Software Technical Design Specification

for

SBAC-11 Monitoring and Alerting

Version 1.0

Revision History

Date	Version	Description	Author
04-22-2013	1.0	Initial Draft	Russ Hammond
04-29-2013	1.1	Added content, diagrams	Mike Stern

Table of Contents

Software Technical Design – Monitoring and Alerting

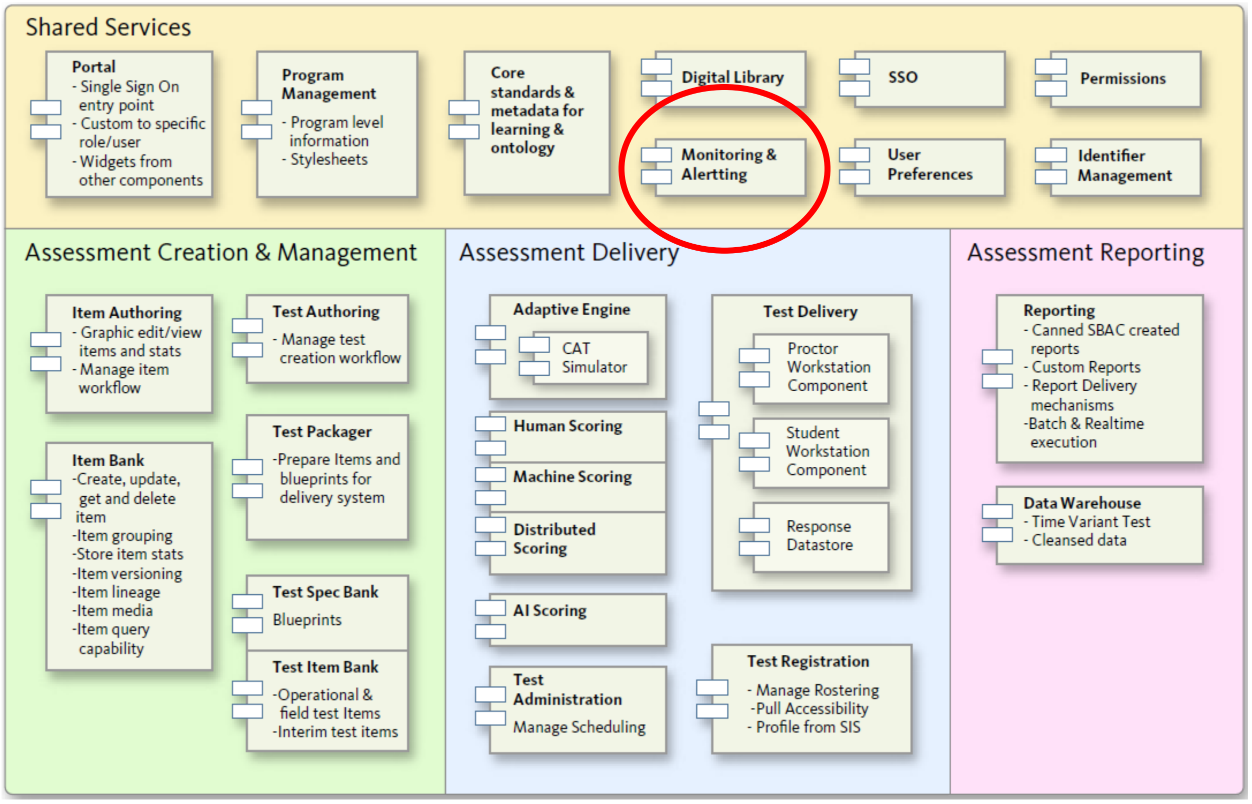
1. Introduction

1.1 Purpose

The objective of this document is to create SBAC-11 Monitoring and Alerting software technical design specifications. All architectural information related to the software within the scope of this document will be included and is intended to be a living document updated as needed throughout the life-cycle of the project. On project close the design document will be included in the project close documentation and ownership of the document will fall to the Production support of the application(s) described within. Production support teams are then responsible for updating documentation upon completion of architecturally significant changes.

1.2 Smarter Balanced Assessment Consortium logical components overview

The diagram below depicts the components of the assessment system. The focus of this document is 'Monitoring and Alerting' which is highlighted in red.



Shared Services
Portal - Single Sign On entry point - Custom to specific role/user - Widgets from other components
Program Management - Program level information - Stylesheets
Core standards & metadata for learning & ontology
Digital Library
SSO
Permissions
Monitoring & Alertting
User Preferences
Identifier Management
Assessment Creation & Management
Item Authoring - Graphic edit/view items and stats - Manage item workflow
Test Authoring - Manage test creation workflow
Test Packager -Prepare Items and blueprints for delivery system
Item Bank -Create, update, get and delete item -Item grouping -Store item stats -Item versioning -Item lineage -Item media -Item query capability
Test Spec Bank Blueprints
Test Item Bank -Operational & field test Items -Interim test items
Assessment Delivery
Adaptive Engine
CAT Simulator
Human Scoring
Machine Scoring
Distributed Scoring
AI Scoring
Test Administration Manage Scheduling
Test Delivery
Proctor Workstation Component
Student Workstation Component
Response Datastore
Test Registration - Manage Rostering -Pull Accessibility - Profile from SIS
Assessment Reporting
Reporting - Canned SBAC created reports - Custom Reports - Report Delivery mechanisms -Batch & Realtime execution
Data Warehouse - Time Variant Test - Cleansed data

2. Design Goals, Constraints and Assumptions

2.1 Design Goals

- Monitor system health
- Monitor component availability
- Provide consistent framework for persisting logs and errors, metrics, and alerts
- Allow searchable access to Monitoring and Alerting messages
- Allow users to define notification rules based on logs, metrics and errors
- Define groups of users to receive email notification of events

The primary purpose of Monitoring and Alerting is to allow administrators to identify situations where the system is experiencing problems which require intervention to keep the system operational. It is responsible for tracking the performance of the servers and the availability of system components, and provides for the centralized persistence and searching of logs and errors, metrics, and alerts. Monitoring and Alerting is designed to serve two audiences: technical administrators and business users. To meet the needs of technical administrators who must monitor the health of the components and hardware, Monitoring and Alerting incorporates the Hyperic HQ open source product. Monitoring and Alerting also provides a custom-coded user interface for technical and non-technical users to search logs and business alerts and set up email notification of desired users. Monitoring and Alerting's services are available to all system components to record important system or business events which are persisted in a central location.

Technical system administrators will access the Hyperic HQ user interface to monitor system health. An agent which monitors server health is deployed to each server in the system. Metrics such as CPU and disk utilization are recorded, and alerts can be configured based on the number of occurrences of an event in a period of time. Technical administrators can identify situations where intervention is required to keep the system operating effectively. Since the open source version of Hyperic HQ does not offer any reporting capabilities, no reporting has been provided with the Monitoring and Alerting component.

Technical and non-technical users can access the custom-coded user interface to search logs and business alerts. Monitoring and Alerting does not send users notification of workflow events, but users are able to check the status of actions they have taken. For example, the addition of new items to an item bank will not cause a notification to be sent to an item approver but the user who requested the upload of items can find the status of the upload through the custom-coded interface. Users are able to create notification rules that scan incoming Monitoring and Alerting entries for a regular expression, and can create notification groups of email addresses which will receive an email when a notification rule has been triggered.

Monitoring and Alerting entries are persisted in a centralized data store for ease of access. Logging is also done on each server to provide information in the case of system failure. MongoDB was chosen as the database used for the centralized persistent data store because MongoDB handles large volumes of data without a defined schema, is cloud-deployable and open source. Monitoring and Alerting data can be managed by using MongoDB database maintenance utilities. No user interface was created for the management of persisted data.

Servers and components are registered and their availability is monitored by Monitoring and Alerting, but each client component is responsible for using the RESTful services provided by Monitoring and Alerting to record significant events. The API for the Monitoring and Alerting services are included in section 13 of this document.

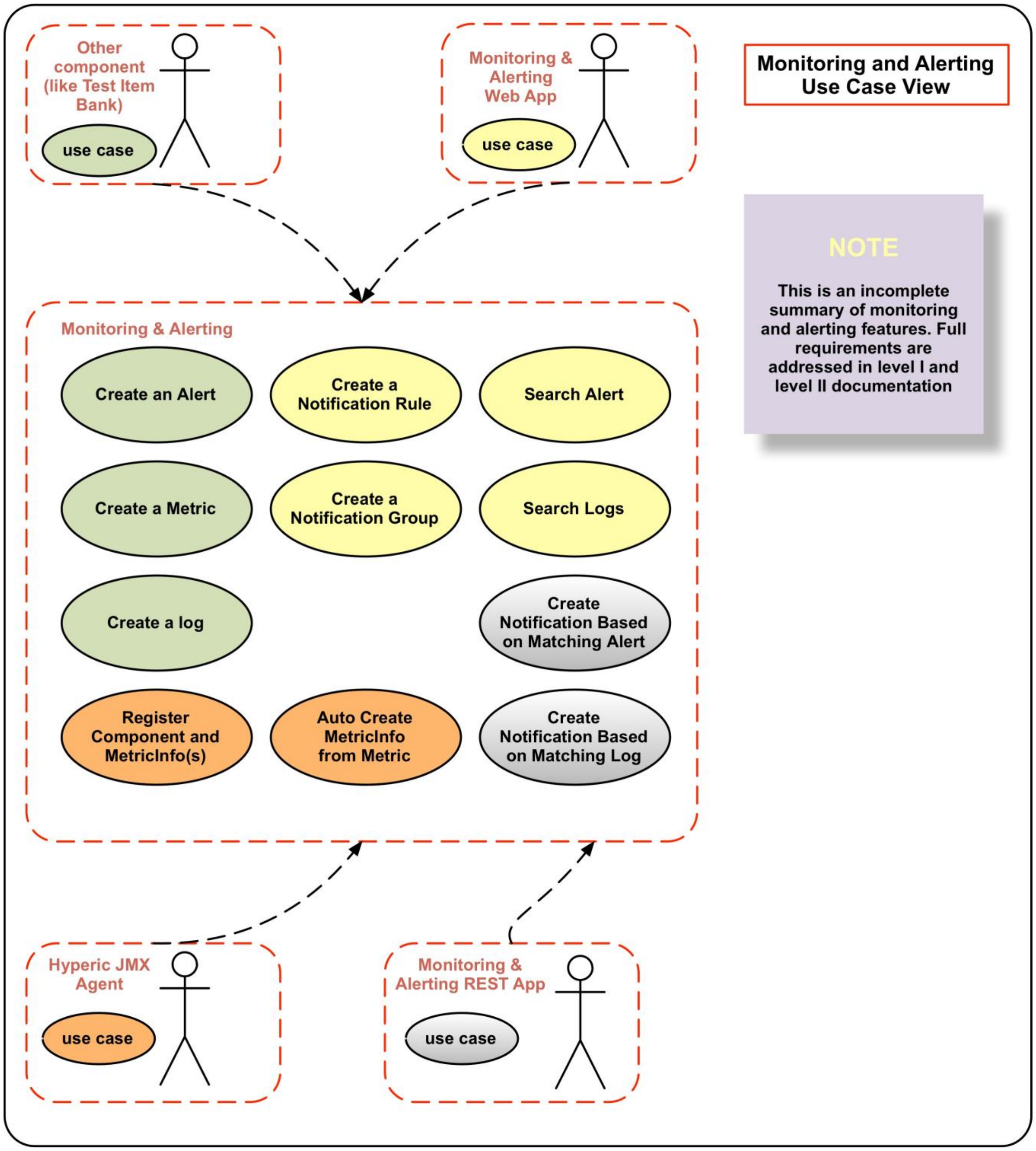
The Smarter Balanced Architecture and Technology Report , section 4.5, states that Monitoring and Alerting is associated with the Test Administration segment of the assessment lifecycle, but Figure 4.2 of the document (repeated in section 1.2 of this document) represents Monitoring and Alerting as a shared component for the entire system. The requirements included in RFP-11 do not limit the scope of Monitoring and Alerting to one portion of the assessment lifecycle; therefore, the component has been created to make services available to all system components. The Monitoring and Alerting component will not be used for managing alerts sent to the proctor workstation during Test Delivery; the Test Delivery component will manage that alerting function. However, the Monitoring and Alerting component is available to the Test Delivery component for recording logs and metrics.

2.2 Design Constraints

- Reporting is limited to the capability of the selected tool No reporting is provided by the open source version of Hyperic HQ.
- Monitoring and Alerting database (MongoDB) will be manipulated via MongoDB utilities rather than creating a user interface in Monitoring and Alerting.
- Logging level will be changed through JConsole and not through the Monitoring and Alerting user interface
- The Monitoring and Alerting component will not be used to send alerts to the Proctor Workstation during Test Delivery. That functionality will be included in the Test Delivery component and its subcomponents. However, the Monitoring and Alerting component is available to Test Delivery for recording logs and metrics as needed.

3. Use-Case View Figure

The following use cases are based on level-II requirements only. It is subject to change based on the review comments.



Other component (like Test Item Bank)
use case
Monitoring & Alerting Web App
use case
Monitoring and Alerting Use Case View
NOTE
This is an incomplete summary of monitoring and alerting features. Full requirements are addressed in level I and level II documentation
Monitoring & Alerting
Create an Alert
Create a Notification Rule
Search Alert
Create a Metric
Create a Notification Group
Search Logs
Create a log
Create Notification Based on Matching Alert
Register Component and MetricInfo(s)
Auto Create MetricInfo from Metric
Create Notification Based on Matching Log
Hyperic JMX Agent
use case
Monitoring & Alerting REST App
use case

4. Process View

4.1 The Monitoring and Alerting component provides services to all components and is not part of a business process.

5. Data View

5.1 Physical Data Model

The Monitoring and Alerting NoSQL physical schema consists of the following collections and nested documents:

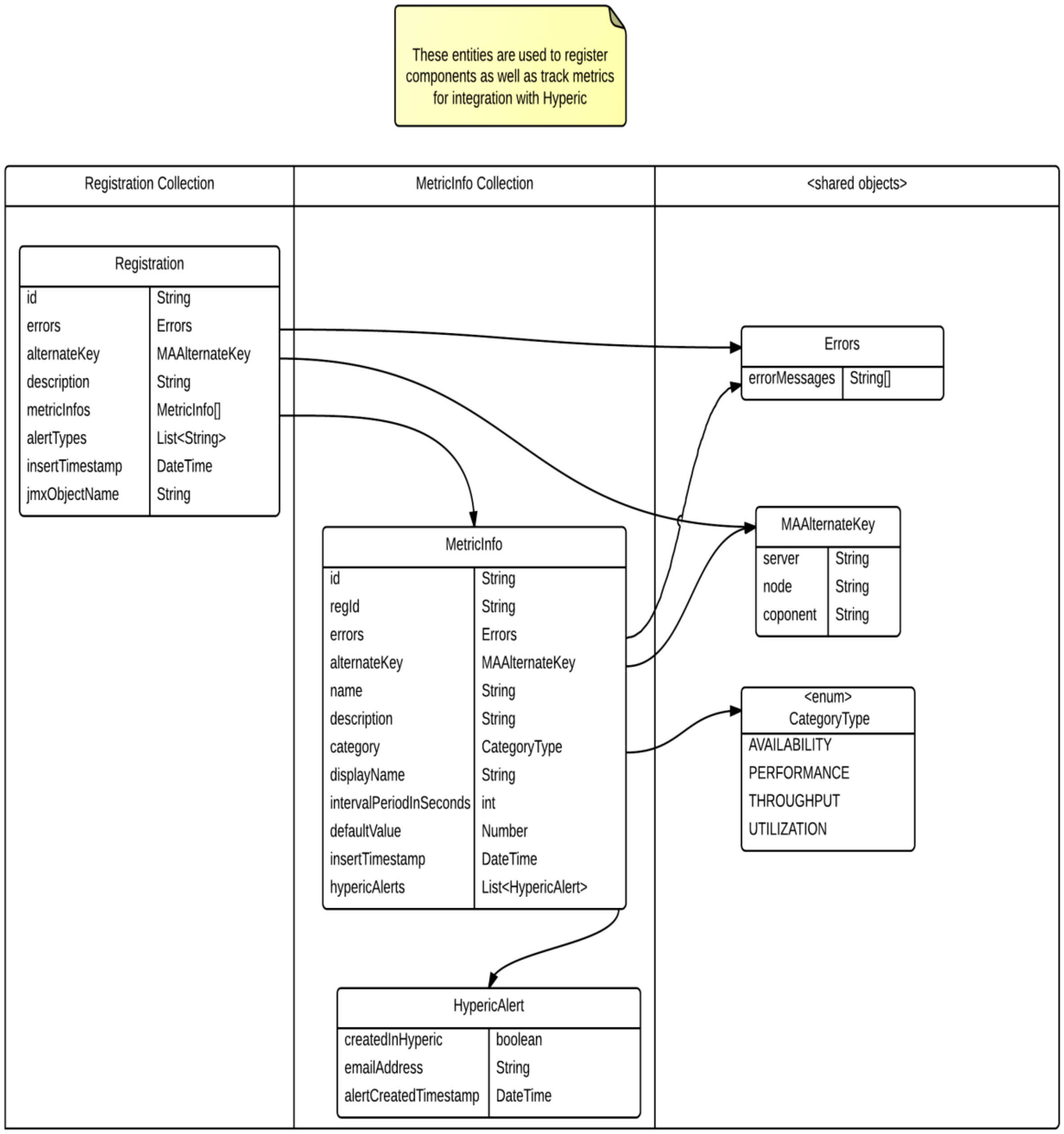


Figure 5.1 - Registration Persistence

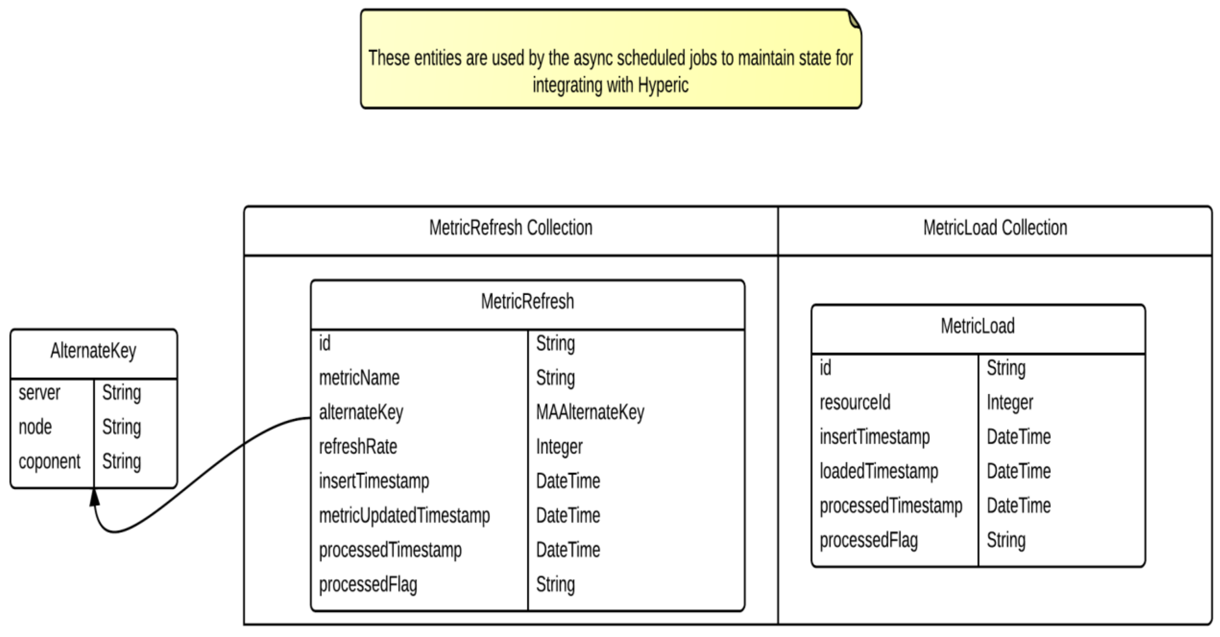


Figure 5.2 - Hyperic Persistence

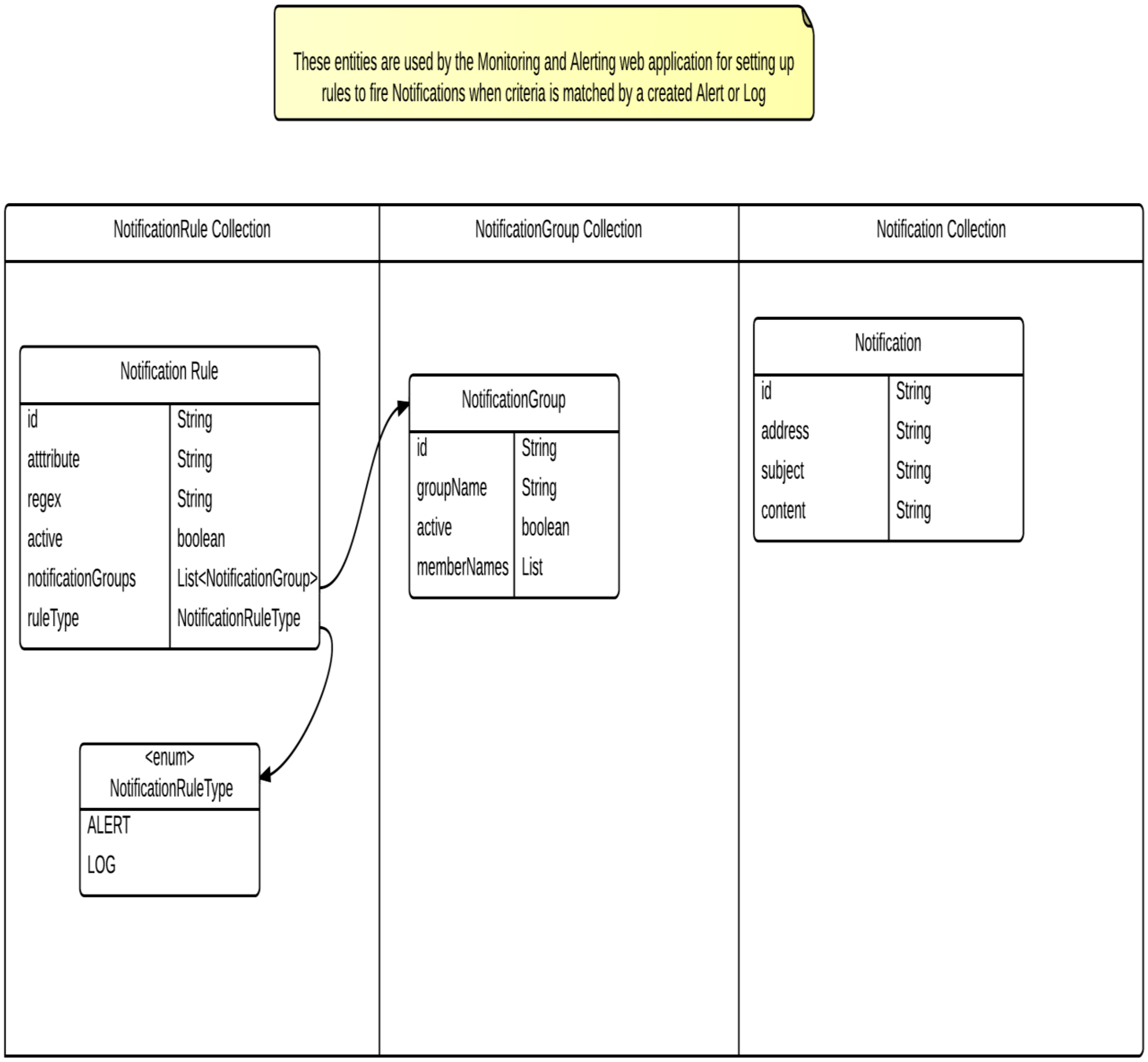

These entities are used by the Monitoring and Alerting web application for setting up rules to fire Notifications when criteria is matched by a created Alert or Log
NotificationRule Collection
NotificationGroup Collection
Notification Collection
Notification Rule
id String
atttribute String
regex String
active boolean
notificationGroups List<NotificationGroup>
ruleType NotificationRuleType
<enum> NotificationRuleType
ALERT
LOG
NotificationGroup
id String
groupName String
active boolean
memberNames List
Notification
id String
address String
subject String
content String

Figure 5.3 - Notification Persistence

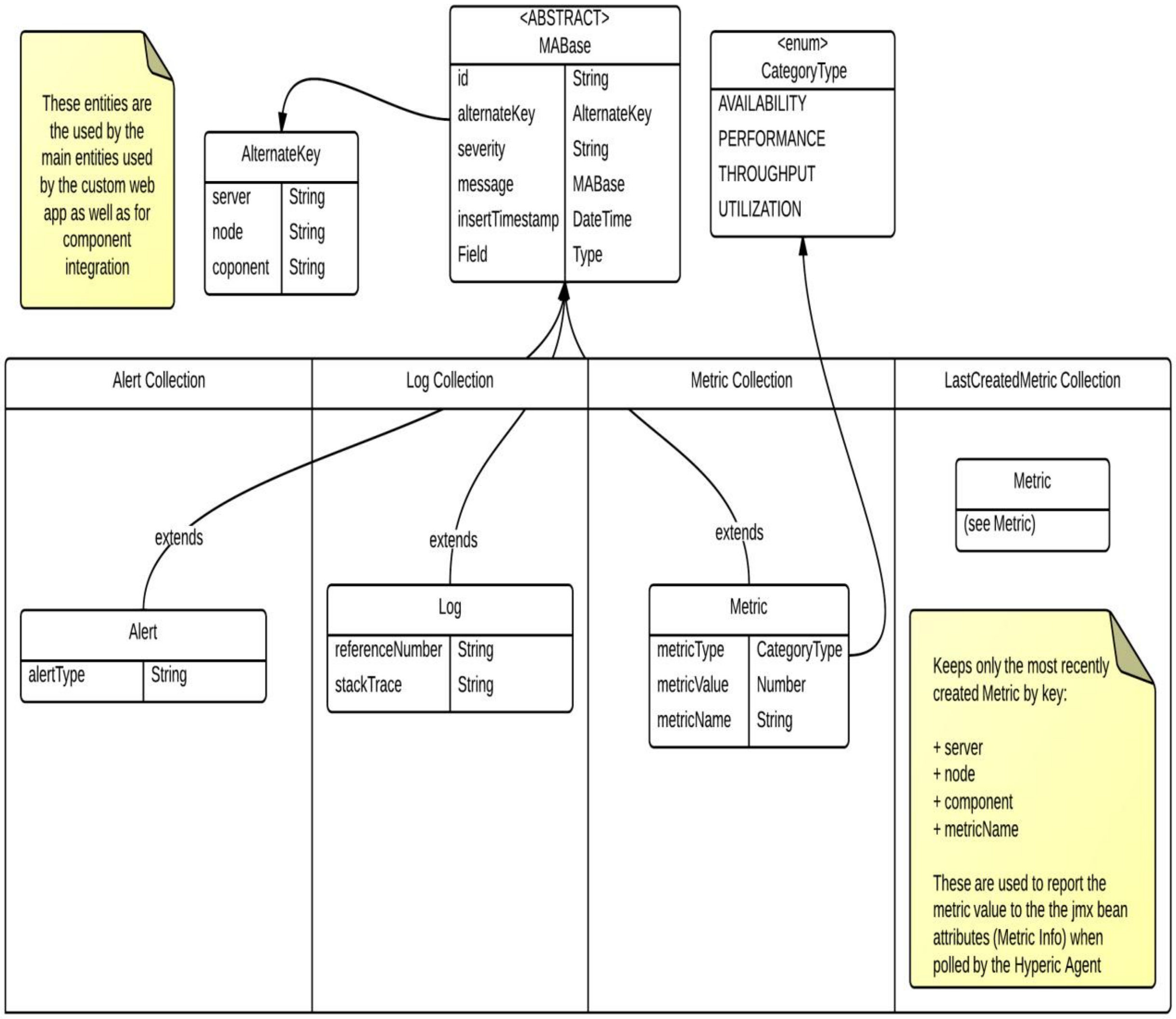


Figure 5.4 - REST Persistence

6. Component Layered View

The component is divided into two major pieces: JMX which is used to register metrics and components and REST endpoints which provide Monitoring and Alerting services to other components.

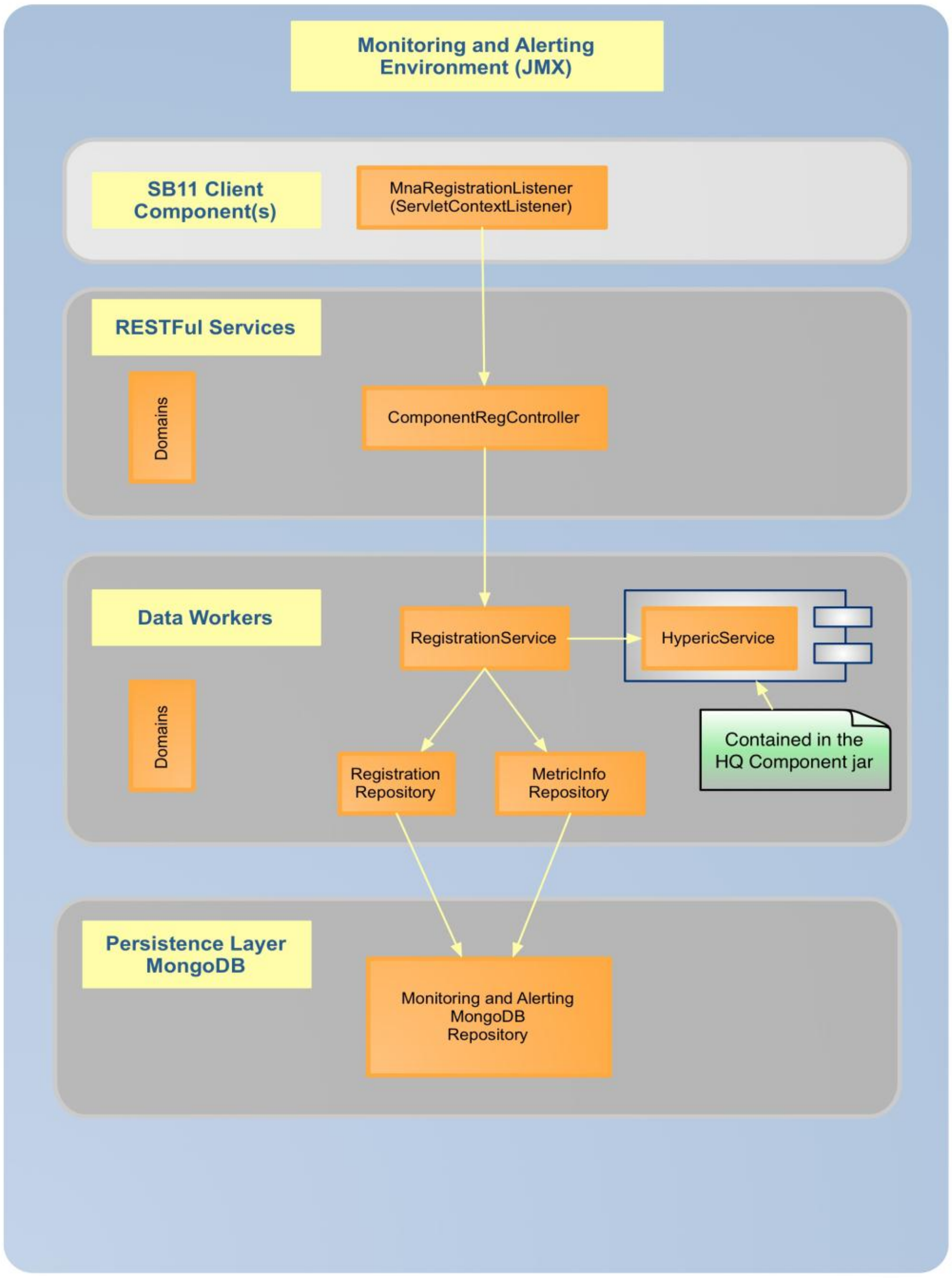
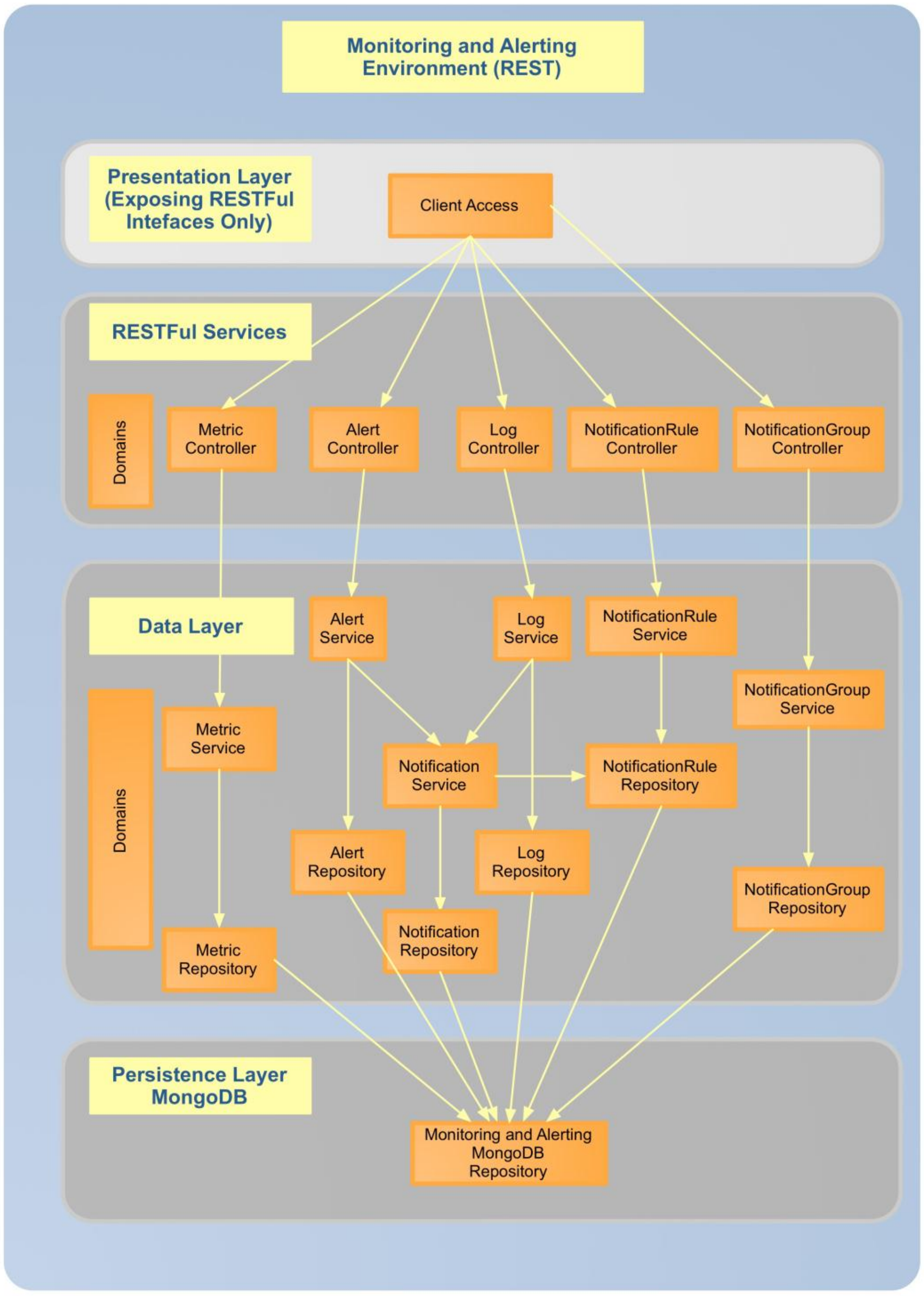

Monitoring and Alerting Environment (JMX)
SB11 Client Component(s)
MnaRegistrationListener (ServletContextListener)
RESTFul Services
Domains
ComponentRegController
Data Workers
RegistrationService
HypericService
Domains
Registration Repository
MetricInfo Repository
Contained in the HQ Component jar
Persistence Layer MongoDB
Monitoring and Alerting MongoDB Repository

Figure 6.1 – JMX



Monitoring and Alerting Environment (REST)
Presentation Layer (Exposing RESTFul Intefaces Only)
Client Access
RESTFul Services
Domains
Metric Controller
Alert Controller
Log Controller
NotificationRule Controller
NotificationGroup Controller
Data Layer
Alert Service
Log Service
NotificationRule Service
NotificationGroup Service
Domains
Metric Service
Notification Service
NotificationRule Repository
Alert Repository
Log Repository
NotificationGroup Repository
Notification Repository
Metric Repository
Persistence Layer MongoDB
Monitoring and Alerting MongoDB Repository

Figure 6.2 - REST

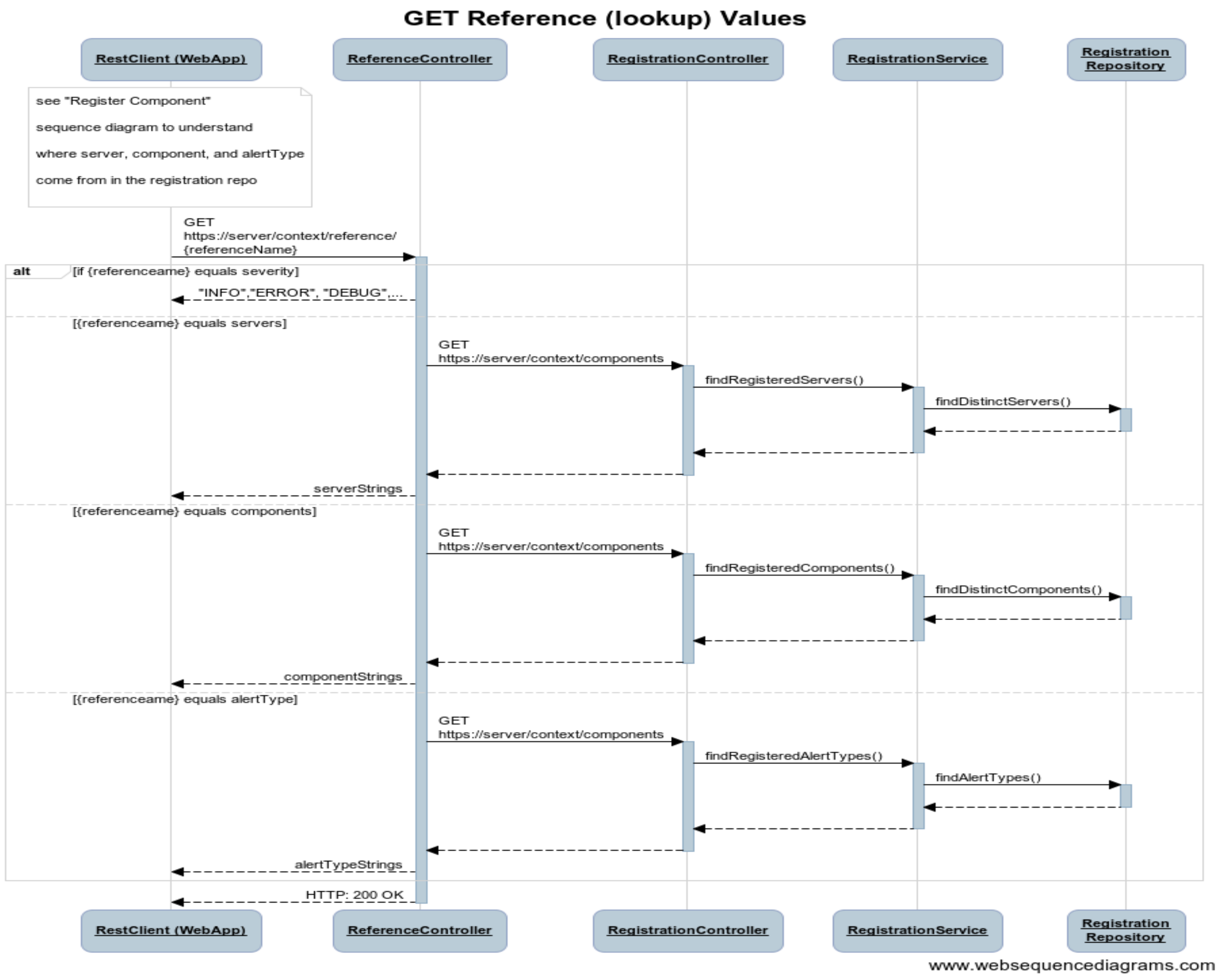
7. Dynamic View

The sequence diagrams have been created to be used as a guided tour through the code. The diagrams do not incorporate every endpoint or process flow, but rather highlight representative patterns within the application's design. The sequence diagrams are best consumed with the code base.

Update and Delete patterns have been omitted since the pattern is very similar to the Create with no significant design points of interest.

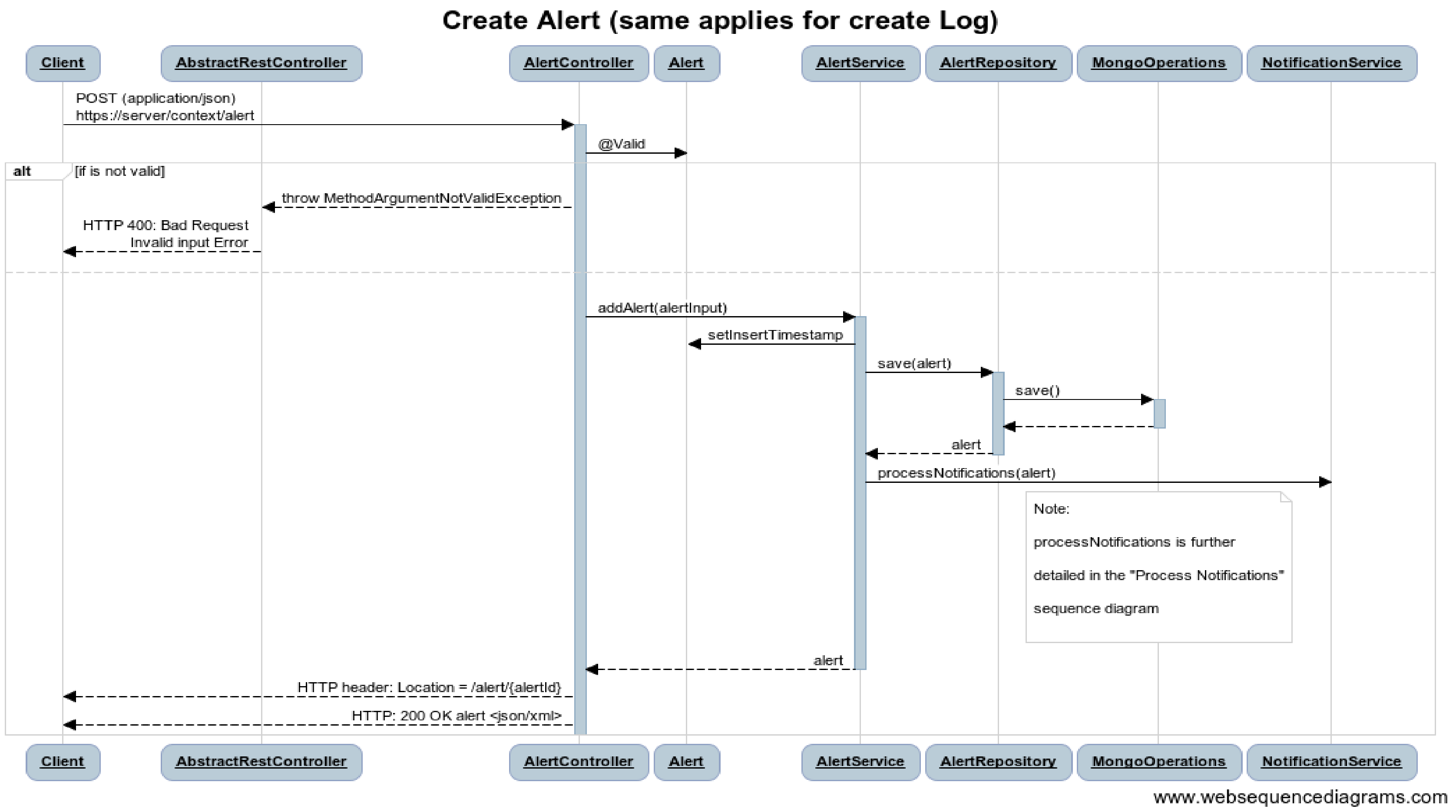
7.1 **Sequence diagram – Get Lookup Values for WebApp**

The following diagram shows the endpoints used for looking up (READ ONLY) reference data used by the webapp. These values are derived from operational and registration data within the Monitoring and Alerting data store.


GET Reference (lookup) Values
RestClient (WebApp)
ReferenceController
RegistrationController
RegistrationService
Registration Repository
see "Register Component" sequence diagram to understand where server, component, and alertType come from in the registration repo
GET https://server/context/reference/{referenceName}
alt
[if {referenceame} equals severity]
"INFO","ERROR", "DEBUG",...
[{referenceame} equals servers]
GET https://server/context/components
findRegisteredServers()
findDistinctServers()
serverStrings
[{referenceame} equals components]
GET https://server/context/components
findRegisteredComponents()
findDistinctComponents()
componentStrings
[{referenceame} equals alertType]
GET https://server/context/components
findRegisteredAlertTypes()
findAlertTypes()
alertTypeStrings
HTTP: 200 OK
www.websequencediagrams.com

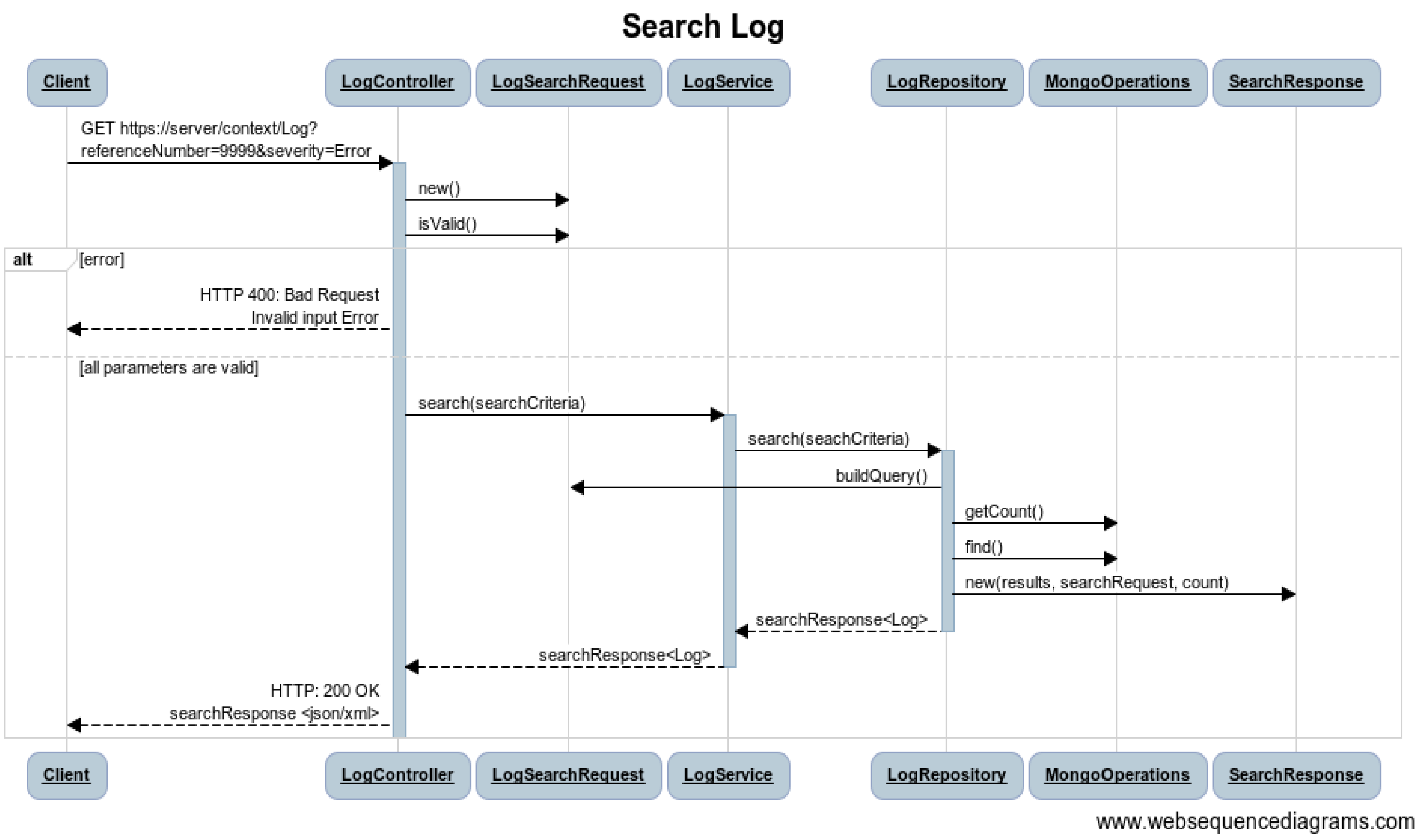
7.2 Sequence diagram – Create Alert

The following diagram is representative of the Create pattern used for most persisted entitites in Monitoring and Alerting. The general pattern applies for most entities appearing in **Section 5.1 Physical Data Model.** Any special cases (such as Create Metric) have been included as separate models.


Create Alert (same applies for create Log)
Client
AbstractRestController
AlertController
Alert
AlertService
AlertRepository
MongoOperations
NotificationService
POST (application/json) https://server/context/alert
@Valid
alt
[if is not valid]
throw MethodArgumentNotValidException
HTTP 400: Bad Request Invalid input Error
addAlert(alertInput)
setInsertTimestamp
save(alert)
save()
alert
processNotifications(alert)
Note: processNotifications is further detailed in the "Process Notifications" sequence diagram
alert
HTTP header: Location = /alert/{alertId}
HTTP: 200 OK alert <json/xml>
www.websequencediagrams.com

7.3 Sequence diagram – Search Log

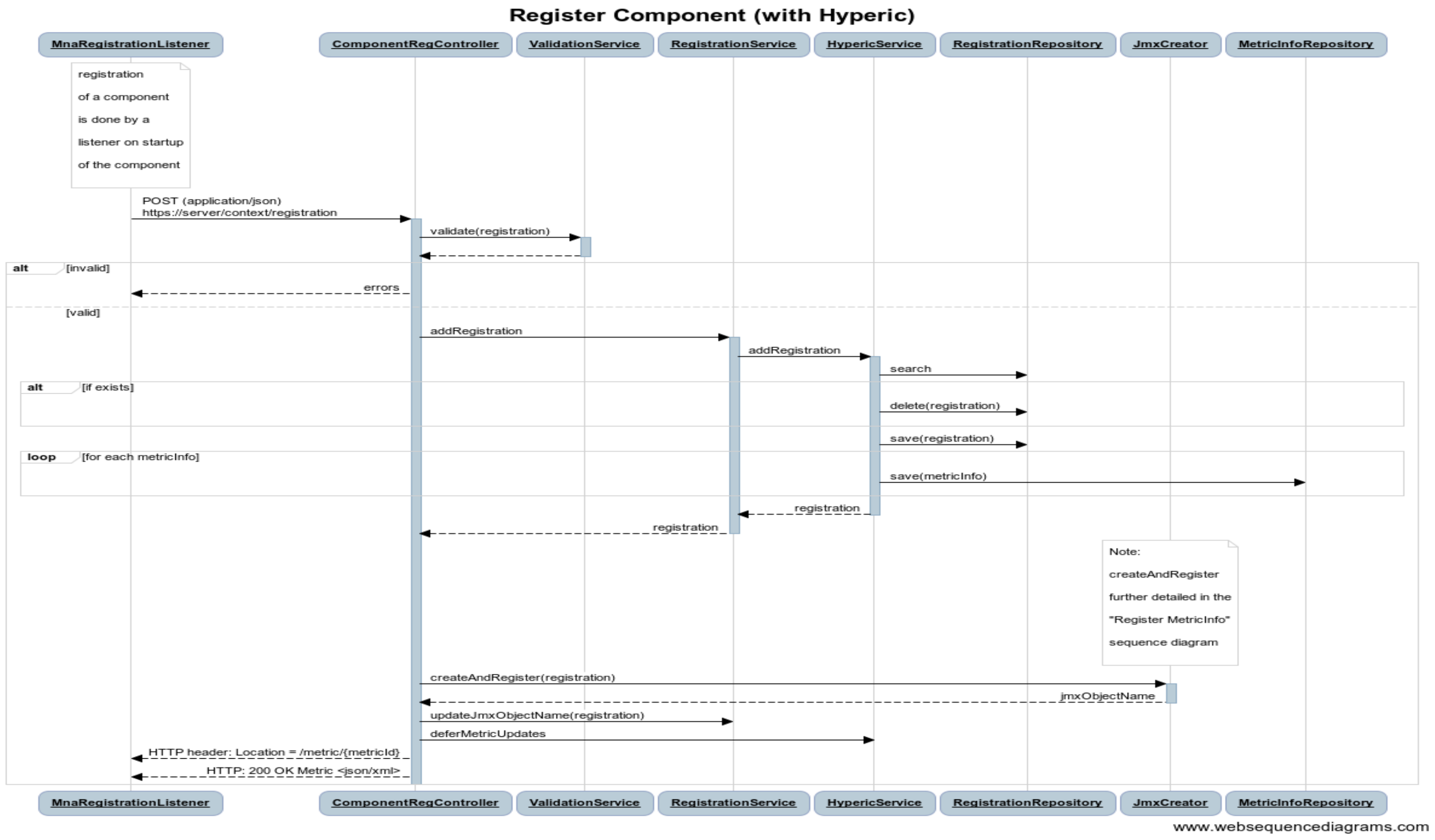
The following diagram is representative of the Search pattern used for most persisted entities in Monitoring and Alerting. The general pattern applies for most entities appearing in **Section 5.1 Physical Data Model**.



Search Log
Client
LogController
LogSearchRequest
LogService
LogRepository
MongoOperations
SearchResponse
GET https://server/context/Log?referenceNumber=9999&severity=Error
new()
isValid()
alt
[error]
HTTP 400: Bad Request Invalid input Error
[all parameters are valid]
search(searchCriteria)
search(seachCriteria)
buildQuery()
getCount()
find()
new(results, searchRequest, count)
searchResponse<Log>
searchResponse<Log>
HTTP: 200 OK searchResponse <json/xml>
www.websequencediagrams.com

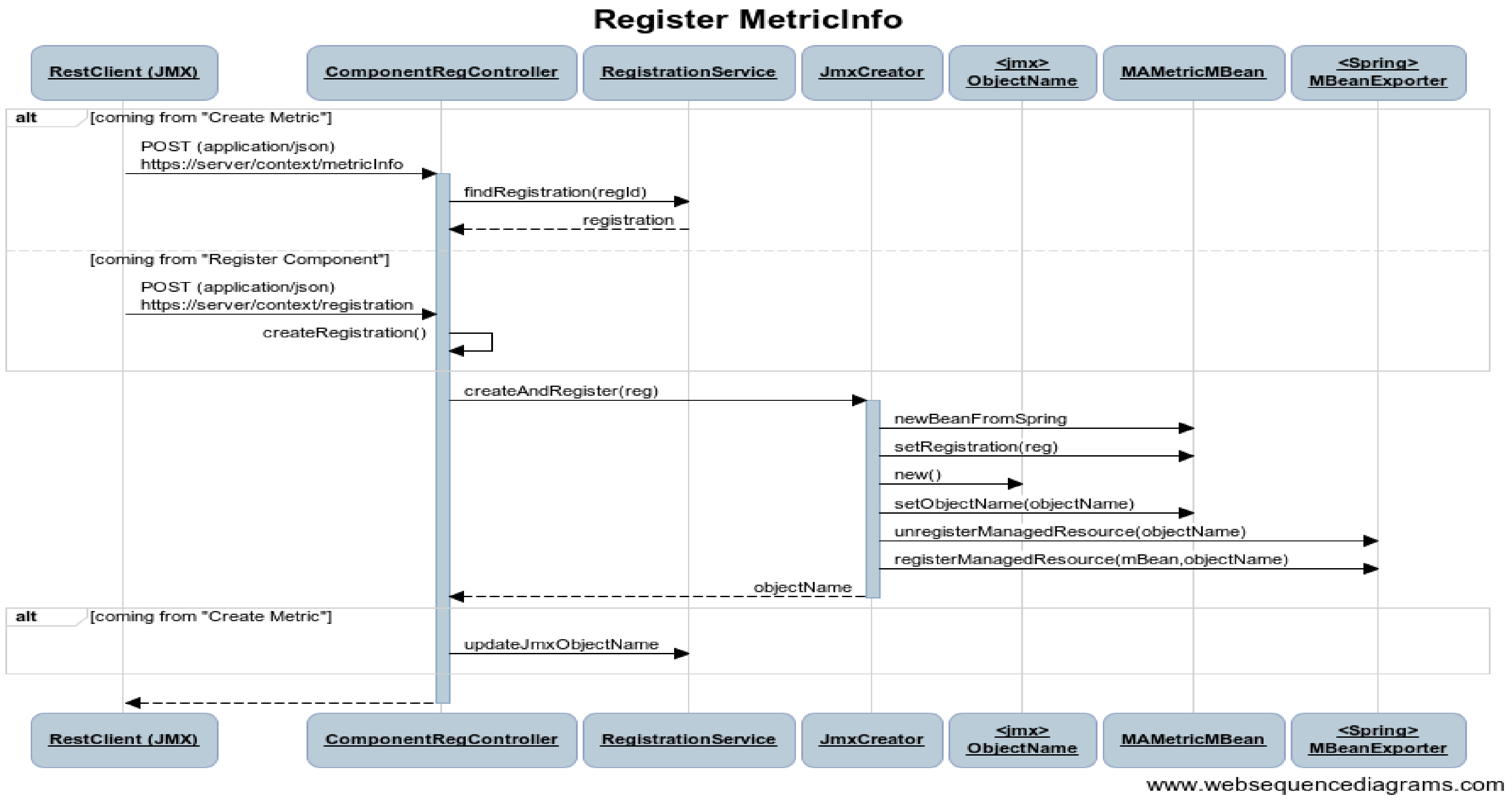
7.4 Sequence diagram – Register Component (with Hyperic)

The following diagram shows the component registration process that integrates into the Hyperic HQ application. By registering a component, administrators can see a component (and its associated metrics) in the Hyperic console. By default, an Availability metric is also registered at startup time. This Availability metric periodically provides a 'heartbeat' that will create an alert if not provided for a period of time.


Register Component (with Hyperic)
MnaRegistrationListener
ComponentRegController
ValidationService
RegistrationService
HypericService
RegistrationRepository
JmxCreator
MetricInfoRepository
registration of a component is done by a listener on startup of the component
POST (application/json) https://server/context/registration
validate(registration)
alt [invalid]
errors
[valid]
addRegistration
addRegistration
search
alt [if exists]
delete(registration)
save(registration)
loop [for each metricInfo]
save(metricInfo)
registration
registration
Note: createAndRegister further detailed in the "Register MetricInfo" sequence diagram
createAndRegister(registration)
jmxObjectName
updateJmxObjectName(registration)
deferMetricUpdates
HTTP header: Location = /metric/{metricId}
HTTP: 200 OK Metric <json/xml>
www.websequencediagrams.com

7.5 Sequence diagram – Register MetricInfo

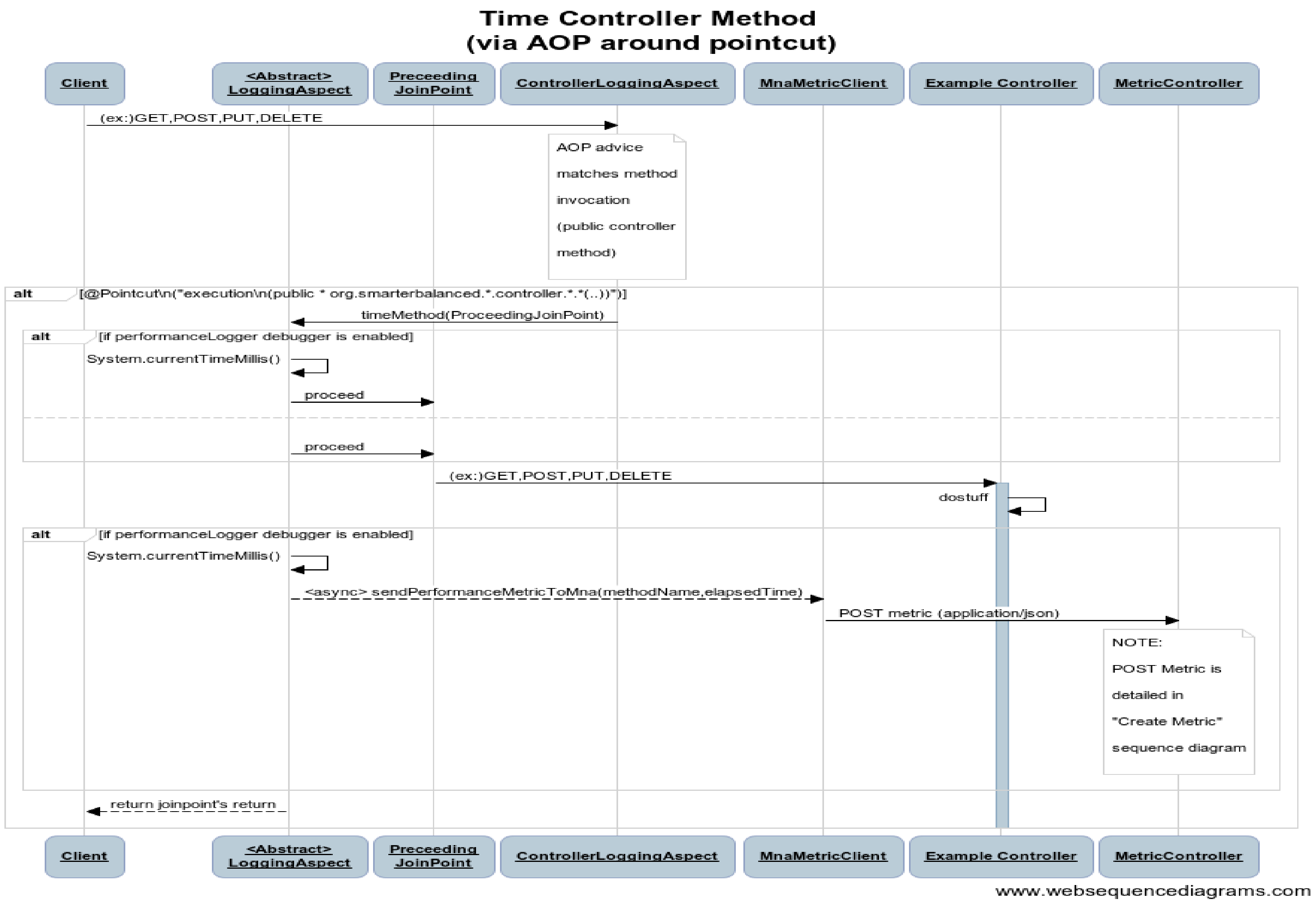
The following diagram shows the metric registration process that integrates a specific data element with the Hyperic HQ application. By registering a metric, administrators can see the components data within the Hyperic console. Some examples of the Smarter Balanced metrics are Availability and Performance (which logs the time taken to perform a specific method). A metricInfo can be registered ahead of time or as needed when a metric is created (see **7.7 Sequence Diagram – Create Metric**).



Register MetricInfo
RestClient (JMX)
ComponentRegController
RegistrationService
JmxCreator
<jmx> ObjectName
MAMetricMBean
<Spring> MBeanExporter
alt
[coming from "Create Metric"]
POST (application/json) https://server/context/metricInfo
findRegistration(regId)
registration
[coming from "Register Component"]
POST (application/json) https://server/context/registration
createRegistration()
createAndRegister(reg)
newBeanFromSpring
setRegistration(reg)
new()
setObjectName(objectName)
unregisterManagedResource(objectName)
registerManagedResource(mBean,objectName)
objectName
alt
[coming from "Create Metric"]
updateJmxObjectName
www.websequencediagrams.com

7.6 Sequence Diagram – Create Performance Metric

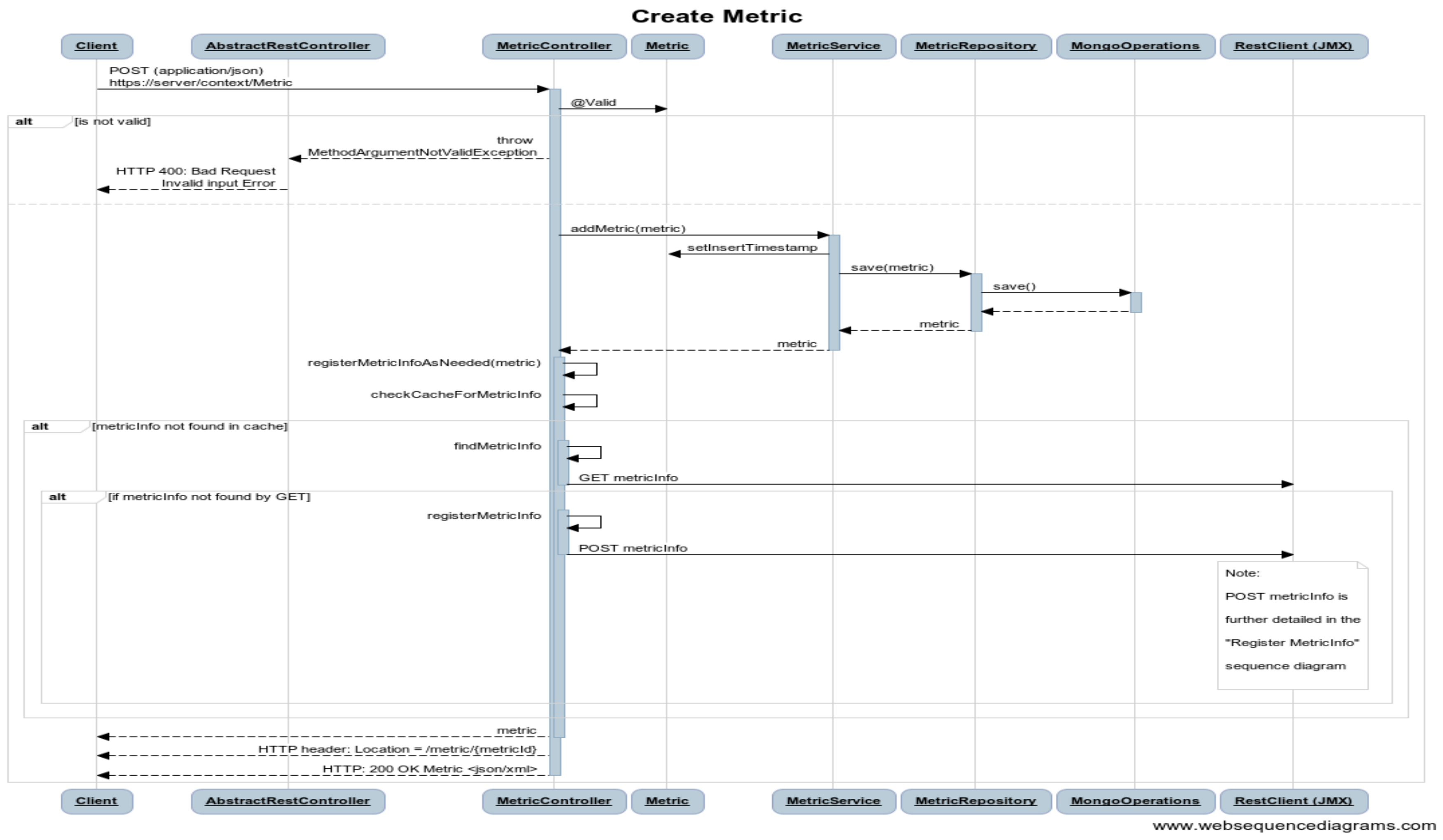
The following diagram shows how a metric is created using Aspect Oriented Programming (AOP). The benefit of AOP is that a method can be 'decorated' to log performance without having to change the logic of the method itself. The use of Spring AOP and Pointcuts separates the concerns. Given that the application may not want to register every possible method that may have its performance measured, MetricInfos can by dynamically registered when a metric is created (see **7.7 Sequence Diagram – Create Metric**).



Time Controller Method
(via AOP around pointcut)
Client
<Abstract> LoggingAspect
Preceeding JoinPoint
ControllerLoggingAspect
MnaMetricClient
Example Controller
MetricController
(ex:)GET,POST,PUT,DELETE
AOP advice matches method invocation (public controller method)
alt [@Pointcut\n("execution\n(public * org.smarterbalanced.*.controller.*.*(..))")]
timeMethod(ProceedingJoinPoint)
alt [if performanceLogger debugger is enabled]
System.currentTimeMillis()
proceed
proceed
(ex:)GET,POST,PUT,DELETE
dostuff
alt [if performanceLogger debugger is enabled]
System.currentTimeMillis()
<async> sendPerformanceMetricToMna(methodName,elapsedTime)
POST metric (application/json)
NOTE: POST Metric is detailed in "Create Metric" sequence diagram
return joinpoint's return
www.websequencediagrams.com

7.7 Sequence Diagram – Create Metric

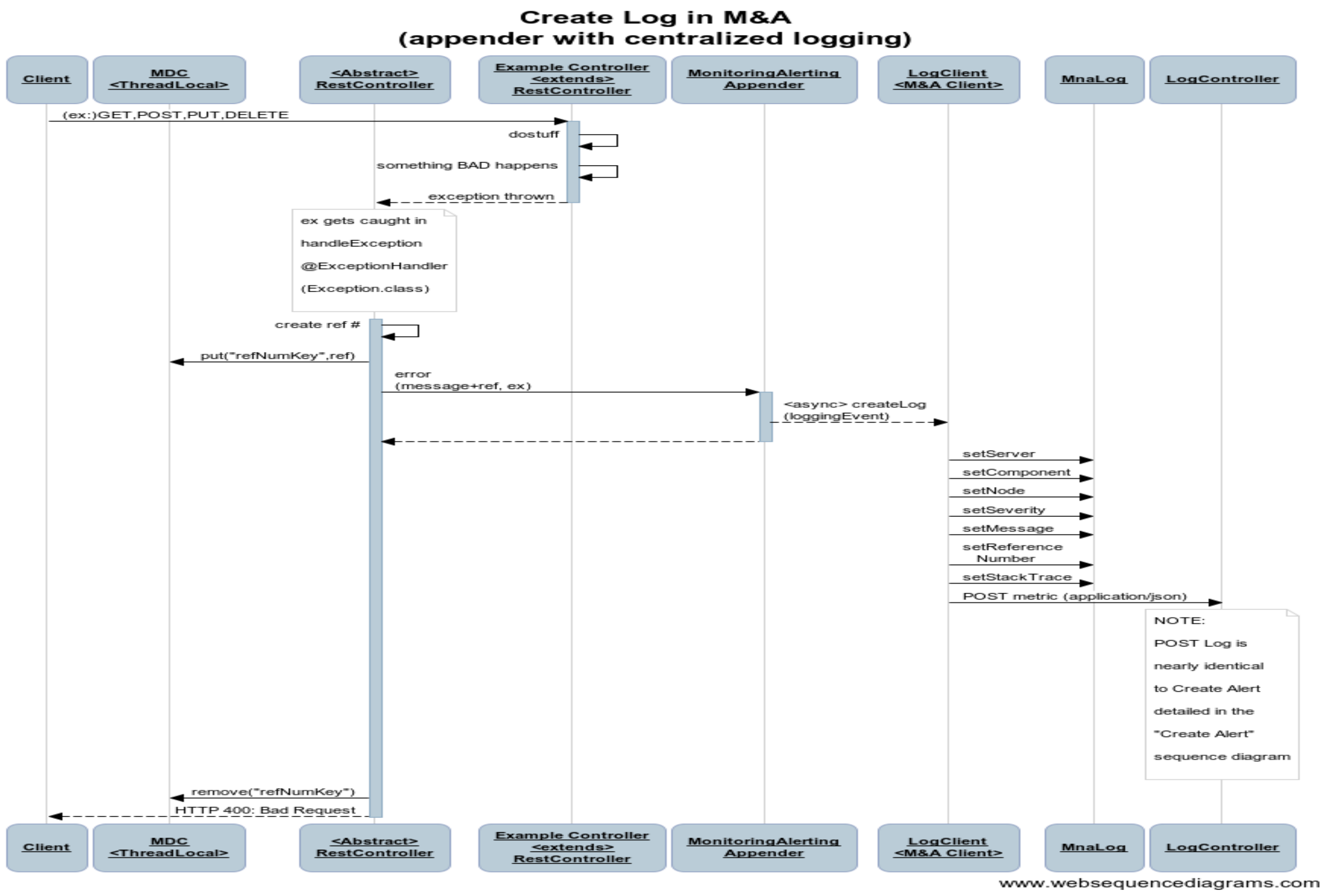
The following diagram illustrates the creation of a metric. Also included in this flow is the dynamic registration of a MetricInfo as needed.


Create Metric
Client
AbstractRestController
MetricController
Metric
MetricService
MetricRepository
MongoOperations
RestClient (JMX)
POST (application/json) https://server/context/Metric
@Valid
alt [is not valid]
throw MethodArgumentNotValidException
HTTP 400: Bad Request Invalid input Error
addMetric(metric)
setInsertTimestamp
save(metric)
save()
metric
metric
registerMetricInfoAsNeeded(metric)
checkCacheForMetricInfo
alt [metricInfo not found in cache]
findMetricInfo
GET metricInfo
alt [if metricInfo not found by GET]
registerMetricInfo
POST metricInfo
Note: POST metricInfo is further detailed in the "Register MetricInfo" sequence diagram
metric
HTTP header: Location = /metric/{metricId}
HTTP: 200 OK Metric <json/xml>
www.websequencediagrams.com

7.8 Sequence Diagram –

The following diagram illustrates the creation of a log. This highlights the use of a custom Log4J appender which utilizes the Monitoring and Alerting Client .jar.

NOTE: The Test Item Bank has been integrated to use this appender to centralize its logging into Monitoring and Alerting.



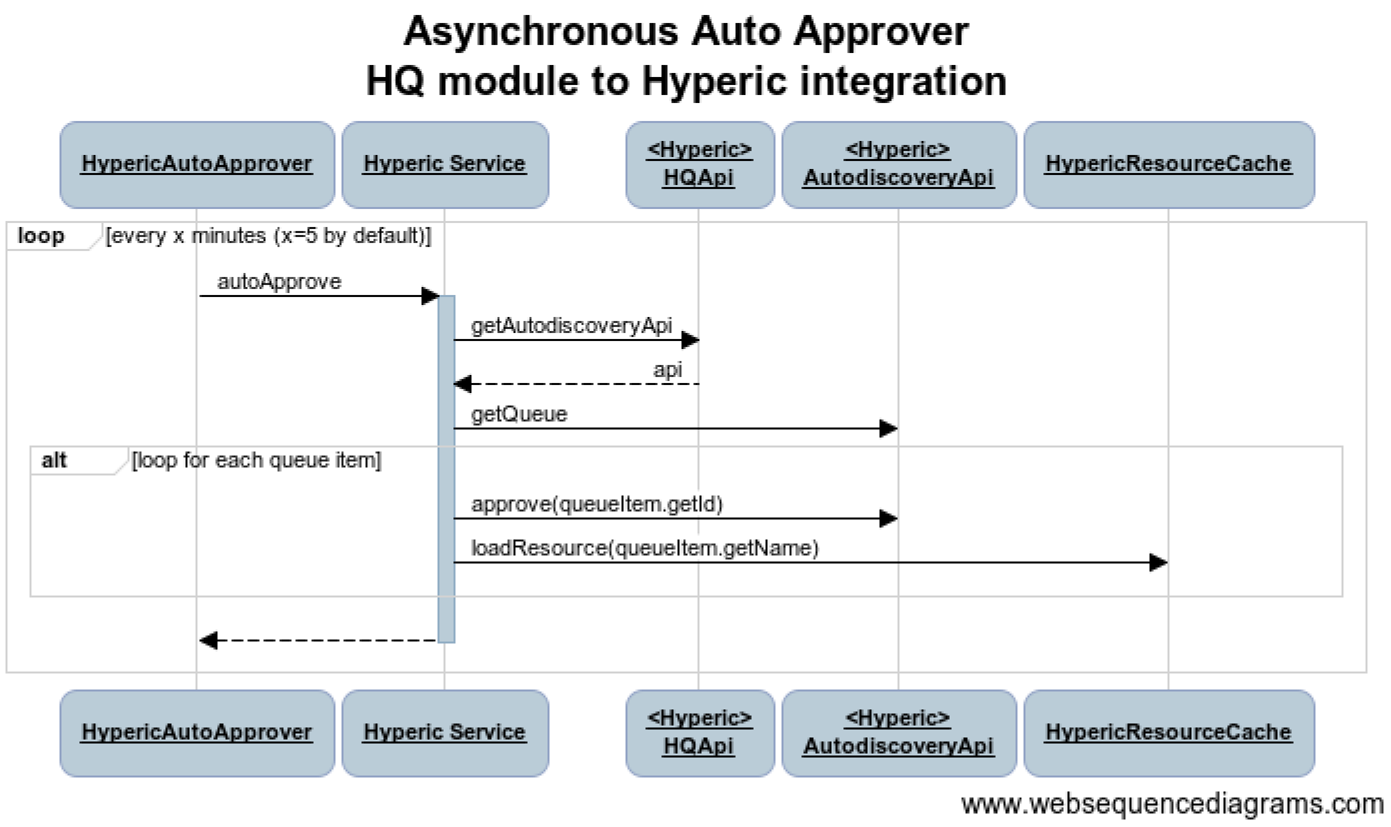
Create Log in M&A
(appender with centralized logging)
Client
MDC <ThreadLocal>
<Abstract> RestController
Example Controller <extends> RestController
MonitoringAlerting Appender
LogClient <M&A Client>
MnaLog
LogController
(ex:)GET,POST,PUT,DELETE
dostuff
something BAD happens
exception thrown
ex gets caught in handleException @ExceptionHandler (Exception.class)
create ref #
put("refNumKey",ref)
error (message+ref, ex)
<async> createLog (loggingEvent)
setServer
setComponent
setNode
setSeverity
setMessage
setReference Number
setStackTrace
POST metric (application/json)
NOTE: POST Log is nearly identical to Create Alert detailed in the "Create Alert" sequence diagram
remove("refNumKey")
HTTP 400: Bad Request
www.websequencediagrams.com

7.9 **Sequence Diagrams –**

The following four sequence diagrams depict asynchronous scheduled jobs that perform various maintenance tasks at regular intervals. The majority of the tasks need to be scheduled due to timing prerequisites constrained by the Hyperic API. Additionally, the dynamic nature of metrics and components requires the ability to change Hyperic without requiring downtime.

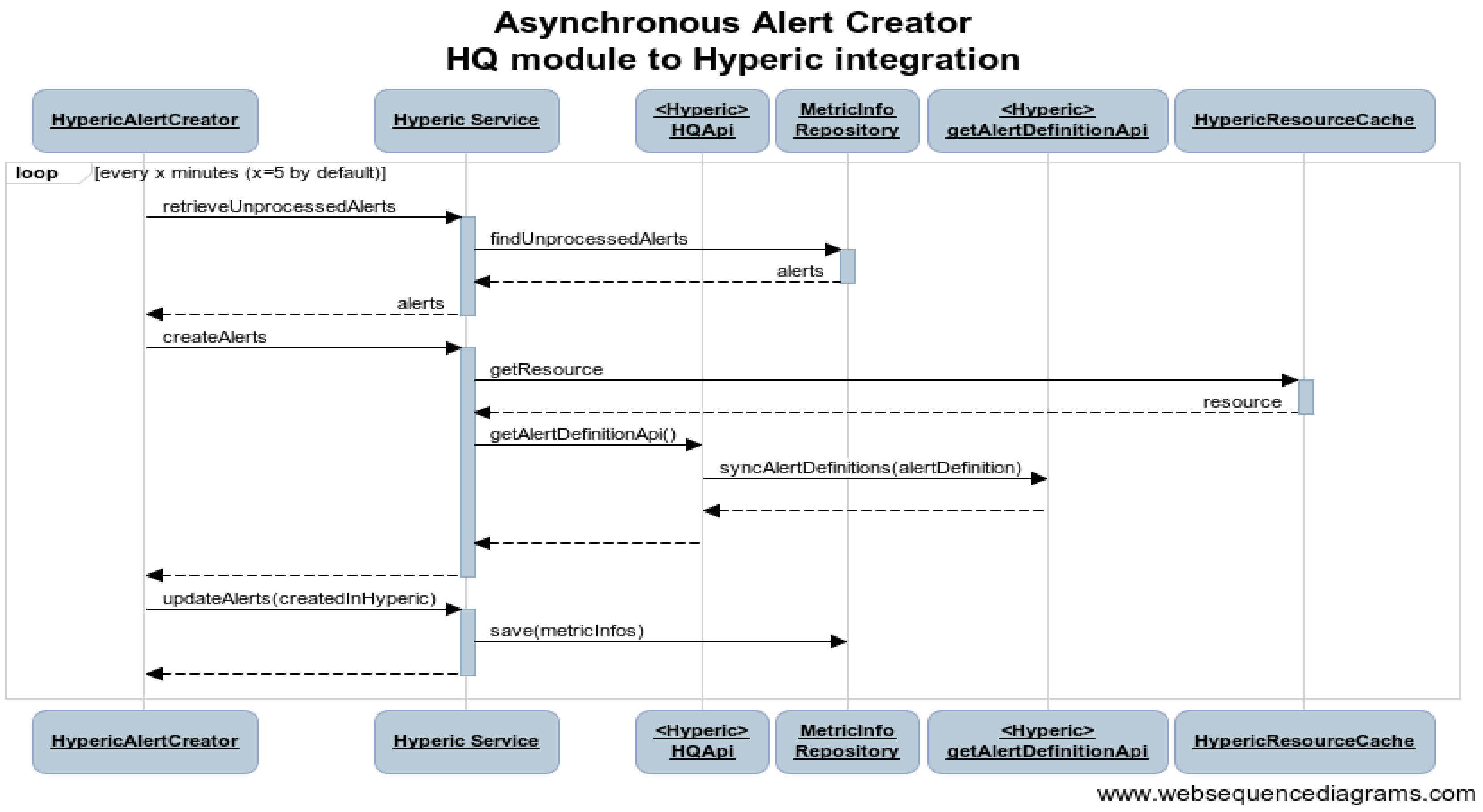
7.9.1 Sequence Diagram – Asynchronous Alert Creator

The following sequence diagram captures the scheduled process that approves resources which were auto discovered by the Hyperic agent/HQ. These resources may include servers, as well as Smarter Balanced components and metrics.


Asynchronous Auto Approver
HQ module to Hyperic integration
HypericAutoApprover
Hyperic Service
<Hyperic> HQApi
<Hyperic> AutodiscoveryApi
HypericResourceCache
loop
[every x minutes (x=5 by default)]
autoApprove
getAutodiscoveryApi
api
getQueue
alt
[loop for each queue item]
approve(queueItem.getId)
loadResource(queueItem.getName)
HypericAutoApprover
Hyperic Service
<Hyperic> HQApi
<Hyperic> AutodiscoveryApi
HypericResourceCache
www.websequencediagrams.com

7.9.2 Sequence Diagram – Asynchronous Alert Creator

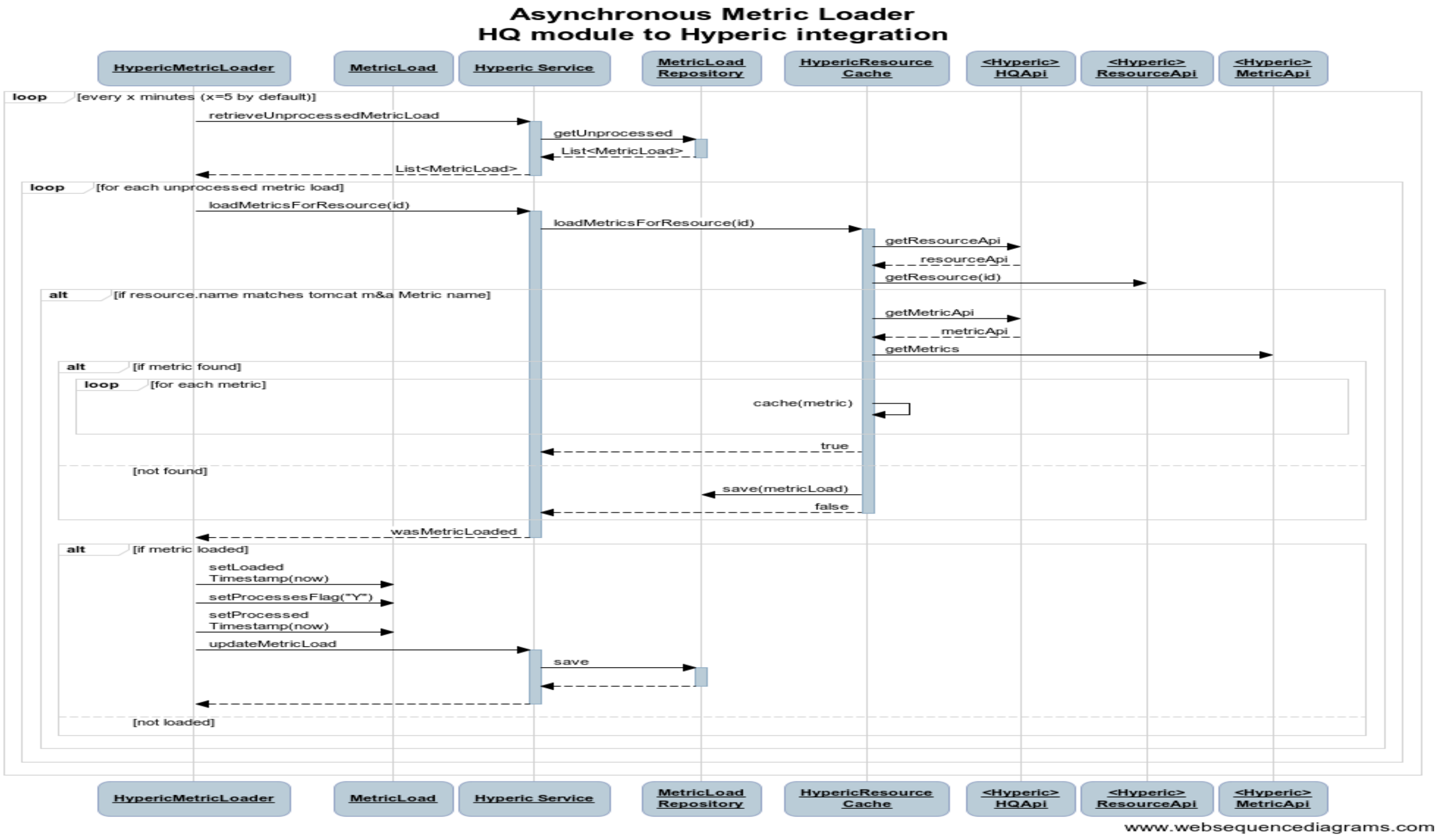
The following sequence diagram captures the schedule process that creates a default alert for Availability Metrics when a component registers itself. The alert is a simple email alert that is customizable in the property files utilized by Monitoring and Alerting. The alert will be generated by Hyperic HQ utilizing some defaulted values that are applicable to Availability ('heartbeat') metrics. The reason this process happens on a schedule is due to the need for a metric to have been created before an alert definition can be created in Hyperic.



Asynchronous Alert Creator
HQ module to Hyperic integration
HypericAlertCreator
Hyperic Service
<Hyperic> HQApi
MetricInfo Repository
<Hyperic> getAlertDefinitionApi
HypericResourceCache
loop [every x minutes (x=5 by default)]
retrieveUnprocessedAlerts
findUnprocessedAlerts
alerts
alerts
createAlerts
getResource
resource
getAlertDefinitionApi()
syncAlertDefinitions(alertDefinition)
updateAlerts(createdInHyperic)
save(metricInfos)
www.websequencediagrams.com

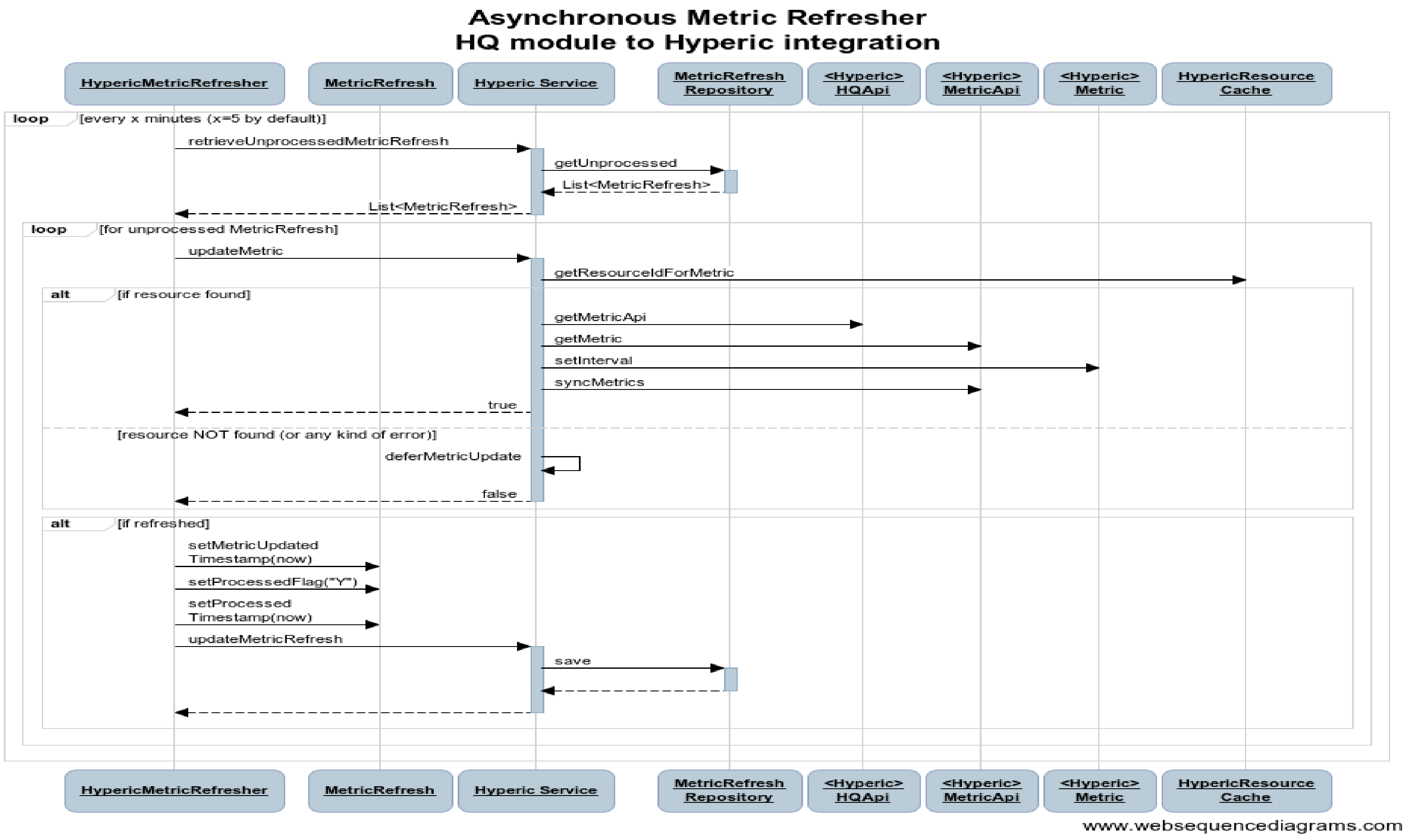
7.9.3 Sequence Diagram – Asynchronous Metric Loader

The following sequence diagram captures the schedule process that prepopulates the resource cache utilized by the other scheduled asynchronous processes. The reason this process happens on a schedule is due to the need for a metric to have been created before a resource can be accessed in Hyperic HQ.



7.9.4 Sequence Diagram –

The following sequence diagram captures the scheduled process that customizes attributes on created metrics. The attribute currently modified is the refresh rate. This allows the component to customize how frequently a metric will be captured and graphed in HQ. For example, the refresh rate varies by metric type, and Availability may have a less frequent refresh rate than a Performance metric.



8. Implementation View

This view describes the organization of static software modules (source code, data files, executables, documentation etc.) in SBAC-11 Monitoring and Alerting component development in terms

- Source code, Packaging and layering
- Configuration Management (build and release strategy etc.)

8.1 Source Code

The source code for Monitoring and Alerting is implemented independent of other components. The component source repository is divided into sub-projects as follow:

- **mna-client**: code to be used by client components for utilizing Monitoring and Alerting services
- **domain**: objects shared by the data workers and controllers. The sb11-shared-code project contains code that can be shared across multiple components. Dependencies on sb11-shared-code and sb11-shared-build are managed with maven.
- **hq**: code used to integrate with the Hyperic HQ open source product
- **jmx**: source code used to register components and metrics with Monitoring and Alerting
- **parent**: the master project which organizes the other subprojects
- **persistence**: data workers and the MongoDb implementation
- **rest**: controllers implementing the RESTful API
- **webapp**: the custom, non-Hyperic user interface

8.2 Build Process

Monitoring and Alerting is using maven to automate the compile/test/package process. A parent pom.xml at the top level defines common dependencies for each component. Child pom.xml files add dependencies unique to each sub-project.

8.3 Third party dependencies

Monitoring and Alerting is relying on the following third party components:

- org.springframework: spring-beans, spring-context, spring-core, spring-web
- org.springframework.data: spring-data-commons-core, spring-data-mongodb
- org.springframework.integration.spring-integration-core
- org.mongodb.mongo-java-driver
- javax.servlet: jstl, servlet-api, jsp.jsp-api
- slf4j & log4j
- cglib
- javax.inject
- org.aspectj.aspectjrt
- org.codehaus.jackson.jackson-mapper-asl
- org.hibernate.hibernate-validator

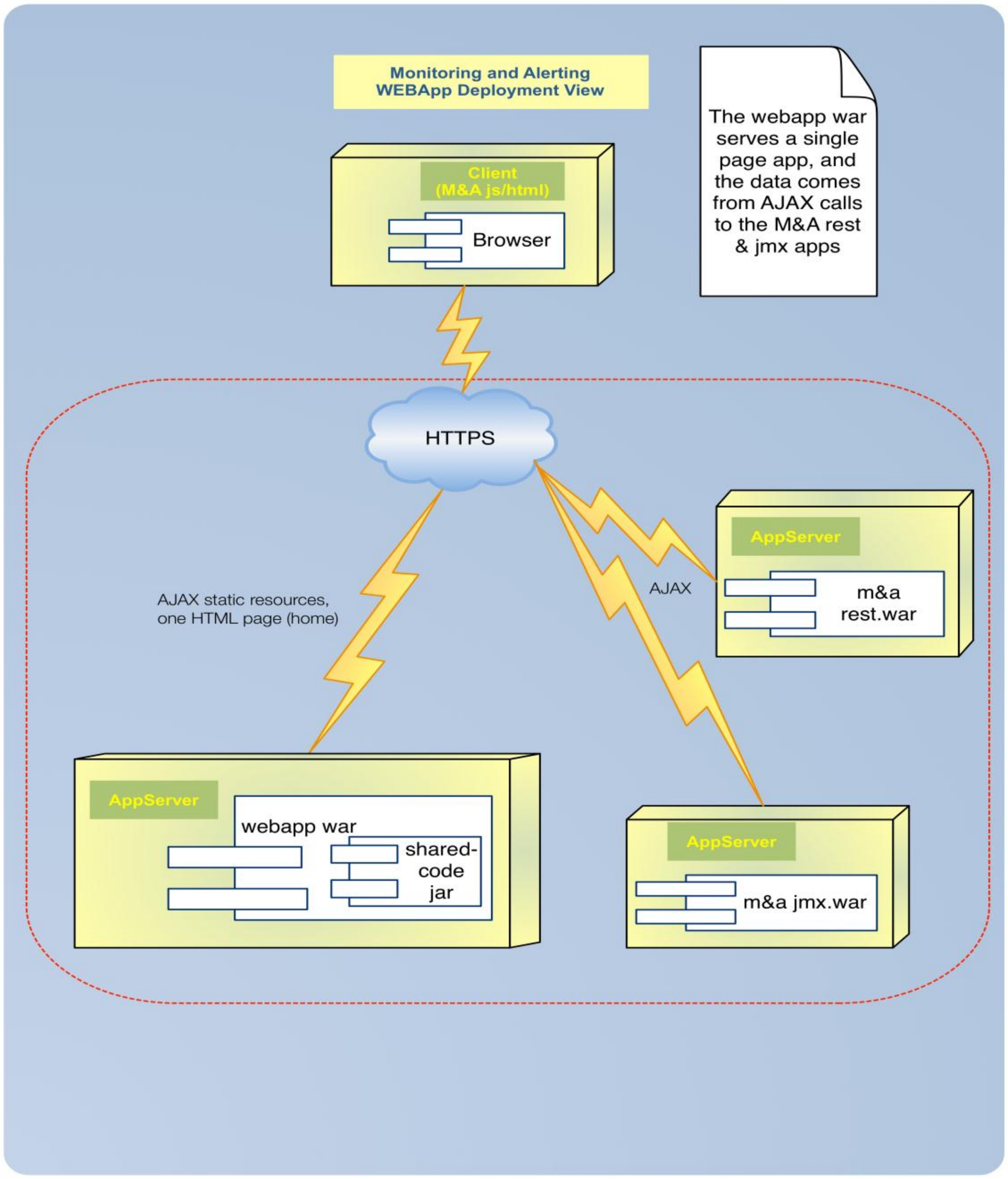
8.4 Test Automation Dependencies

Monitoring and Alerting is leveraging following testing frameworks:

- Junit
- org.springframework spring-test
- org.springframework spring-test-mvc

9. Deployment View

The component is divided into two major pieces: JMX which is used to register metrics and components and REST endpoints which provide Monitoring and Alerting services to other components.



Monitoring and Alerting WEBApp Deployment View
The webapp war serves a single page app, and the data comes from AJAX calls to the M&A rest & jmx apps
Client (M&A js/html)
Browser
HTTPS
AJAX static resources, one HTML page (home)
AJAX
AppServer
m&a rest.war
AppServer
webapp war
shared-code jar
AppServer
m&a jmx.war

Figure 9.1 - Web App Deployment

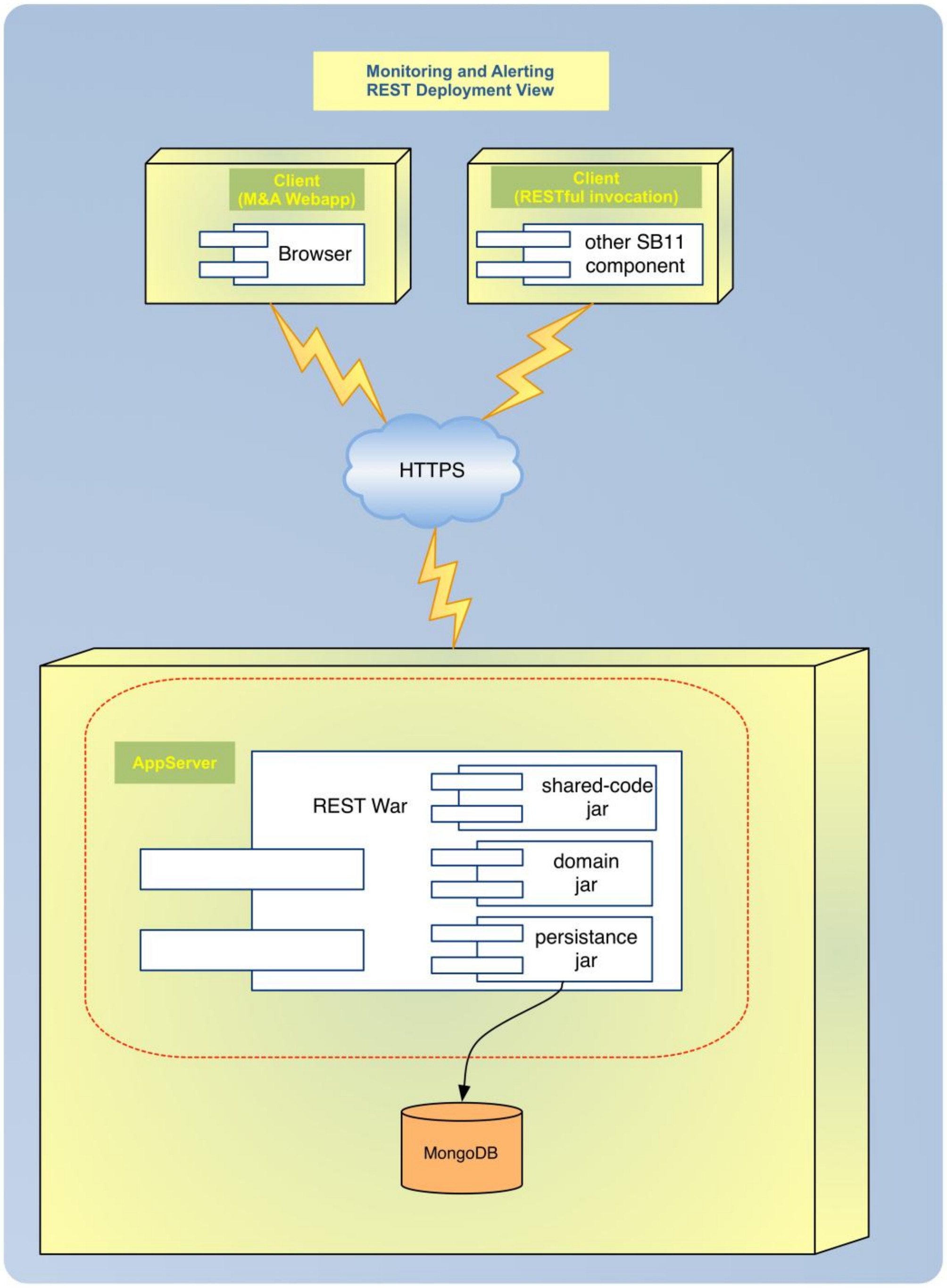


Figure 9.2 - REST

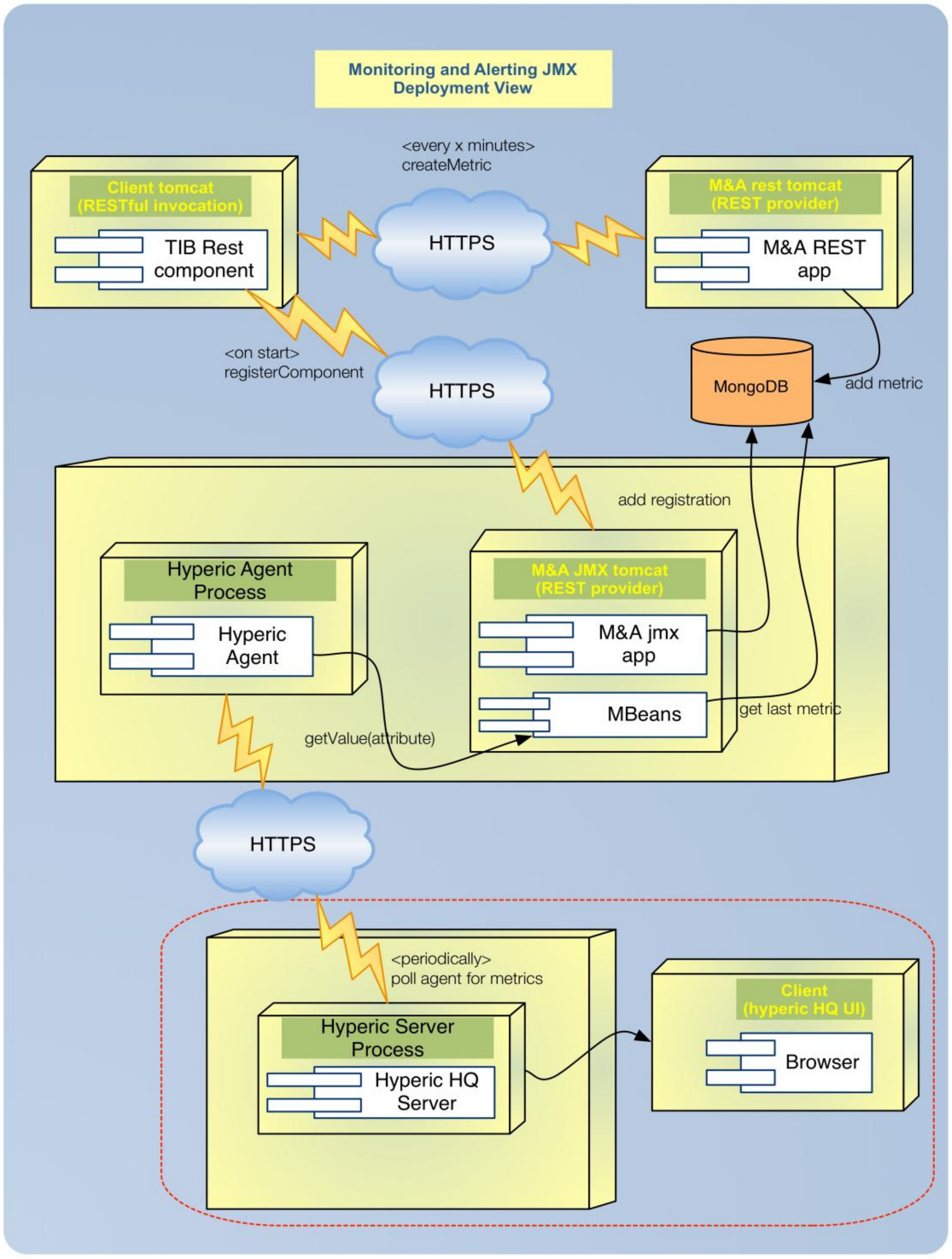

Monitoring and Alerting JMX Deployment View
<every x minutes> createMetric
Client tomcat (RESTful invocation)
TIB Rest component
HTTPS
M&A rest tomcat (REST provider)
M&A REST app
<on start> registerComponent
HTTPS
MongoDB
add metric
add registration
Hyperic Agent Process
Hyperic Agent
M&A JMX tomcat (REST provider)
M&A jmx app
MBeans
get last metric
getValue(attribute)
HTTPS
<periodically> poll agent for metrics
Hyperic Server Process
Hyperic HQ Server
Client (hyperic HQ UI)
Browser

Figure 9.3 - JMX Deployment

10. Security

HTTPS will be provided in the hosted environment. This is not a concern of the application layer as all negotiation and encryption is handled between the network and application server container. Authentication and Authorization is an orthogonal concern within the application. At a servlet container level, Spring Security allows for global and/or more specific security masking of web resources. For example, all web service endpoints can require clients to be authenticated while leaving other assets (such as images or static content) unsecured.

All web service endpoints will be secured using Spring Security Annotations. These annotations allow for role based security. Authentication and user/role mapping will be provided by a shared component. Within the functional components, such as this one, the annotations will be configured to ensure the roles provided by the shared authentication component are in alignment. Spring Security handles insufficient authorization by denying access with a HTTP 403 error returned. Given that the shared component is currently not completed, the exact mechanism of security will remain uncompleted until the Authorization & Authentication can be integrated together.

11. Exception and error handling

11.1 Error handling

Known error conditions such as validation rules will be reported to the client in a consistent manner. The errors will be returned to the user with an HTTP 400: BAD REQUEST and the error message text included in the returned payload. Known errors are enumerated within a component and the dynamic portion of the message parameterized. This allows for static portions of the messages to have multiple translations (a development time concern as defined by the requirements). Spring Validation is being used as the base framework for error checking and validation logic is extended within the application as necessary. Some of the Monitoring and Alerting error handling examples are:

POST: the service will return HTTP status 201 (created) when a log entry is added

POST: the service will return HTTP status 201 (created) when a notificationRule is added, and an id will be returned which can be used to fetch the rule

GET with query params: return a list of results and HTTP 200 when valid query parameters are specified, and HTTP 400 Bad Request if the query parameters are not valid.

11.2 Exception handling

For all exceptions generated from unanticipated conditions, a fault barrier has been put in place to ensure that no details of the originating exception are exposed to the client of the web service. Instead of exposing the exception details (which may contain implementation details), a customizable error message including a unique reference to the logged exception is returned to the user. The unique reference allows the user to communicate with technical support in an unambiguous manner to quickly locate the original root cause for problem resolution. The logged exception will be logged to the local application server environment as well as to the Monitoring and Alerting component for centralized aggregation.

12. Quality

12.1 Size & Performance

Monitoring and Alerting is leveraging NoSQL MongoDB for storage which provides the following size & performance benefits:

- MongoDB eliminates object-relation-mapping work and the so-called “impedance mismatch by leveraging JSON object-style-data.

- Allows dynamic schemas or schemaless operations
- Easy to store and manipulate complex and polymorphic data
- Reduces the amount of work required to scale out the application and increase system speed

Monitoring and Alerting services use asynchronous transmission of messages to eliminate I/O bottlenecks.

12.2 **Scalability**

The design of Monitoring and Alerting's RESTful services allows horizontal scaling. Monitoring and Alerting's persistence layer uses MongoDB which has auto-sharding capability to scale from a single server deployment to large, complex multi-site architectures.

12.3 **Reliability**

Monitoring and Alerting logs messages on each local server as well as writing messages to a centralized data store for persistence. The local log can be used by system administrators if communication with the Monitoring and Alerting component is lost. The MongoDB data store has built-in replication with automated failover which provides enterprise-grade reliability and operational flexibility.

Each system deployment must provide at least two component servers to host the Monitoring and Alerting component to provide failover.

12.4 **Availability**

Each system deployment must provide at least two component servers to host the Monitoring and Alerting component to provide failover. Monitoring and Alerting leverages MongoDB's built-in replication with automated failover to provide high availability for data.

13. API

The component is divided into two major pieces: JMX which is used to register metrics and components, and REST endpoints which provide Monitoring and Alerting services to other components. Because of the length of the API documentation, it is included in a separate document. Once the Monitoring and Alerting component is installed and running, the API can be viewed by entering one of the following links:

- Server-context/jmx/api
- Server-context/rest/api