JRest

Rajendra Kulkarni
Harish

Topics

1. What is JRest
2. A bit on JSON
3. Benefits of JRest
4. Architecture
5. JRest MPL
6. Writing def files
7. Compiler Internals
8. Execution Engine
9. Session Logic
10. /jrest/auth
11. /jrest/pull
12. /jrest/push
13. Deployment
14. Performance
15. Demo
16. Questions

What is JRest

Is a Meta Programming Language that builds REST services on a webserver automatically, using JSON as definition language

Pre-requisites

- Tools
  - Java
  - Maven (maven.apache.org)
  - Jetty or Tomcat
  - Text editor of your choice
  - Browser
    - Firefox (RESTClient for testing REST Service)
    - Chrome (Postman for testing REST Service)
  - SQL database (MySQL/PostgreSql/SQLServer/Oracle)

Bit on JSON

- *General JSON Body*

```
{ … }
```

- *Defining K, V Pairs*

```
{ “key1” : “value1”,
  “key2” : “value2”,
  . . .
}
```

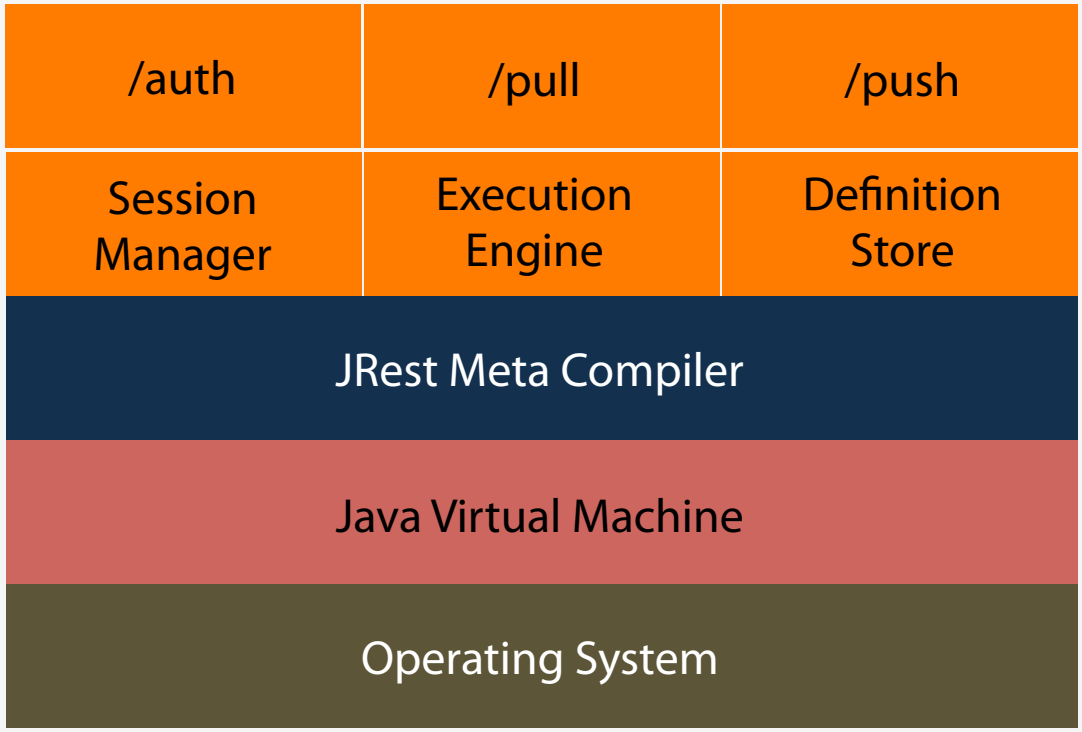
- *Defining Arrays*

```
{ “key1” : “value 1”,
  “key2” : [
              “array-value1”,
              “array-value2”,
              . . .
              ]
}
```

Benefits of JRest

- Innovate faster, less PDLC
- Low Web Server memory footprint
- Application cluster ready *(v 1.1 onwards)*

Architecture



JRest Key Points

- Is a Meta Programming Language (MPL)
- Is "*case sensitive*" just like UNIX!
- DBs *currently supports* MySql, PostgreSql, SqlServer, Oracle; *in future* Elasticsearch, MongoDb,
- Is meant for RESTful service development!
- Interaction with JRest is through *headerparams*; UI must handle *body/form* based requests
- Execution within JRest is always O(1) order!
- Is not an answer to everything

MPL – Environment Variables

- JREST_DEFINITION_PATH → *should point to your *.json repo (JDP)*
- JREST_LOG4J_PROP_FILE → *fully qualified log4j property file name with the path (JLPF)*
- JREST_REFRESH_INTERVAL → *how often you wish JRest to peep to JDP for new files; seconds (JRI)*
- JREST_DB_MAX_CONNECTIONS → *max connections that JRest can have with DB; initial size is 25% of this value (JDMC)*

JRest Key Points

- .jrest → hidden repository under JDP where successfully compiled files are kept; upon reboot files are read from here
- Compiler has a watchdog service which looks at JDP for new or modified files every JRI seconds; keep lower value during dev but higher during deployment
- JDMC allocation happens in equal 25% growth rate; watchdog also cleans this up ☺; so no tension on excesses connections to DB
- In doubt take a look at log files (*usually catalina.out or jrest.log or your log4j log files*)
- JREST_KEY must be *unique* across all the definition files

Keywords

- Query
  - Represents a database statement. *A semicolon (;) at the end of each statement is mandatory this is easier to make mistake beware*
- Delim
  - Used only with AUTH key; *more details in following slides*
- Type
  - Tells JRest whether a definition is a *pull (GET) or push (SET)*; within JDBC definition represents database type (*MySql, SQLServer, PostgreSql, Oracle)*
- Roles
  - Strings or ids that restricts the access to a REST API
- Before
  - Represents a public function of a Java class that will be invoked before calling the REST API
  - Output of Before is passed as data to REST call; this is optional
- After
  - Opposite of Before; will be called after REST API
  - Output of REST, original JSON data and result of Before *(if both are used)* is passed to After

Reserved Words/Chars

- AUTH
- JDBC

Reserved JREST_KEY

- GET
- SET

REST call Types

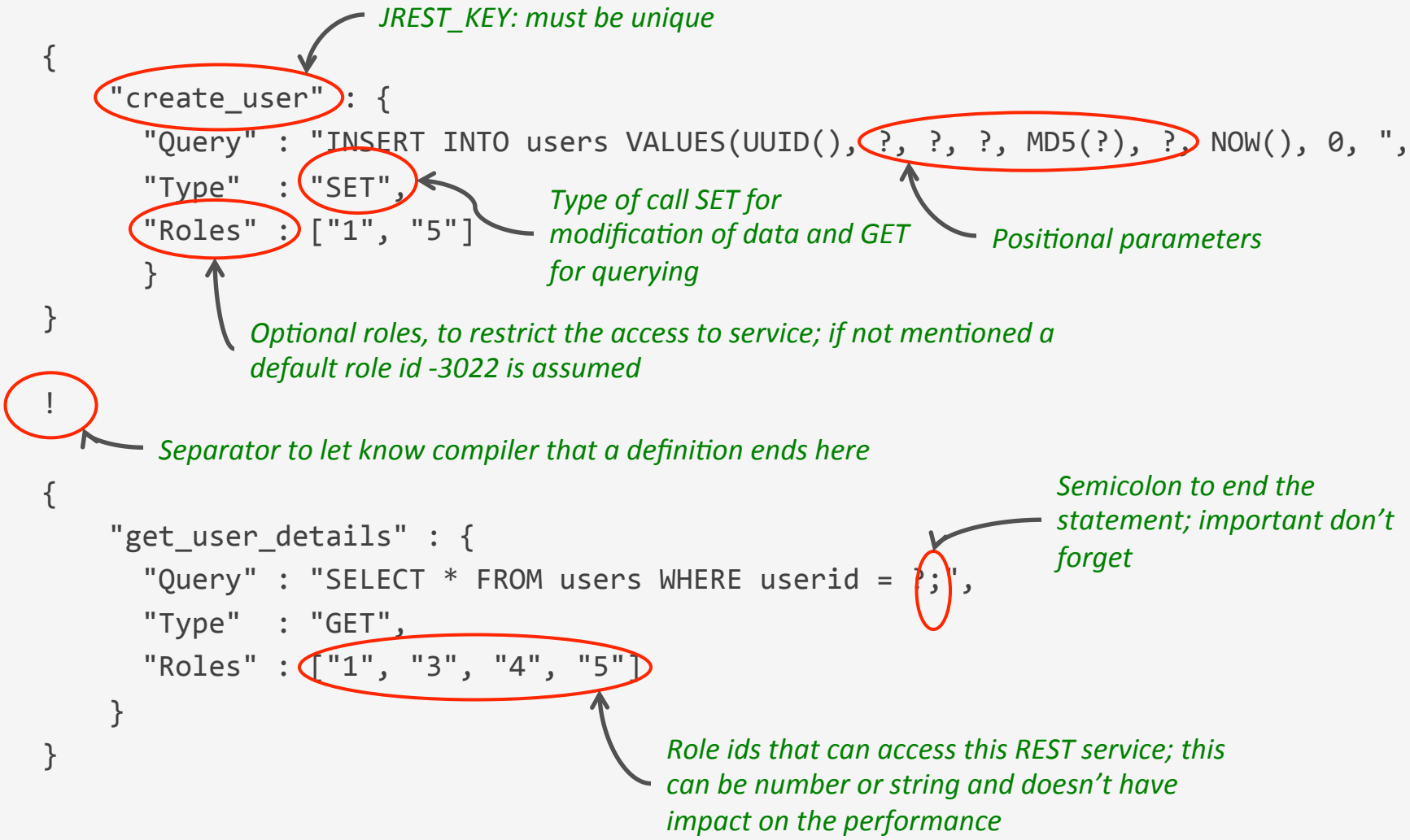
- DB – database name
- FQCN – fully qualified class name
- Method
- Consume
- t/f (true/false)
- Host
- Port
- User
- Pass
- .json – definition file extension
- MySql
- PostgreSql
- SQLServer
- Oracle

Supported Database Types

- ? – positional parameter
- ! – definition separator
- ,
- #
- :
- ;

Valid set of delimiters

Sample Definition File



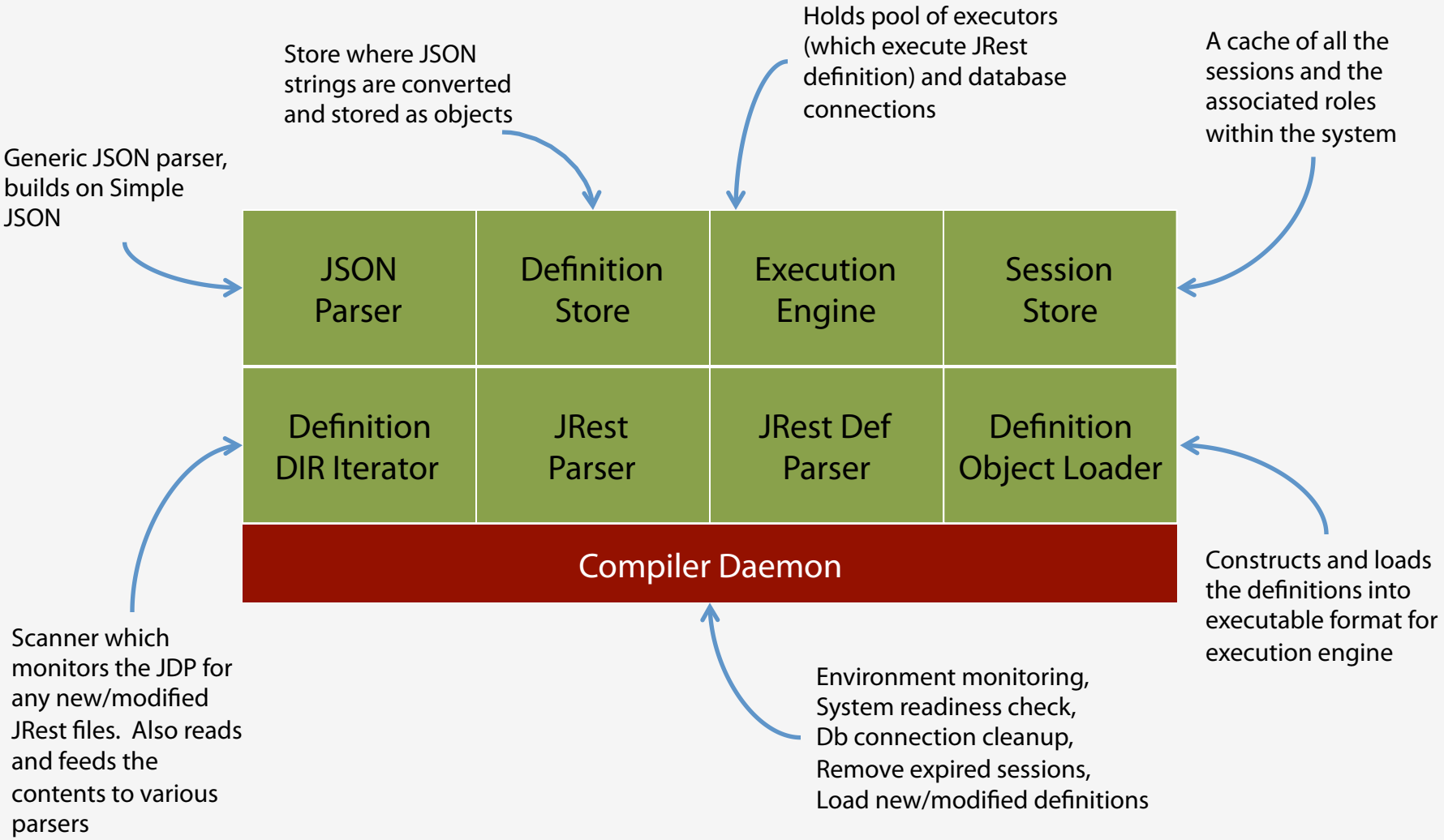
Writing Definition File

- Enclose every definition within { }
- Definition within jrest.json is limited to AUTH and JDBC details only; so don't put anything there
- A single file can have multiple definitions; separate them using ! (bang) character
- Like URIs the JRest key must be unique in your project or deployment
- Put relevant definitions into a single file; *this helps in fishing out the errors*
- Put SET and GET types for a module into separate files; helps a lot during development, *e.g. user-get.json and user-set.json*
- Try your queries (SQL and NoSQL) before you put them into definition files; *helps in ruling out your issues with JRest issues!*
- Don't worry about employing Before and After, they are designed for high performance
- Group cascading business logic into a single function, if you intend to use Before or After

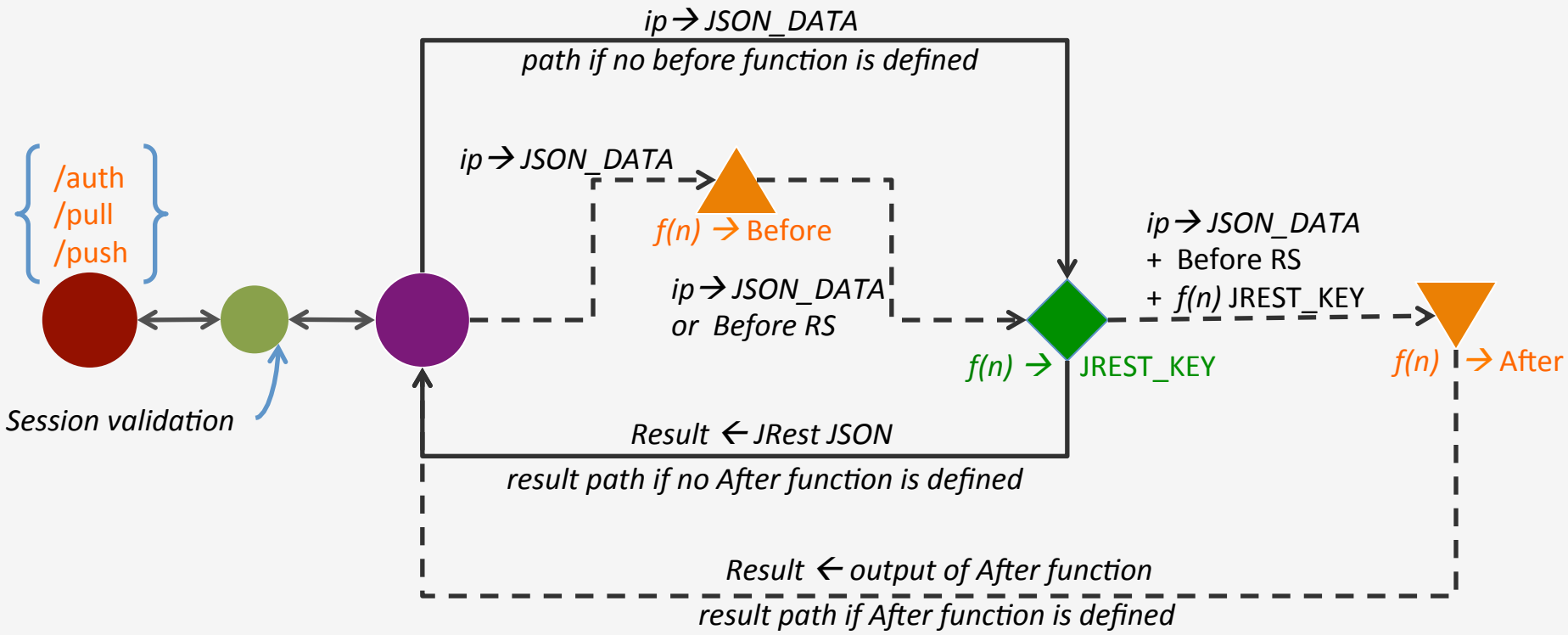
jrest.json

```
{
  "AUTH" : {
    "Query" : "SELECT roles FROM Users WHERE MD5(username) = ? AND password
= MD5(?);",
    "Delim" : ",”
  }
}

!

{
  "JDBC" : {
    "Host" : "localhost",
    "Port" : "3306",
    "User" : "root",
    "Pass" : "xmc4vhcf",
    "Db"   : "Darwin",
    "Type" : "MySql”
   }
}
```

Compiler


Store where JSON strings are converted and stored as objects
Holds pool of executors (which execute JRest definition) and database connections
A cache of all the sessions and the associated roles within the system
Generic JSON parser, builds on Simple JSON
JSON Parser
Definition Store
Execution Engine
Session Store
Definition DIR Iterator
JRest Parser
JRest Def Parser
Definition Object Loader
Compiler Daemon
Scanner which monitors the JDP for any new/modified JRest files. Also reads and feeds the contents to various parsers
Environment monitoring,
System readiness check,
Db connection cleanup,
Remove expired sessions,
Load new/modified definitions
Constructs and loads the definitions into executable format for execution engine

Inside Execution Engine



ip = Input RS = Result Set

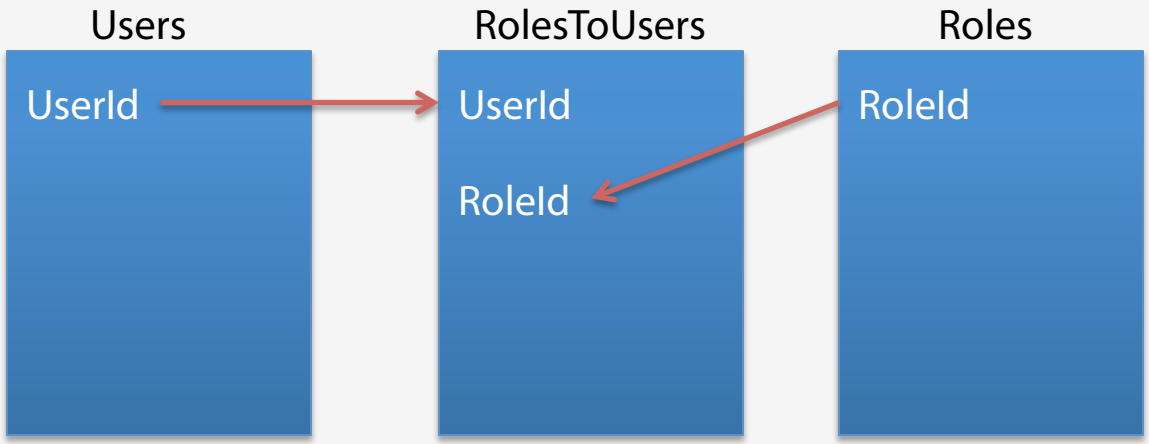
Session, Keys

- Authentication returns MD5 key
- One way encryption used for better security
- Session Key ← CurrentTimeInMillis + 64 bit Long Random number + Random UUID → MD5
- *Beware!* JRest doesn't recognize uniqueness of authentication information
- Multiple *auth* calls with same info generates multiple session keys
- Don't let that bother you, the daemon purges all inactive sessions every JRI seconds
- Your deployment should address *Denial Of Service* attacks; security implementation is not part of JRest

Authentication

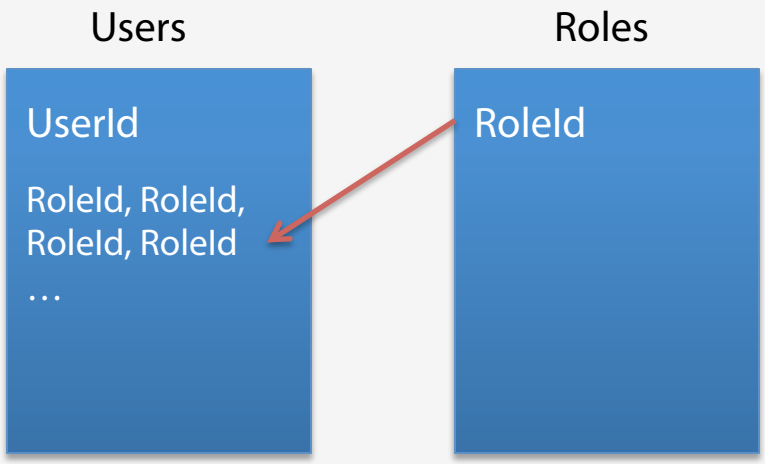
- Has its own session and authentication system
- Authentication is built over Role Based design
- If you don’t have an authentication system of your own and don’t want to implement one then write a query to return -3022 as part of authentication
- When no roles are given for a definition, an automatic role with id -3022 is assumed

Role Based Authentication



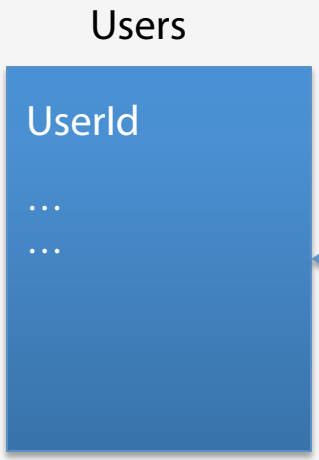

```
{
  "AUTH" : {
    "Query" : "SELECT roles FROM <...> WHERE <...>;”
  }
}
```

Role Based Authentication




```
{
  "AUTH" : {
    "Query" : "SELECT roles FROM <...> WHERE <...>;",
    "Delim" : ","
  }
}
```

Auto Role Authentication



Plain user table

Every user is equally powered; you may want to consider some sort of case or if in SQL statement to differentiate admin to others

```
{
  "AUTH" : {
    "Query" : "SELECT -3022 FROM <...> WHERE <...>;”,
  }
}
```

/jrest/auth

- Header Parameters
  - JSON_DATA → {"1":"<val>", ….} ← must match your "Query" placeholder parameters given in "AUTH"
- Return Values
  - SERVICE_UNAVAILABLE(503) ← if JREST is not ready yet; check log to make sure everything needed by JREST is in right place
  - UNPROCESSABLE_ENTITY(422) ← your "Query" in AUTH is either wrong or not matching number of expected parameters
  - INTERNAL_SERVER_ERROR(500) ← check whether your end-point database is still alive
  - UNAUTHORIZED(401) ← Either the data didn't result in any valid user info or has no roles assigned; check the logs for clarity
  - OK(200) ← when everything is fine and you should have session key as part of entity data in the response

jrest.json

```
{
  "AUTH" : {
    "Query" : "SELECT roles FROM Users WHERE MD5(username) = ? AND password
= MD5(?);",
    "Delim" : ",”
  }
}

!

{
  "JDBC" : {
    "Host" : "localhost",
    "Port" : "3306",
    "User" : "root",
    "Pass" : "xmc4vhcf",
    "Db"   : "Darwin",
    "Type" : "MySql”
   }
}
```

/jrest/pull … push

- Header Parameters
  - SESSION_KEY → key returned by authentication
  - JREST_KEY → key of the definition to be invoked
  - JSON_DATA → any data to be passed to JRest service/Before/After APIs
- Return Values
  - SERVICE_UNAVAILABLE(503) ← if JREST is not ready yet; check log to make sure everything needed by JREST is right place; or your database is not reachable
  - PRECONDITION_FAILURE(412) ← if Before API produces a null result or Before threw an exception
  - UNPROCESSABLE_ENTITY(422) ← your "Query" is either wrong or not matching number of expected parameters
  - EXPECTATION_FAILED(417) ← the After call failed to work the way you had anticipated it to be
  - FORBIDDEN(403) ← your session key is invalid or you don't have necessary permission to execute this JRest service; make appropriate modifications to your definition file
  - NO_CONTENT(204) ← if your Pull call didn't yield any result
  - UNAUTHORIZED(401) ← either the data didn't result in any valid user info or has no roles assigned; check the logs for clarity
  - NOT_FOUND(404) ← check the JREST_KEY you are passing; beware the keys are case sensitive

Performance Overhead

- During our comparative stress test we found that
  - JRest brings in a overhead of 1 – 3ms against hard written Jersey REST service
  - 1 – 2ms overhead on Before and After invocation
  - If you think a particular JRest service is going slow, check your DB or your logic in Before or After
- If you think these overheads are unacceptable you must dump JRest; and good luck with that!
- Yeah 1 – 3ms matters a lot in mission critical applications

Deploying JRest

- Configuring environment variables
  - Minimalistic expectation is to set JREST_DEFINITION_PATH
  - Fine tuning, play with rest of the MPL variables (*ref: slide 9*)
- Running from source
  - cd <source dir>
  - mvn jetty:run
- Deploying war file
  - Copy jrest-x.x.war to your webapps folder