression.	`login(matches: "[a-zA-Z]+")`
max	Validates that a value does not exceed the given maximum value.	`age(max: new Date()) price(max: 999F)`
maxSize	Validates that a value's size does not exceed the given maximum value.	`children(maxSize: 25)`
min	Validates that a value does not fall below the given minimum value.	`age(min: new Date()) price(min: 0F)`
minSize	Validates that a value's size does not fall below the given minimum value.	`children(minSize: 25)`
notEqual	Validates that that a property is not equal to the specified value	`login(notEqual: "Bob")`
nullable	Allows a property to be set to `null` - defaults to `false`.	`age(nullable: true)`
range	Uses a Groovy range to ensure that a property's value occurs within a specified range	`age(range: 18..65)`

Grails – Modelling – Constraints

Constraint	Description	Example
scale	Set to the desired scale for floating point numbers (i.e., the number of digits to the right of the decimal point).	`salary(scale: 2)`
size	Uses a Groovy range to restrict the size of a collection or number or the length of a String.	`children(size: 5..15)`
unique	Constrains a property as unique at the database level	`login(unique: true)`
url	Validates that a String value is a valid URL.	`homePage(url: true)`
validator	Adds custom validation to a field.	See documentation

Grails – Modelling – Constraints

```
class User {
    String username
    String email
    static hasOne = [address:Address]
    static hasMany = [messages: Message]

    static constraints = {
        // username can't be empty or null and must be between 5 and 15 char
        username    blank: false, nullable: false, size: 5..15
        // email can't be empty or null, has a valid email format and is unique
        email       blank: false, nullable: false, email: true, unique: true
        // address for a User has to be defined
        address     nullable: false
    }
}
```

Grails – Modelling – Mapping

- Custom ORM mapping
- Customize
  - Tables name
  - Join tables name
  - Fields name
  - Version handling
  - Lazy / Eager fetching
  - Caching strategy
  - Auto timestamping
    - `dateCreated`
    - `lastUpdated`
  - And many more …

```
class User {
    String username
    String email
    static hasOne = [address:Address]
    static hasMany = [messages: Message]
    static mapping = {
        // Custom table name
        table 'frontend_users'
        // Prevent version handling
        version false
        // Custom column name
        username column: 'user_name'
        // Eager fetching for messages
        messages lazy: false
    }
}
```

Grails – Modelling – Mapping

- Custom ORM mapping
- Customize
  - Tables name
  - Join tables name
  - Fields name
  - Version handling
  - Lazy / Eager fetching
  - Caching strategy
  - Auto timestamping
    - `dateCreated`
    - `lastUpdated`
  - And many more …

```
class User {
    […]
    static hasMany = [messages: Message]
    String description
    Date dateCreated
    Date lastUpdated
    static mapping = {
        // Disable autotimestamping
        autoTimestamp false
        // Force field type to 'text'
        description type: 'text'
        // Custom join table
        messages joinTable: [name   : 'user_messages',
                             key    : 'user_id',
                             column: 'message_id']
    }
```

Grails – Modelling – Mapping

- Cascade behaviour
  - `save-update` : cascades only saves and updates
  - `delete` : cascades only delete
  - `all` : cascades saves, updates and deletes
  - `all-delete-orphan` : only with one-to-many, cascades all and delete child when removed from association, also delete children when parent is deleted
  - other cases : http://gorm.grails.org/6.0.x/hibernate/manual/#customCascadeBehaviour

- Defaults
  - `hasMany` cascades save-update
  - `belongsTo` cascades all-delete-orphan

```
class User {
    static hasMany = [messages: Message]

    static mapping = {
        // Define cascade behaviour
        messages cascade: 'all-delete-orphan'
    }
}
```

DATUM ACADEMY

Grails – Controllers

Structure & scopes

eMBDS

Controllers – Basics

- Handle `requests`
- Return `responses`
- Convention / default configuration
  - Each `action` maps to a `URI` (cf. `UrlMappings.groovy`)
  - Default `action` within `controller`

```
// default overridable value
static defaultAction = "list"
```

  - If `index` action is defined, it's the default one

Controllers – Scopes (Objects)

- Define objects used to store variables
- Accessible from controllers
- Access using the scopes name in controllers
- 5 Scopes available
  - servletContext
  - session
  - request
  - params
  - flash

Controllers – Scopes (Objects)

- **`servletContext`**
  - Instance of `ServletContext`
  - Also know as Application scope
  - Available across the entire web application
  - One context per web application per Java Virtual Machine
- `session`
- `request`
- `params`
- `flash`

Controllers – Scopes (Objects)

- `servletContext`
- **`session`**
  - Instance of HttpSession
  - Used to store information associated to a user owning an active session
  - Usually handled with cookies
- `request`
- `params`
- `flash`

Controllers – Scopes (Objects)

- `servletContext`
- `session`
- **`request`**
  - Used to store information relative to the current request
  - Instance of HttpServletRequest
  - Contains all the information about the request
    - Cookies
    - Format
    - Locales
    - Security definitions
    - Request data
    - Host information
    - …
- `params`
- `flash`

Controllers – Scopes (Objects)

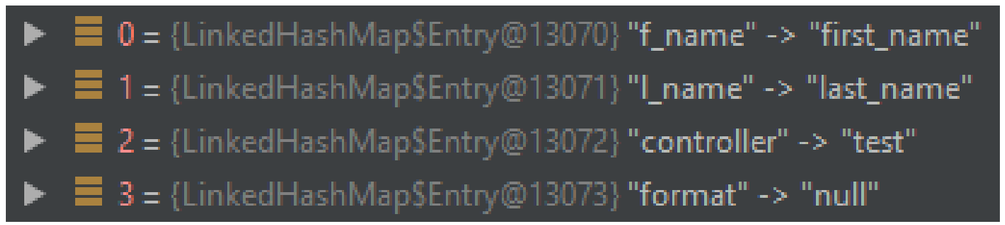
- `servletContext`
- `session`
- `request`
- **`params`**
  - Multi dimensional mutable map
  - Same scope (range) as request
    - Sort of a “map version” of the request scope
  - Contains all request parameters
    - Usually use this scope to get request GET / POST parameters
  - Often used to perform data binding
- `flash`

Controllers – Scopes (Objects)

- `servletContext`
- `session`
- `request`
- **`params`**
  - Called url

```
http://myserver.url/test?f_name=first_name&l_name=last_name
```

  - params scope content



- `flash`

Controllers – Scopes (Objects)

- `servletContext`
- `session`
- `request`
- `params`
- **`flash`**
  - Temporary scope
  - Information available for **current AND next request**
  - Flushed after the next request
  - Useful for specific cases like defining data before a redirect
    - Often used to display confirmation or informative messages about the previous action

Controllers – Scope (range/reach)

- Define the scope (range/reach) of a controller
- Default scope for controller is **`prototype`**
- Can be overridden globally (`application.yml`)
- 3 scopes available
  - prototype
  - session
  - singleton

Controllers – Scope (range/reach)

- prototype
  - A new controller instance is created for each request
- session
  - A new controller instance is created for each user session
- singleton
  - A unique global instance is created and shared
  - Care with this scope : do not set user specific properties as it will be shared with everyone

Controllers – Interceptors

- Based on controllers
- Used to trigger some actions
  - Before controller / action execution
  - After controller / action execution
  - After the view rendering
- Can include / exclude controllers / actions based on names
  - Can use regular expressions
- Can define order / priority notion
- Can be used for basic security implementation

DATUM ACADEMY

Grails – Controllers – Data handling

Data binding & response handling

eMBDS

Data handling

- Data binding
  - Transition between the web / form request data type to Groovy / Java objects
  - Data validation
  - Security

- Responding
  - How to render data to the “user”
  - Formats
  - Using converters
  - Marshallers
  - Handy methods (respond)
  - JSONBuilder

Data handling – Binding

- Request to the server
  - Forms
  - Direct HTTP invocation
- Convert String / Numeric content of the request to the real property type

```
class User {
    String username
    Integer age
}
```

```
// Data is set in a map
def map = [username: "username", age: "50"]
// we then bind data to the model object
def newUserInstance = new User(map)
// and save the newly created User
newUserInstance.save()
```

Data handling – Binding

- Working with associations
  - Single reference

```
class User {
    String username
    Integer age
    static hasOne = [address: Address]
}
```



```
class Address {
    String address
    User   user
}
```

```
// Data is set in a map
def map = [username: "username", age: "50", address: [address: "3 Groovy Place"]]
```

Data handling – Binding

- Working with associations
  - One-to-Many

```
class User {
    String username
    Integer age
    static hasOne = [address: Address]
    static hasMany = [messages: Message]
}
```

```
class Message {
    String content
    static belongsTo = [user: User]
}
```

```
// Data is set in a map
def map = [username: "username", age: "50", address: [address: "3 Groovy Place"],
           "messages[0]": [content: "Message content"],
           "messages[1]": [content: "Other message content"]]
```

Data handling – Binding

- Can be used to update data
  - Slightly different syntax

```
// Load user
def userInstance = User.get(1)
// Data is set in a map
def map = [ username: "username", age: "50", address: [address: "3 Groovy Place"],
            "messages[0]": [content: "Message content"],
            "messages[1]": [content: "Other message content"]]
// we then bind data to the model object this way
userInstance.properties = map
// and save the updated user
userInstance.save(flush: true)
```

Data handling – Binding

- Many other possibilities and cases
- Documentation, pretty rich on this subject
- Can get around binding data manually
  - Retrieve data from the request
  - Control format
  - Update properties accordingly

Data handling – Response

- Model and views
- Respond method
- WithFormat method
- Builders
- Marshallers

Data handling – Response - Model and views

- Model for controllers is a Map handed to the view
- Different ways to hand this model to the view
- Selecting the view
  - Handling with convention
  - Specific rendering

Data handling – Response - Model and views

- Declare and return a Map instance

```
def show()
{
    // Render the user instance in the "user" variable
    [user: User.get(params.id)]
}
```

- Convention for view selection

```
/views/controllerName/actionName.gsp
```

Data handling – Response - Model and views

- Example : Calling - http://myserver.url/user/show?id=1

```
// UserController.groovy
def show()
{
    // Render the user instance in the "user" variable
    [user: User.get(params.id)]
}
```

- Will render
  - Data : User instance in a “user” variable
  - View : Will target the view defined under

```
/views/user/show.gsp
```

Data handling – Response - Model and views

- Other methods
  - return `ModelAndView`
  - `render` method

```
// Render the user instance in the "user" variable
[user: User.get(params.id)]

// Strictly equivalent
return new ModelAndView("/user/show", [user: User.get(params.id)])

// Also equivalent
render(view: "/user/show", model: [user: User.get(params.id)])
```

Data handling – Response - Respond

- Preferred way to return data
- Autonomously handle content negotiation analysing
  - HTTP Header “Accept”
  - Request parameter
  - URI extension
- Find the best suited matching mime type
- Tries to render the response with the corresponding type

Content negotiation

- Default mime types
- Defined in `application.yml`

```yaml
types:
    all: '*/*'
    atom: application/atom+xml
    css: text/css
    csv: text/csv
    form: application/x-www-form-urlencoded
    html:
      - text/html
      - application/xhtml+xml
    js: text/javascript
    json:
      - application/json
      - text/json
    multipartForm: multipart/form-data
    pdf: application/pdf
    rss: application/rss+xml
    text: text/plain
    hal:
      - application/hal+json
      - application/hal+xml
    xml:
      - text/xml
      - application/xml
```

Data handling – Response - Respond

- Example

```
class User {
    String username
    Integer age
    static hasOne = [address: Address]
    static hasMany = [messages: Message]
}
```

```
// UserController.groovy
def list()
{
    // Will try to handle content negotation
    // Then return the data the most appropriate way
    respond User.list()
}
```

Data handling – Response - Respond

- Calling - http://myserver.url/user/list

- From web browser
  - Will try to return corresponding web page (HTTP Error 500 if page does not exist)

- Forging HTTP request (with cURL for example) without any hint about type gives the same result

```
curl http://myserver.url/user/list -I
```

  - Will return `HTTP/1.1 500` if pages does not exist
  - Will return de page html content if it exist
    - Reminder about conventions

Data handling – Response - Respond

- Same scenario with “Accept” header set

  `curl http://myserver.url/user/list -H "Accept: application/json“`

  - Will return `HTTP/1.1 200`, meaning “OK” with data

```
[
      {
            "id":1,
            "address":{"id":1},
            "age":25,
            "username":"username-5",
            "messages":[{"id":1}]
      }
]
```

Data handling – Response - Respond

- Can alternatively set the format using URI extension
  `curl http://myserver.url/user/list.JSON`
  - Will return `HTTP/1.1 200`, meaning “OK” with data

```
[
      {
            "id":1,
            "address":{"id":1},
            "age":25,
            "username":"username-5",
            "messages":[{"id":1}]
      }
]
```

Data handling – Response - WithFormat

- Objective is the same
- Do not handle any autonomous content negotiation analysis
- Way to provide specific rendering for each content type
- Useful when each content type should not render the same data
- Wildcard to handle « all the other cases »

```
// Store the user list
def userList = User.list()

// Specific rendering for predefined content types
withFormat {
    html {render (template: 'user', model:[userList: userList])}
    json {render userList as JSON}
    '*' {render userList as XML}
}
```

Data handling – Response – Builder

- Used to build specific response format
- Handy for isolated needs
- No opportunity to customize global rendering for an object
  - Use marshallers instead
- Extremely verbose and messy

Data handling – Response – Builder

```
def userList = User.list()
def builder = new JsonBuilder()
// Build JSON
def result = builder.users {
    // Iterate over each user of the list
    userList.each {
        // Create new node for each
        User userInstance -> user {
            // Declare all properties
            id userInstance.id
            address (id: userInstance.address.id)
            age userInstance.age
            username userInstance.username
            // Can go deeper in associations
            messages {
                userInstance.messages.each {
                    Message messageInstance -> message(id: messageInstance.id)
                }
            }
        }
    }
}
// Render content as String
render builder.toPrettyString()
```

Data handling – Response – Marshallers

- Extremely useful if you need to customize the rendering of a given Object
- Same syntax as Builder
  - Also extremely verbose and messy
- Allow global declarations
- Can be declared
  - In the `Bootstrap.groovy` but not ideal
  - In the Groovy sources directory, cleaner (`src/main/grooy`)
- Alternatively some plugins helps to keep project cleaner
- Simply a closure returning map on domain class
- Have to build custom marshaller for
  - Domain class
  - Domain class collection

Data handling – Response – Marshallers

```
// Register marshaller on domain class
JSON.registerObjectMarshaller(User)
{
    def result = [:]
    result.id = it.id
    result.age = it.age
    result.username = it.username
    // Should also declare custom marshaller for associations
    // Address & Message
    result.address = it.address
    result.messages = it.messages
    return  result
}
```

DATUM ACADEMY

Grails – Services

Basics

eMBDS

Services – Basics

- Should contain all the reusable business logic
  - Avoid business logic in controllers
- Located under « `grails-app/services` » directory
- Convention : Class name should end with « Service »
- Often in charge of persistence operation
  - Should be Transactional in most cases
  - Default Transactional before Grails 3.1
  - On demand since
    - Transactional Annotation
    - withTransaction method

Reminder – Transactions – A.C.I.D.

- Transactional environment must respect rules : A.C.I.D. properties

- A for Atomicity
  - Most basic principle of transactions
  - Either everything or nothing is saved
  - Commonly achieved by using BEGIN, COMMIT and ROLLBACK keywords
    - Once a transaction is started (BEGIN)
    - Everything will be executed and saved (COMMIT)
    - Or everything will be reverted to its original state (ROLLBACK)
  - In Grails environment, Exceptions throw within a transaction can trigger a rollback
    - Rollback can occur event when not explicitly called

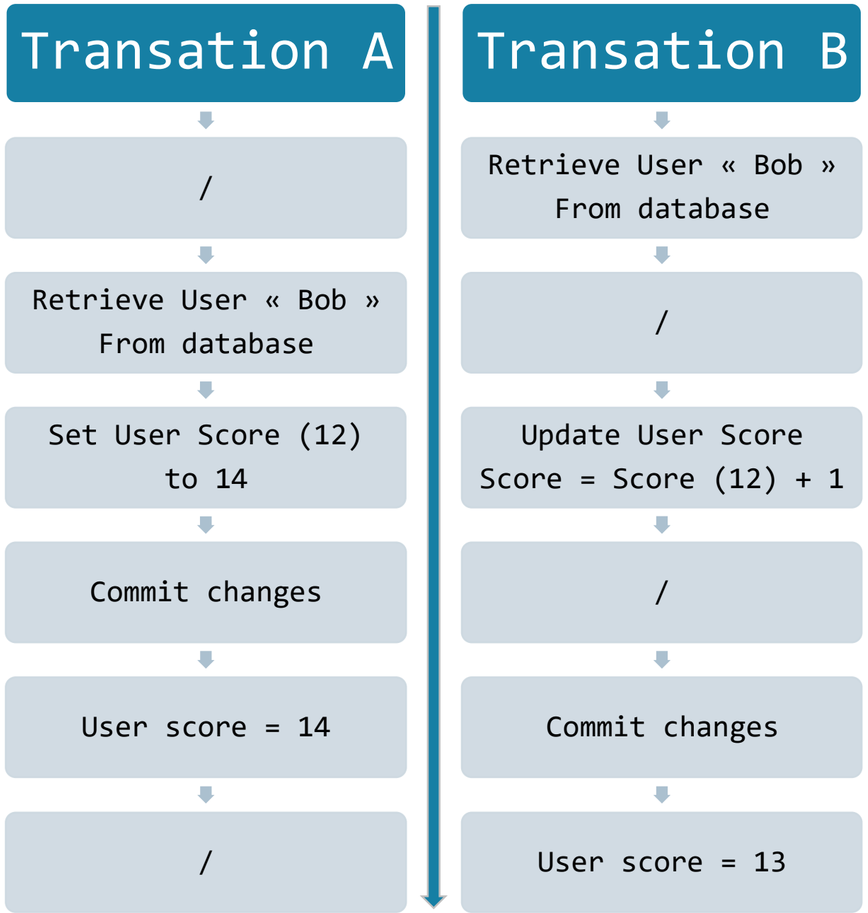
Reminder – Transactions – A.C.I.D.

- C for Consistency
  - Database state before and after transaction must respect unicity, foreign key or other constraints
    - Consistency must be preserved
    - Some exceptions for the « in the middle » state

- I for Isolation
  - Most difficult issue
  - Define how to handle concurrent read / update / delete on a database
    - The data I am going to update may have changed between the time read it and the moment I will update
    - Write the data without checking may revert previous changes
    - Revert the changes I was going to make cancels an action that should have been executed
    - Going in the middle may lead to a data corruption in the database

Reminder – Transactions – A.C.I.D.

- I for Isolation
  - Many mechanism to handle this
    - Customizable isolation level
    - Depends on the DBMS
  - Drawback : Deadlock
  - Forces DBMS to Rollback “some” transactions



Reminder – Transactions – A.C.I.D.

- D for Durability
  - If a commit is validated by the DBMS data integrity must be maintained
  - Even if
    - An error occurs
    - Server is shut down
  - Server must make sure data are not lost

- Conclusion
  - Transactions are essentials for data integrity
  - Transactions will lead to issues that should be resolved with good development practices

Services – @Transactional

- Can define a service as « Transactional » with this Annotation
- Can fine tune Transactional behaviour
  - readOnly option
- Can override at method level
- Rollback triggered on RuntimeException throw

```
@Transactional
class UserService {
    def create() { }

    @ReadOnly
    def list() { }

    @Transactional(readOnly = true)
    def get() { }

    @NotTransactional
    def doThis() { }
}
```

Services – withTransaction

- Programmatic transaction
- No annotation needed

```
class UserService {
    def create(String name, Integer age) {
        User.withTransaction {
            status ->
                new User(name: name, age: age).save()
        }
    }
}
```

Services – withTransaction

- Programmatic rollback using TransactionStatus
- setRollbackOnly method will set the transaction state as “rollback-only”

```
def edit(Long id, String name, Integer age) {
    User.withTransaction {
        status ->
            def userInstance = User.get(id)
            userInstance.name = name
                userInstance.age = age
            // If conditions are not met, rollback
            if (age < 18)
                status.setRollbackOnly()
            else userInstance.save()
    }
}
```

Services – Scopes

- Like Controllers Services has scopes
- Default scope
  - Singleton – One overall instance of the service
- Available scopes
  - Prototype – new Service instance for each **injection**
  - Request – new Service per request
  - Session – new Service for each user session
  - Flash – new Service for current and next request
  - Flow – new Service for the scope of the flow (web flow)
  - Conversation – new Service for the scope of the conversation (web flow)

Services – Injection

- Usage after injection
- Injection by convention
- Same goes for
  - Services
  - Bootstrap
  - Taglibs

```groovy
// UserService.groovy
class UserService {
    def create(…) {
        […]
    }
}
```

```groovy
// UserController.groovy
class UserController {

    // Injection by name (first is lower case)
    def userService

    // Can define type, equivalent
    UserService userService

    def index() {
        […]
        userService.create("Bob", 25)
        […]
    }
}
```

DATUM ACADEMY

Grails – GORM & Hibernate

Querying

eMBDS

Grails – GORM – Querying – Basic CRUD

DATUM ACADEMY

- Create
  - Create an object instance
  - Call `save()` method to ask Hibernate to persist

```
class Example {
    String name
    Integer rating
    Boolean isValid
}
```

```
// Create an object instance
def exampleInstance = new Example(name: "name", rating: 10, isValid: true)

// Ask GORM > Hibernate to persist this instance
exampleInstance.save()
```

Grails – GORM – Querying – Basic CRUD

DATUM ACADEMY

- **save** & **delete**
  - `flush` option : ask an immediate flush of the persistence context, persist or delete immediately
  - returns null on validation fail

- **save** specific options
  - `validate` : define if validation should be skipped
  - `insert` : define if Hibernate should use SQL INSERT, useful to clarify INSERT / UPDATE when assigning ID
  - `failOnError` : if set to “true”, will throw an exception and stop the application server if validation fails. Must use when setting essential data.
  - `deepValidate` : default “true”, if set to false, do not validate associations

Grails – GORM – Querying – Basic CRUD

- Example

```
// Instantiate new User
def userInstance =
        new User(username: "username",
                 email: "user@email.com",
                 description: "description",
                 address: new Address())
                 .addToMessages(new Message(content: "message content"))

// If immediate save is not successful (validation fail)
if (!userInstance.save(flush: true))
    println "Validation failed, User has not been saved to database"
```

Grails – GORM – Querying – Basic CRUD

- Read
  - Choose wisely, strong impact long term

```
// Retrieve the object from database
def exampleInstance = Example.get(1)

// Retrieve a read-only object from the database
def exampleInstanceReadOnly = Example.read(1)

// Retrieve a proxy for the designated object instance
def exampleInstanceLoad = Example.load(1)
```

Grails – GORM – Querying – Basic CRUD

- Update
  - Load object instance
  - Edit properties
  - Call persist method

```
// Update
def exampleInstance = Example.get(1)
exampleInstance.name = "new name"
exampleInstance.save()

// Delete
exampleInstance.delete()
```

- Delete
  - Same as Create with `delete()` method

Grails – GORM – Querying - Dynamic Finders

DATUM ACADEMY

- Dynamic Finders
  - “Auto-Magically” methods generated using code synthesis at runtime
  - Based on the properties of the class
  - Use like a static method
  - Ideal for simple queries
    - Quickly difficult to read / understand with long queries
    - No optimisation
- Where Queries
- Criteria Queries
- Hibernate Query Language (HQL)

Grails – GORM – Querying - Dynamic Finders

- Data fetching

```
// Get from ID, uses cache, one request
def userInstance = User.get(1)

// Strictly identical using dynamic finders
// Slower than previous method
def sameUserInstance = User.findAllById(1)

// Get all
def allUsers = User.list()
def alsoAllUsers = User.getAll()

// Get & get all on specific property
def user = User.findByUsername("username")
def userList = User.findAllByEmailLike("%@email.com")
```

Grails – GORM – Querying - Dynamic Finders

DATUM ACADEMY

- Data fetching

```
// Use this when you need all raw data
def userList = User.getAll()

// Same as getAll, can use HQL, support pagination
def sameUserList = User.findAll("from User as u where u.username=?",
                                ['username'],
                                [max: 10, offset: 5])

// Same as getAll, support pagination and few more options
def anotherUserList = User.list(max: 10, offset: 5)
```

Grails – GORM – Querying - Dynamic Finders

DATUM ACADEMY

- Dynamic Finders

```
class Book {
    String title
    Author author
}
```

```
class Author {
    String name
}
```

```
def bookInstance = Book.findByTitle("title")
assert bookInstance instanceof Book

def bookList = Book.findAllByTitle("title")
assert bookList instanceof List // even if empty or only one entry

def authorInstance = Author.findByName("authorsName")
def otherBookInstance = Book.findByTitleLikeAndAuthor("%title%", authorInstance)
```

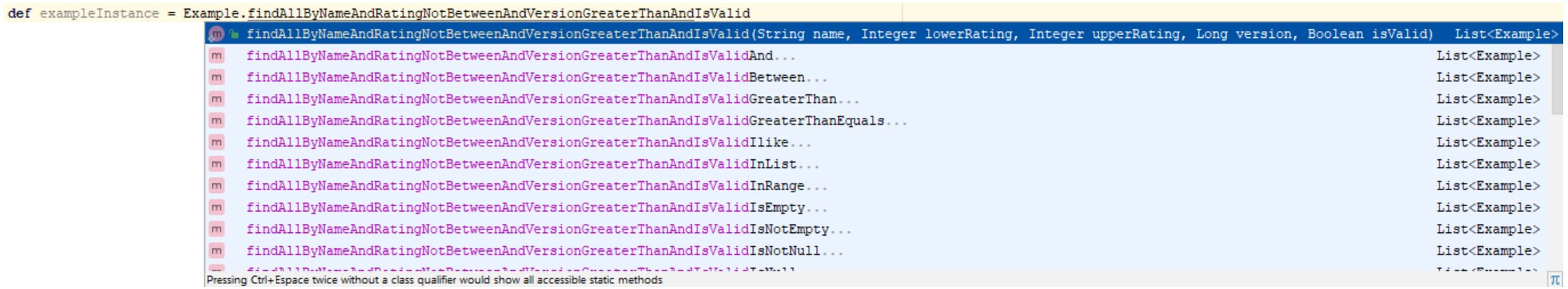
Grails – GORM – Querying - Dynamic Finders

DATUM ACADEMY

- Operators
  - InList
  - Like / Ilike
  - LessThan / LessThanEquals / GreaterThan / GreaterThanEquals
  - IsNull / IsNotNull
  - Between
  - Rlike
  - NotEqual
  - InRange

Grails – GORM – Querying - Dynamic Finders

- Useful for simple cases
- Quickly messy



Grails – GORM – Querying – Where Queries

DATUM ACADEMY

- Dynamic Finders
- Where Queries
  - More flexible than **Dynamic finders**
  - Less verbose than **Criteria**
  - Use regular Groovy comparison operators
  - Two steps
    - Build Query
    - Execute request
- Criteria Queries
- Hibernate Query Language (HQL)

Grails – GORM – Querying – Where Queries

Operators	Description
==	Strict equality
!=	Difference
>	Greater than
<	Less than
>=	Greater than or equal
<=	Less than or equal
in	Contained within list
==~	Case sensitive like
=~	Case insensitive like

Grails – GORM – Querying – Where Queries

DATUM ACADEMY

- Examples

```
class Book {
    String title
    Author author
}
```

```
class Author {
    String name
}
```

```
// Build the query
def bookQuery = Book.where { title == "title" }

// Execute the built query
Book bookInstance = bookQuery.find()
List bookList = bookQuery.findAll()

// Alternative for immediate execution
List otherBookList = Book.findAll { title == "title" }
List sortedBookList = Book.findAll (sort: "title") { title == "title" }
```

Grails – GORM – Querying – Where Queries

Method	Description
`second / minute / hour`	The second / minute / hour of a date property
`day / month / year`	The day / month / year of a date property
`lower / upper`	Converts a string
`length`	The length of a string
`trim`	Trims a string

```
class Comment {
    Date dateCreated
}
```

```
// Select comments where creation year is 2019
def commentQuery = Comment.where {
    year(dateCreated) == 2019
}
```

Grails – GORM – Querying – Where Queries

- Complex queries

```
def bookQuery = Book.where {
    // Complex query on title
    (title =~ "%title%" && title != "wrong title")
    // Query on association
    author { name =~ "%authorsName%" && name != "authorsName1" }
}
Book bookInstance = bookQuery.find()
```

Grails – GORM – Querying – Where Queries

Subquery functions	Description
`avg`	The average of all values
`sum`	The sum of all values
`max`	The maximum value
`min`	The minimum value
`count`	The count of all values
`property`	Retrieves a property of the resulting entities

Grails – GORM – Querying – Where Queries

DATUM ACADEMY

- Subqueries
  - Impossible with Dynamic finders

```
class Comment {
    Integer rating
    String  comment
    String  author

    static belongsTo = Book
}
```

```
// Select comments with rating above average
def commentQuery = Comment.where {
    rating > avg(rating)
}

// Adding additional criteria with closure
// Select comments
// > with rating above average rating where comment author is like Bob%
// > where comment author is not Bob
def otherCommentQuery = Comment.where {
    rating > avg(rating).of { author ==~ "Bob%" } && author != "Bob"
}
```

Grails – GORM – Querying – Criteria

DATUM ACADEMY

- Dynamic Finders
- Where Queries
- Criteria Queries
  - More flexible than **Where queries**
  - A lot more verbose than **Where queries**
  - Build with `createCriteria()` or `withCriteria()` method
  - Top level conditions bound with logical “AND”
  - Can group criteria with `and` / `or` / `not` blocks
  - Last resort before using HQL
- Hibernate Query Language (HQL)

Grails – GORM – Querying – Criteria

DATUM ACADEMY

Method	Description
`eq`	Strict equality
`ne`	Difference
`gt`	Greater than
`lt`	Less than
`ge`	Greater than or equal
`le`	Less than or equal
`inList`	Contained within list
`like`	Case sensitive like
`ilike`	Case insensitive like

Grails – GORM – Querying – Criteria

- Example

```
// Create criteria on Comment domain
def criteria = Comment.withCriteria {
    // Check "author" property value is equal to "Bob"
    eq("author","Bob")
    // Check the "comment" property is like "%comment%" with case insensitive
    ilike("comment","%comment%")
    // Or rating value is greater than 3
    or {
        gt("rating", 3)
    }
}
```

Grails – GORM – Querying – HQL

- Dynamic Finders
- Where Queries
- Criteria Queries
- Hibernate Query Language (HQL)
  - Most flexible option
  - Addressing queries directly to the ORM
  - Lesser abstraction, better performances overall
  - Building queries with `find()`, `findAll()` or `executeQuery()` methods
  - Better than SQL to remain compliant with any DBMS

Grails – GORM – Querying – HQL

DATUM ACADEMY

- Example

```
// Classic HQL select
def exampleInstance = Example.find("from Example as e where e.name = 'name'")

// With named paramter, always use this method
def otherExampleInstance =
        Example.find("from Example as e where e.name = :name",
                [name: "name"])
```

eMBDS

Grails – GORM - Conclusion

DATUM ACADEMY

	dynamic finder	where clause	criteria	HQL	SQL
simple queries	x	x	x	x	x
complex filters		x	x	x	x
associations		x	x	x	x
property comparisons		x	x	x	x
some subqueries		x	x	x	x
eager fetches w/ complex filters			x	x	x
projections			x	x	x
queries with arbitrary return sets				x	x
highly complex queries (like self joins)					x
some database specific features					x
performance-optimized queries					x

eMBDS

DATUM ACADEMY

Module 3: REST API

Concepts – Development – Testing – Security

eMBDS

DATUM ACADEMY

REST API

Concepts

eMBDS

Plan

- API REST
- Principe d’implémentation
- Verbes HTTP
- Génération de messages
- HTTP Status Code
- Formats d’échange
- HATEOAS : HAL
- Exemple détaillé
- Mécanismes de protection

REST

DATUM ACADEMY

- Born around 2000
- Roy Fielding PhD
- REpresentational State Transfer
- Relies entirely on HTTP
- Accessing resources
- HTTP verbs usage
- REST ⇔ RESTful

REST & SOAP

- REST
  - Architectural principles
- SOAP
  - Specification of a standard communication protocol

REST & SOAP

- Architectures to provide web

- Major differences
  - Implementation
  - Readability
  - WSDL
    - Required message format
    - Accessible methods
    - Service location

REST & SOAP – Pros & Cons

- REST
  - Achitecture / Pattern
  - Format agnostic
  - Relies entirely on HTTP
  - Lightweight
  - Easily readable
  - Cache handling (HTTP GET)
  - Should provide documentation
  - Stateless
  - No overload
  - Less verbose
  - No added functionalities (Security ?)
- SOAP
  - Protocol
  - XML only
  - Not limited to HTTP
  - Heavy duty
  - Less readable
  - HTTP POST, no Cache
  - WSDL
  - Either Statefull or Stateless
  - SOAP Enveloppe + Optional Headers
  - Well structured
  - Optional layers to add functionalities

REST & SOAP

DATUM ACADEMY

- No real winner
  - Both viable solution for different cases
- SOAP
  - Security is a must
    - Handling health data
    - Banking area
  - Need other layers to add functionalities
- REST
  - Prototypes
  - Need for a high Client – Server segmentation
  - Need something else than XML

eMBDS

REST – Properties & Constraints

- Client – Server Segmentation
  - Improve portability of user interfaces
  - Improve scalability
    - Focus on Server
  - Independent components
    - Simplifies evolutions

- Cacheability
  - Clients and Servers can cache responses data
  - Responses must be declared as cacheable (if it’s true)
    - Prevent User from getting outdated or wrong data
  - Improve extensibility

REST – Properties & Constraints

- Stateless
  - State management is achieved by the Client
  - Server do not store any context about Client
  - Improve Client – Server segmentation
  - Client requests must contain all necessary information to allow Server to respond accordingly
  - Exception for session information storage
    - So User doesn’t have to send identification data each time
  - No permanent link between Client and Server
    - No Server monopolization by a Client
    - No Server saturation if handling multiple Clients at once
    - Better extensibility

REST – Properties & Constraints

- Layered system
  - System may be layered with hierarchy notion
    - Constrain component behaviour
    - Each component cannot access component beyond n+1 or n-1
      - Each component is less complex
      - Reduce overall complexity
  - Ability to handle growing needs
    - Load balancers
    - Intermediary servers
    - Must be transparent
- Drawback, more layers, more latency
  - Can be limited with shared caches

REST – Properties & Constraints

- Code on demand (Optional)
  - Server can temporarily extend client functionalities providing executable code
    - Java applets
    - Javascript
  - Increase flexibility
  - Reduce visibility
  - Security

REST – Properties & Constraints

- Uniform interface – 4 constraints
  - Resources identification
    - Use URIs to idenfify a resource
    - Representation is not the resource itself
    - Allow components to evolve independently
  - Resource manipulation through representations
    - Once a resource is known, can manipulate with HTTP methods (GET, POST, PUT & DELETE)
  - Self descriptive messages
    - Each message include enough information to understand how to handle it (formats, cache timestamps)
  - Hypermedia as the engine of application state (HATEOAS)
    - Responses should include links, allowing client to « browse » available actions on the current resource

- Pros & cons
  - Data is normalized, structire is predictable
  - Component evolution is easier
  - Client side implementation can be more challenging
  - Reduce overall performances

DATUM ACADEMY

REST API

Implementation principle

eMBDS

REST – Implementation principle

- Define resources and resources collections
- Create resources
  - Attributes
  - Constraints
- Define exchange format
  - Unique format
  - Multi format

REST – Implementation principle

- Message semantic linked to HTTP methods

- HTTP Verbs / Methods RFC2616
  - GET : Retrieve a resource / resource collection representation
  - POST : Create a new resource
  - PUT / PATCH : Update an existing resource
  - DELETE : Delete a resource
  - HEAD : Retrieve metinformation about a resource (~GET)
    - Should not have any body
    - Only metadata
  - OPTIONS : Determine requirements and available actions without initiating a resource retrieval
    - Underestimated part of HTTP protocol
    - Can be used to improve services interconnection

REST – Format handling

- Client asks format with « Accept » Header

```
Accept: <MIME_type>/<MIME_subtype>
Accept: <MIME_type>/*
Accept: */*
```

- Server confirms with « Content-type » Header

```
Content-Type : <MIME_type>/<MIME_subtype>
Content-Type : <MIME_type>/*
Content-Type : */*
```

DATUM ACADEMY

REST API

HTTP Methods

eMBDS

REST – HTTP Methods – GET

- Used to retrieve a resource representation
  - No body
  - Parameters after URI
  - Cacheable
  - Bookmarkable
- Never create / update / delete a resource
- Idempotent
- Return
  - HTTP 200 (OK)
  - HTTP 404 (NOT FOUND)
  - HTTP 400 (BAD REQUEST)

REST – HTTP Methods – POST

- Used to create a resource
  - Not cacheable
  - Not bookmarkable
- Alternative to GET when too much parameters
- Return
  - HTTP 201 (CREATED)
  - HTTP 204 (NO CONTENT)
  - HTTP 404 (NOT FOUND)
  - HTTP 400 (BAD REQUEST)

REST – HTTP Methods – PUT

- Used to update a resource
  - Not cacheable
  - Not bookmarkable
- Some APIs create resource if targetted resource is not found
- Idempotent
- Return
  - HTTP 200 (OK)
  - HTTP 201 (CREATED)
  - HTTP 204 (NO CONTENT)
  - HTTP 404 (NOT FOUND)
  - HTTP 400 (BAD REQUEST)

REST – HTTP Methods – PATCH

- Used to update a resource
  - Not cacheable
  - Not bookmarkable
- Should be preferred to PUT for partial updates
- Return
  - HTTP 200 (OK)
  - HTTP 204 (NO CONTENT)
  - HTTP 404 (NOT FOUND)
  - HTTP 400 (BAD REQUEST)

REST – HTTP Methods – DELETE

- Used to delete a resource
  - Not cacheable
  - Not bookmarkable
- Return
  - HTTP 200 (OK)
  - HTTP 204 (NO CONTENT)
  - HTTP 404 (NOT FOUND)
  - HTTP 400 (BAD REQUEST)

REST – HTTP Methods – OPTIONS

- Used to retrieve available communication options
  - Not cacheable
  - Not bookmarkable
- No action on resources
- Return
  - HTTP 200 (OK)
  - HTTP 404 (NOT FOUND)

```
HTTP/1.1 200 OK
Allow: GET,HEAD,POST,OPTIONS
Content-Type: text/html; charset=UTF-8
Date: Wed, 25 Sep 2019 13:17:54 GMT
Content-Length: 0
```

REST – HTTP Methods – HEAD

- Used to retrieve metainformation
  - Cacheable
  - Not bookmarkable
- Should not return any body
- Same header as GET on same resource
- Return
  - HTTP 200 (OK)
  - HTTP 404 (NOT FOUND)

```
HTTP/1.1 200 OK
Content-Type: text/html; charset=UTF-8
Date: Wed, 25 Sep 2019 13:17:54 GMT
Content-Length: 15426
Last modified: Tue, 24 Sep 2019 00:00:00 GMT
```

DATUM ACADEMY

REST API

Exchange

eMBDS

REST – Exchange (1)

- Client side – Request building
  - Define resource
    - Singleton resource
    - Resource collection
  - Define the action, use appropriate HTTP method
    - GET
    - POST
    - PUT
    - PATCH
    - DELETE
    - HEAD
    - OPTIONS

REST – Exchange (2)

- Server side – Request handling & resolution
  - Define requested resource (URI)
  - Define HTTP method used
  - (Optional) Access control & permissions
    - Read and check token
  - Resolve action
  - Generate response
    - Define format
    - Return resource or resource collection representation with appropriate format
    - Return appropriate HTTP response code

REST – Exchange (3)

- Client side – Response reading
  - Check HTTP response code
    - HTTP 2xx response code : success
    - HTTP 3xx response code : redirection
    - HTTP 4xx / 5xx response code : error
  - HTTP 2xx, if request method was
    - GET : Parse response to retrieve resource representation
    - POST : Resource was successfully created
    - PUT / PATCH : Resource was successfully updated
    - DELETE : Resource was sucessfully deleted
  - HTTP 4xx / 5xx
    - An error occured
    - May find reasons in the response
    - HTTP response code may help

REST – Exchange – Conclusion

- Very similar to regular HTTP exchanges

- Request building is basic
  - Depends on environment
  - Native mobile development using basics libraries
  - Web development using Ajax

- Response handling
  - XML / JSON / … parser

REST – Exchange – Conclusion

- Testing
  - Browser development and debugging tools
    - Chrome DevTools
    - Firefox Firebug
  - cURL
  - Postman
- Knowledge of HTTP is a must : RFC2616
- HTTP response codes : List
- Basic knowlegde about redirections

DATUM ACADEMY

REST API

HTTP Response Codes

eMBDS

REST – HTTP Response Codes

- 200 OK
  - Request was successfull
  - Content depends on used HTTP method
- 201 CREATED
  - Request was successfull
  - Resource was created
- 204 NO CONTENT
  - Request was successfull
  - No data was returned

REST – HTTP Response Codes

- 400 BAD REQUEST
  - Server wont process request due to an error
    - Malformed request
    - Error on provided parameters
    - ...
- 401 UNAUTHORIZED
  - Server will not allow access to define resource without proper authentication
- 403 FORBIDDEN
  - Request was valid but the provided authentication do not allow access to this resource
- 404 NOT FOUND
  - Requested resource is not be found

REST – HTTP Response Codes

- 405 METHOD NOT ALLOWED
  - HTTP method used is not supported
- 500 INTERNAL SERVER ERROR
  - Generic error message
  - Often returned by application or web server
  - Should not be intentionally returned, too vague
- 301 MOVED PERMANENTLY
  - Provide a new URI for requested resource
  - Use this to redirect request with trailing slash
- 302 FOUND – MOVED TEMPORARILY
  - Provide a new URI for requested resource

DATUM ACADEMY

REST API – HATEOAS & HAL

Hypermedia As The Engine Of Application State

eMBDS

REST – HATEOAS – Concept

- Hypermedia As The Engine Of Application State
  - REST Constraint
  - Client does not have to know the deployed application
  - Increase Client – Server segmentation

REST – HATEOAS – Concept

- Classic response from a REST API

```
HTTP/1.1 200 OK
Content-Type: application/xml
Content-Length: ...

<?xml version="1.0"?>
<account>
    <account_number>12345</account_number>
    <balance currency="usd">100.00</balance>
</account>
```

REST – HATEOAS – Concept

- Response from a REST API implementing the HATEOAS Constraint

```
HTTP/1.1 200 OK
Content-Type: application/xml
Content-Length: ...

<?xml version="1.0"?>
<account>
   <account_number>12345</account_number>
   <balance currency="usd">100.00</balance>
   <link rel="deposit" href="https://somebank.org/account/12345/deposit" />
   <link rel="withdraw" href="https://somebank.org/account/12345/withdraw" />
   <link rel="transfer" href="https://somebank.org/account/12345/transfer" />
   <link rel="close" href="https://somebank.org/account/12345/close" />
 </account>
```

REST – HATEOAS – HAL

- HAL : Hypertext Application Language
- One of many HATEOAS implementation
- Principles
  - Simple format to link resources
  - Make the API browsable
  - More difficult to set up at first
    - Many libraries to produce and consume HAL
  - Define conventions to describe resources as JSON or XML

REST – HATEOAS – HAL

- Objectives
  - Spend less time on format design
  - Focus on implementation and documentation
  - Build « explorable » APIs
  - Associations enable resources identification and possible interractions
  - Base entry point allow developer to discover links

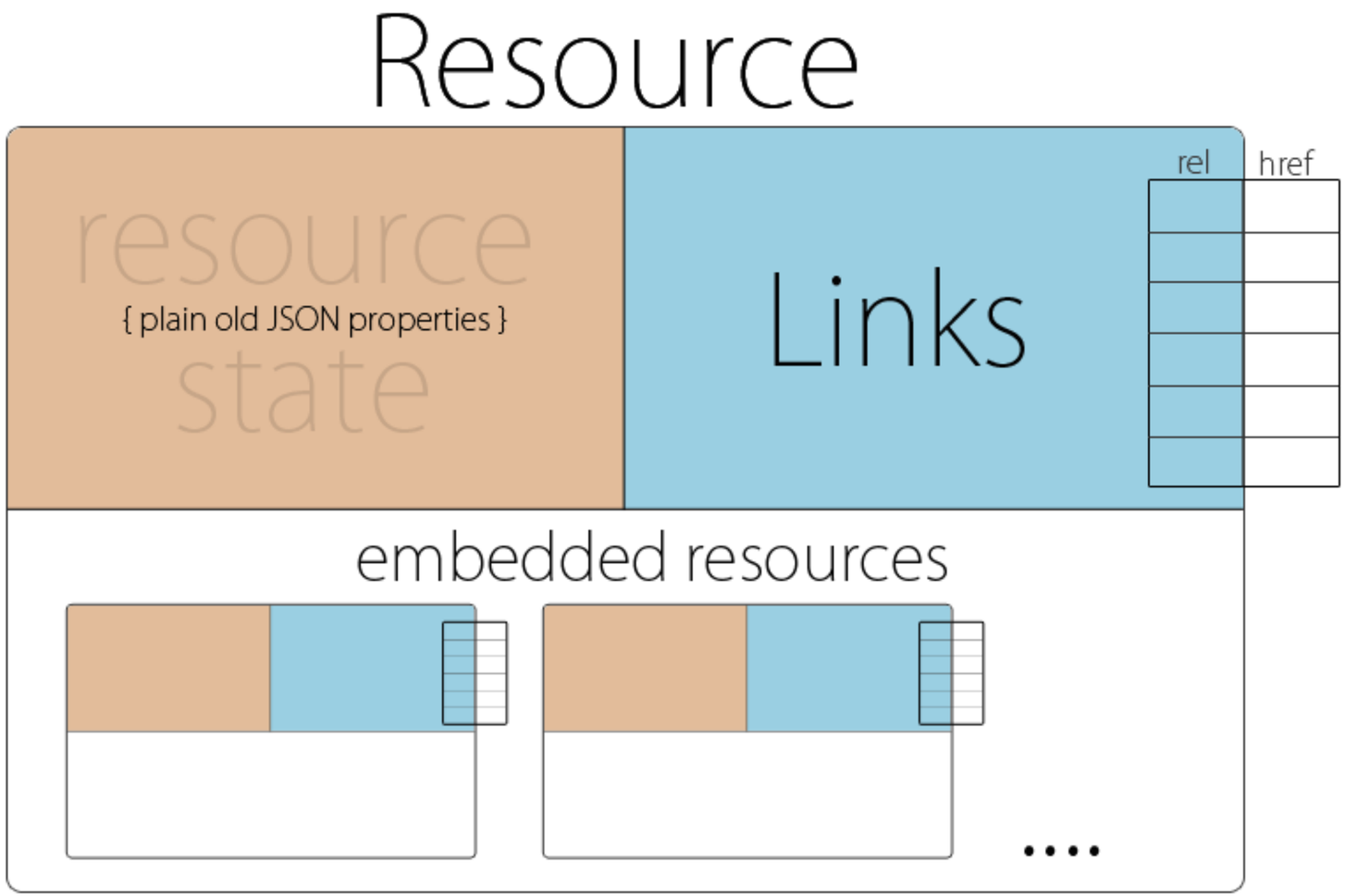
- Can be considered as an API descriptor

- API exploration makes documentation « almost useless »

REST – HATEOAS – HAL

- Based on two types of representations
  - Resources
    - Contain links
    - Define other subresources
    - Define a state
  - Link
    - Define a target (URI)
    - Linked to a relation (rel)
    - Contains optional properties to manage version and format handling

REST – HATEOAS – HAL



REST – HATEOAS – HAL – Example

- Resource links (1)
- Resource properties (2)
- Subresources and their links (3)

```
{
  "_links": {
    "self": { "href": "/orders" },
    "curies": [{ "name": "ea", "href": "http://example.com/docs/rels/{rel}", "templated": true }],
    "next": { "href": "/orders?page=2" },
    "ea:find": {
        "href": "/orders{?id}",
        "templated": true
    },
    "ea:admin": [{
        "href": "/admins/2",
        "title": "Fred"
    }, {
        "href": "/admins/5",
        "title": "Kate"
    }]
  },
  "currentlyProcessing": 14,
  "shippedToday": 20,
  "_embedded": {
    "ea:order": [{
        "_links": {
            "self": { "href": "/orders/123" },
            "ea:basket": { "href": "/baskets/98712" },
            "ea:customer": { "href": "/customers/7809" }
        },
        "total": 30.00,
        "currency": "USD",
        "status": "shipped"
    }, {
        "_links": {
            "self": { "href": "/orders/124" },
            "ea:basket": { "href": "/baskets/97213" },
            "ea:customer": { "href": "/customers/12369" }
        },
        "total": 20.00,
        "currency": "USD",
        "status": "processing"
    }]
  }
}
```

1

2

3

REST – HATEOAS – CURIEs

- Compact URIs
- Shortcut for links
- May have many different
- Use of a placeholder ({rel})
  - Calling find method on http://example.com/docs/rels/orders{?id}

```
"_links": {
    "self": { "href": "/orders" },
    "curies": [{ "name": "ea", "href": "http://example.com/docs/rels/{rel}", "templated": true }],
    "next": { "href": "/orders?page=2" },
    "ea:find": {
        "href": "/orders{?id}",
        "templated": true
    },
```

REST – HATEOAS – HAL – Content Type

- Classic REST

```
json:
  - application/json
  - text/json
```

```
xml:
  - text/xml
  - application/xml
```

- Specific to implementation

```
hal:
  - application/hal+json
  - application/hal+xml
```

DATUM ACADEMY

REST API – Detailed Example

Basic REST API implementation & Testing

eMBDS

REST – Implementation example

- Basic model with two entities

```
class Book {

    String title
    String author
    Date dateCreated

    static belongsTo = [library: Library]
}
```

```
class Library {

    String name
    String address

    static hasMany = [books: Book]
}
```

REST – Implementation example

- Initialisation data

```
class BootStrap {
    def init = { servletContext ->
        new Library(name: "Library name", address: "Library address")
            .addToBooks(new Book(title: "Library's first book", author: "Unknown"))
            .addToBooks(new Book(title: "Library's second book", author: "Unknown"))
            .addToBooks(new Book(title: "Library's last book", author: "Unknown"))
            .save(flush:true, failOnError: true)
    }
}
```

REST – Implementation example – Resources

- Resource available

- Library
  - Library resource representation
    http://my.server.com/library/{id}
  - Library collection representation
    http://my.server.com/libraries

- Book
  - Book resource representation
    http://my.server.com/book/{id}
  - Book collection representation
    http://my.server.com/books

REST – Implementation – Available actions

DATUM ACADEMY

URI	Description	Réponses
GET http://server/library/ GET http://server/libraries/	Retrieve library list	200 OK + Resource collection representation 404 / …
POST http://server/library/ http://server/libraries/	Create library	201 OK 404 / …
GET http://server/library/id	Retrieve library from id	200 OK + Resource representation 404 / …
PUT http://server/library/id	Update library	200 OK 404 / …
DELETE http://server/library/id	Delete library	200 OK 404 / …

REST – Implementation – Available actions

URI	Description	Réponses
GET http://server/library/ GET http://server/libraries/	Retrieve library list	200 OK + Resource collection representation 404 / …

- Two ways to access resource collection (GET / POST)
- http://server/library
  - Considered by many as wrong or not enough precise
- http://server/libraries
  - Considered as the best option
- Some may use both with proper redirection

REST – Implementation – Available actions

URI	Description	Réponses
POST http://server/library/ http://server/libraries/	Create library	201 OK 404 / …
PUT http://server/library/id	Update library	200 OK 404 / …

- Important to differentiate resource from resource collection
- GET and POST requests will target resource collection
- GET / PUT / PATCH / DELETE requests will target a simple resource
  - Identified by ID

REST – Implementation – Available Actions

GET http://server/libraries/


```
{
    id: 1,
    title: "Library's first book",
  - library: {
        id: 1
    },
    dateCreated: "2019-10-02T10:10:29Z",
    author: "Unknown"
}
```



```
[
  - {
        id: 1,
        title: "Library's first book",
      - library: {
            id: 1
        },
        dateCreated: "2019-10-02T10:10:29Z",
        author: "Unknown"
    },
  - {
        id: 2,
```


GET http://server/library/1

REST – Implementation – Testing

- Client Url Request Library
  - Build and execute HTTP requests
  - Retrieve server response

- Options importantes
  - -X HTTP method
  - -I Prints out response header
  - -H Define request header
  - -d Define request data

```
-d '{"title":"Along Came A Spider"}'
```

  - -v Verbose

REST – Implementation – Grails

- Steps for the simplest handmade REST API
- Create a controller to handle client requests « `ApiController` »
- Define methods to handle the differents entry points

```
class Book {

    String title
    String author
    Date dateCreated

    static belongsTo = [library: Library]
}
```

```
class Library {

    String name
    String address

    static hasMany = [books: Book]
}
```

REST – Implementation – Grails

```groovy
// ApiController.groovy
class ApiController {

    /**
     *  Will handle GET / PUT / PATCH / DELETE requests
     */
    def book() {render "ok"}
    /**
     *  Will handle GET / POST requests
     */
    def books() {}

    def libraries() {}

    def library() {}
}
```

REST – Implementation – Grails

- Test it with a cURL request

```
$ curl http://localhost:8080/api/book -I
HTTP/1.1 200
Content-Type: text/html;charset=utf-8
Transfer-Encoding: chunked
Date: Wed, 02 Oct 2019 17:33:29 GMT
```

REST - Implementation

- Basic implementation
  - Handle basic errors
  - Handle request methods
  - Handle basic formats
- In any case
  - Return explicit HTTP code

```
def book() {
    switch(request.getMethod())
    {
        case "GET":
            if (!params.id)
                return response.status = 400
            def bookInstance = Book.get(params.id)
            if (!bookInstance)
                return response.status = 404
            response.withFormat {
                xml { render bookInstance as XML}
                json { render bookInstance as JSON }
            }
        break
        case "PUT":
        break
        case "PATCH":
        break
        case "DELETE":
        break
        default:
            return response.status = 405
        break
    }
    return response.status = 406
}
```

REST – Implementation – Grails

- Test it with a cURL request

```
$ curl http://localhost:8080/api/book/1 -H "Accept: application/json" -X GET -i
HTTP/1.1 200
Content-Type: application/json;charset=UTF-8
Transfer-Encoding: chunked
Date: Wed, 02 Oct 2019 17:47:49 GMT
```

- Data content

```
{
    id: 1,
    title: "Library's first book",
  - library: {
        id: 1
    },
    dateCreated: "2019-10-02T17:45:50Z",
    author: "Unknown"
}
```

REST – Implementation – Grails

- Testing errors as well : unhandled content type

```
$ curl http://localhost:8080/api/book/1 -H "Accept: text/csv" -X GET -i
HTTP/1.1 406
Content-Length: 0
Date: Wed, 02 Oct 2019 17:47:49 GMT
```

- Other case : unhandled HTTP method

```
$ curl http://localhost:8080/api/book/1 -H "Accept: text/json“ -X POST -i
HTTP/1.1 405
Content-Length: 0
Date: Wed, 02 Oct 2019 17:47:49 GMT
```

REST – Implementation – Grails

- Handle format
  - With methods

```
response.withFormat {
    xml { render bookInstance as XML}
    json { render bookInstance as JSON }
}
```

  - With simple logic

```
switch(request.getHeader("Accept"))
{
    case 'json':
    case 'text/json':
    case 'application/json':
        render bookInstance as JSON
        break
    case 'xml':
    case 'text/xml':
    case 'application/xml':
        render bookInstance as XML
        break
}
```

REST – Implementation – Grails

- API generation with Annotations
- Full REST API
- All formats handled

```
@Resource(uri = 'books', formats=['json', 'xml'])
class Book {

    String title
    String author
    Date dateCreated

    static belongsTo = [library: Library]

    static constraints = {
    }
}
```

REST – Implementation – Grails

- Can map REST resources from URLMappings

```
class UrlMappings {
    static mappings = {
        "/books"(resources: "book", excludes:['update','create'])
        "/books"(resources: "book", includes:['index','show','delete'])
        "/books"(resources: "book")
        {
            "/authors"(resources: "author")
        }
    }
}
```

DATUM ACADEMY

REST API – Security

Protection mecanisms

eMBDS

REST – Security

- One of main issues with REST APIs

- Many solutions
  - Authentication
  - Token
  - Signature
  - Single usage signed requests
  - HTTPS

REST – Security – Hash

- Fingerprint of a file / string / …
- Fixed size
- Originally for integrity checks
- Weaknesses : Hash reversal
  - Dictionnaries – Brute force
    - Matching list of strings and hashes
    - 500 most used passwords = 75% of users
    - Few seconds if less than 6 characters (including letters, numbers and special characters)
  - Rainbow tables
    - Same principle but precomputed
    - Trading processing for storage
    - Weak against salted hash (cannot precompute all possible passwords with all possible salt)

REST – Security – Authentication

- Responds to the need to identify the sender of the request

- Classic authentication
  - Send login + password (hash) in the request
  - Always use a "salt" improve safety

- Problems
  - Can brute force easily
  - Credentials sent
  - If the request is intercepted we can have fun on the API with the retrieved identifiers

REST – Security – Token

- Simple implementation to limit exchanges containing credentials
  - The credentials are sent to retrieve a token with a limited lifetime
  - We then use the "Token" to play the role of credentials
  - The token have no obvious link with the credentials
  - We limit the risks

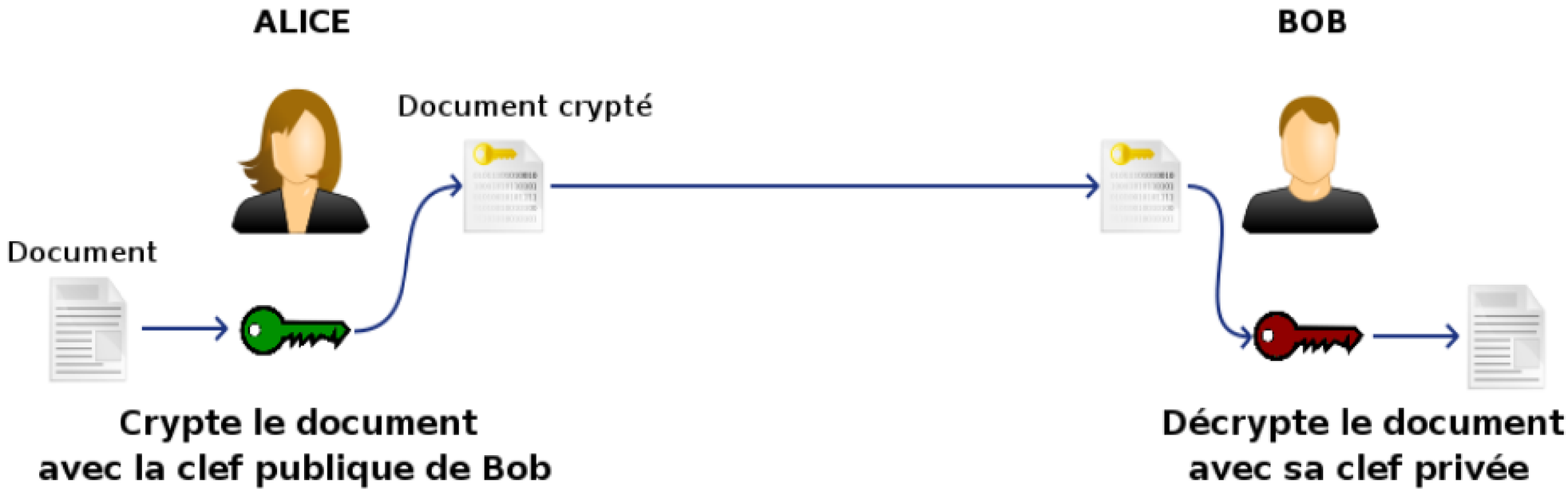
- Problem
  - If the "Token" is intercepted, we can always use the API until the token timeout

REST – Security – Signature

- Aims at guaranteeing the identity of the sender as well as the integrity of the request
  - Using HMAC type algorithms
  - Calculated via a "Hash" in combination with a secret key
  - Sender
    - Encrypt the document and the hash of the document with his private key
  - Receiver
    - Identify the sender by decrypting the whole thing to get the hash (can be created only by the private key wearer)
    - Verify document integrity rebuilding the hash of the document and comparing
- In case of interception, you can not recreate a valid signature for a new request
- Problem
  - If the request is intercepted, it can be replayed

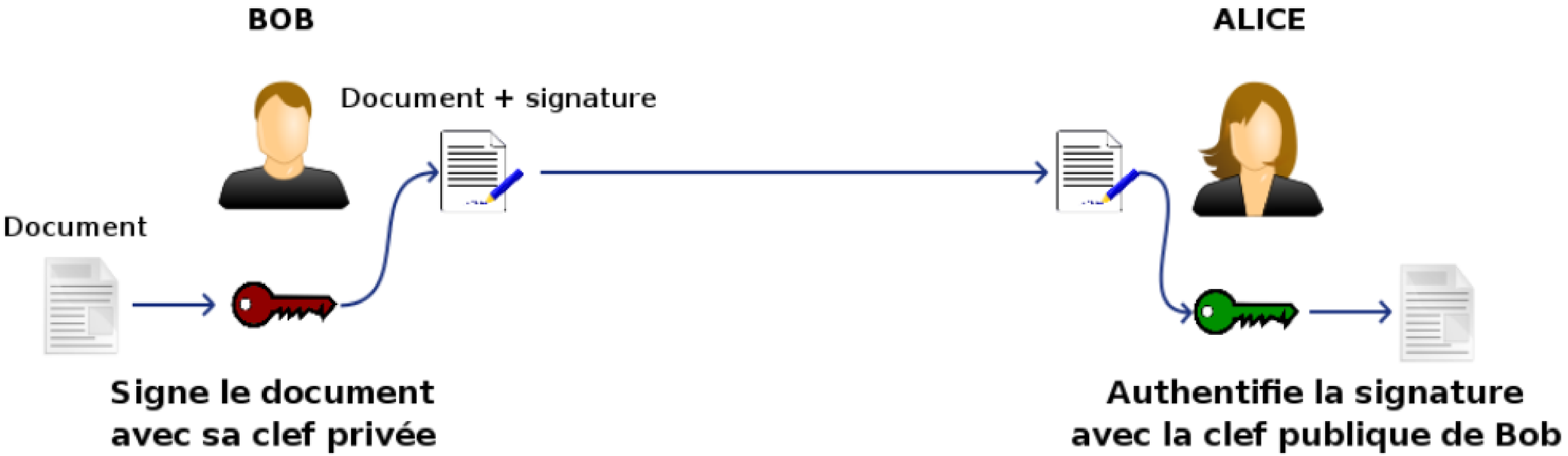
REST – Security – Signature

Encryption/Decryption: Alice souhaite envoyer un document à Bob. Elle dispose de la clef publique de Bob. Bob a également cette clef, ainsi que la clef privée qui lui correspond (et qu'il est le seul à avoir).



REST – Security – Signature

Signature/vérification: Bob souhaite envoyer un document à Alice. Alice pourra le vérifier et garantir qu'il vient bien de Bob.



REST – Security – Single use signed requests

DATUM ACADEMY

- Used by the majority of serious REST API providers
- Additional storage of the "Timestamp" of the last request by the Client and the Server
- Use this timestamp for the creation of the signature
- Overcomes the last remaining problem: even if the request is intercepted, it can not be replayed

REST – Security – HTTPS

- Channel security
- Assurance for the server that the request received is identical to the one sent
- Assurance that the sender is the person claiming to have sent the request
- End-to-end encryption
- Man in the middle useless
- Combined with other methods for added security

REST – Security – HTTPS

- Channel security
- Assurance for the server that the request received is identical to the one sent
- Assurance that the sender is the person claiming to have sent the request
- End-to-end encryption
- Man in the middle useless
- Combined with other methods for added security