WillowTree Apps – Sample Application

Rob Ridings

Introduction

The document provides the information for the sample application developed for WillowTree Apps. The document includes Architecture and Design description, API definitions and Stress Test results and analysis.

Use cases

Problem:
A company requires an intranet webservice for managing assets which will be used to provide data to multiple internal applications. This service will be used by multiple datacenters across the US.

Assumptions:

- An ‘asset’ has two fields ‘uri’ and ‘name’ name being a human readable label
- All assets in the organization are identified by a URI, eg: myorg://users/tswift identifies a user record
- Web services must be available at multiple physical sites
- If the network is segmented between sites, service must remain available at each site
- Upon restoration of network connectivity between sites, all changes must be resolved to deliver a consistent view of the data

Goals:

- a way to add and delete assets
- a way to add notes against arbitrary assets
- notes should not be applied to deleted assets

Outputs:

1. An architecture design for the services, considering:
    a. database recommendation and replication strategy
    b. approach to resolving data consistency after network segmentation
2. Design/documentation for the API endpoints.
3. Database/datastore schema design.
4. An implementation of the service which should be able to handle as many concurrent requests as possible (can use inmemory data store)

Architecture and Design description

After a quick review of REST API frameworks and distributed database software, Spark framework and NuoDB were chosen.

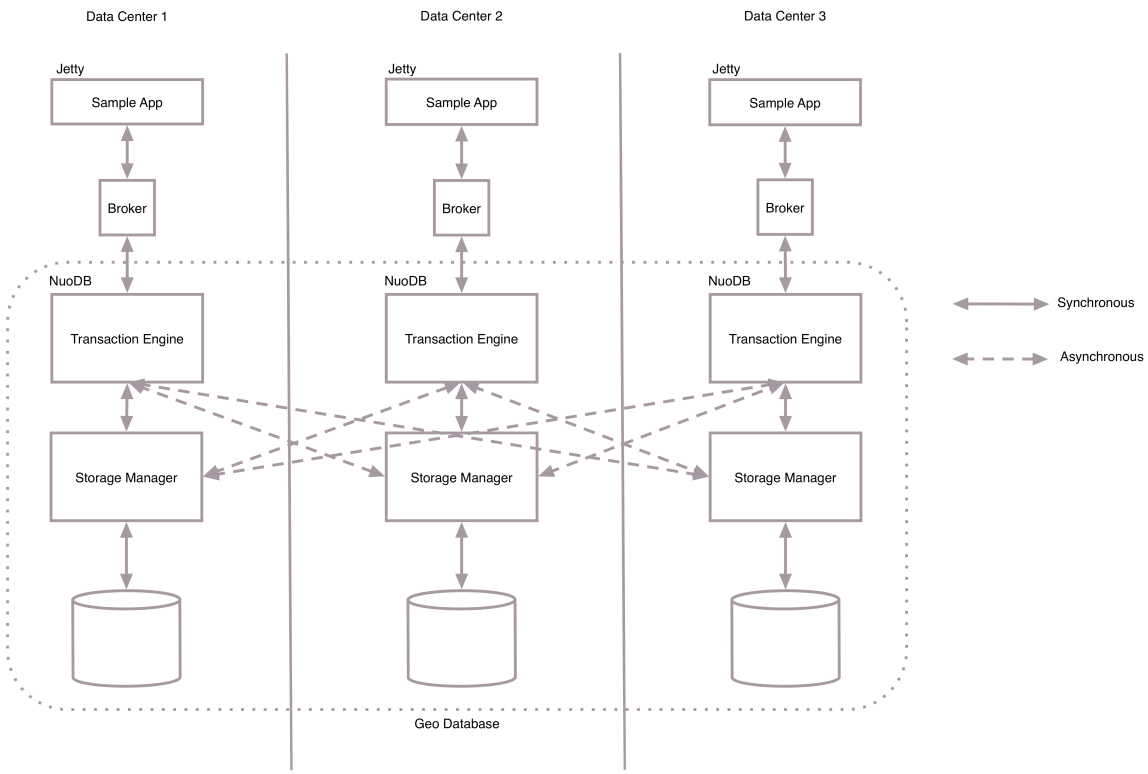
Spark is a java framework for REST APIs. It is simple and lightweight, which allows quick API development. The framework uses an embedded Jetty app server to host the APIs. Spark Framework was inspired by Ruby's framework, Sinatra. It was selected because it is lightweight, easy to create the REST APIs, and simple to run with the embedded.
http://sparkjava.com/

"NuoDB is a distributed database designed with global application deployment challenges in mind. It's a true SQL service: all the properties of ACID transactions, standard SQL language support and relational logic. It's also designed from the start as a distributed system that scales the way a cloud service has to scale providing high availability and resiliency with no single points of failure. Different from traditional shared-disk or sharednothing architectures, NuoDB presents a new kind of peer-topeer, on-demand independence3 that yields high availability, lowlatency and a deployment model that is easy to manage."

NuoDB Technical White paper. http://go.nuodb.com/rs/nuodb/images/Technical-Whitepaper.pdf

NuoDB solves the distribution, replication and segmentation requirement. Additionally, its separation of components allows flexible scalability.

http://www.nuodb.com/



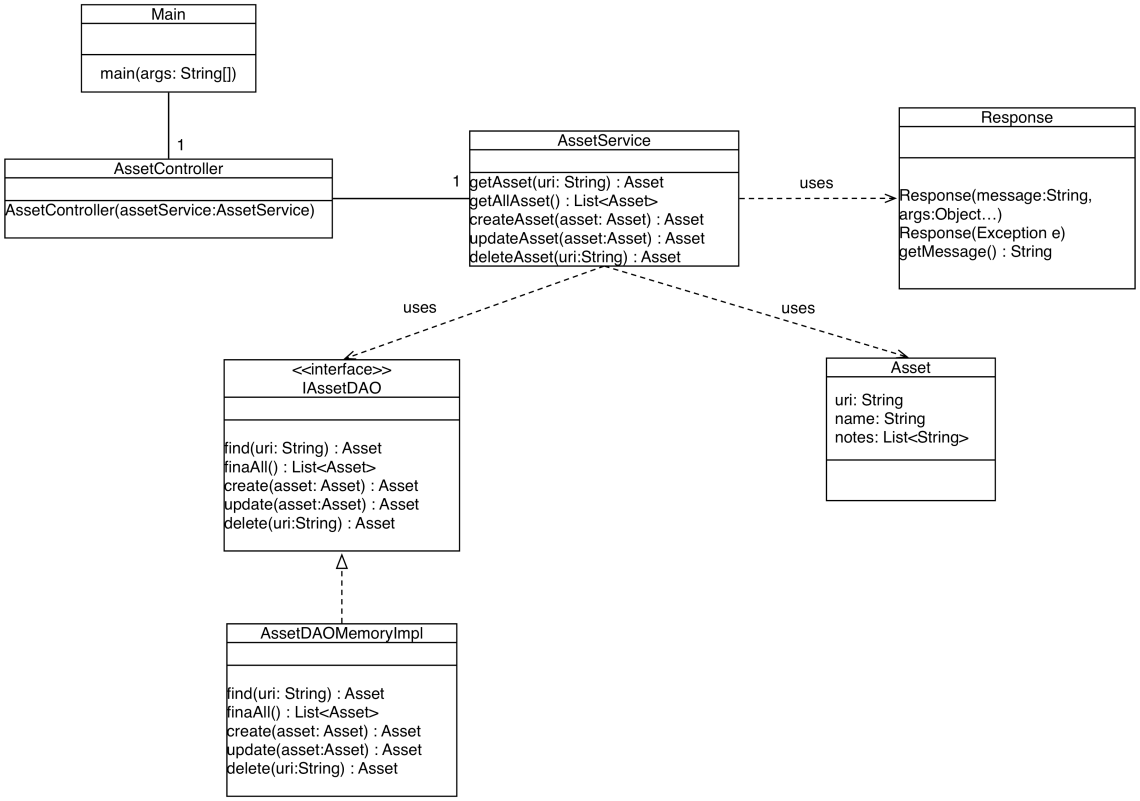
Logical Architecture

The Transaction Engine maintains a cache, which it uses to service each read. If the cache does not have the data, it can retrieve it from another Transaction Engine or a Storage Manager. Information is maintained so the Transaction Engine knows who has the closest copy. If the Transaction Engines are getting overloaded, simply start another one. It will start up and load its cache on-demand.

During a transaction commit, the Transaction Engine notifies other Transaction Engines that have the data cached and each Storage Manager. It communicates synchronously to the local Storage Manager and asynchronously to the others. With the default behavior the Transaction Engine waits for one confirmation to complete the commit. If a Storage Manager was down and did not get the message, at startup it replicates data from another Storage Manager.

The logic architecture makes no assumption about physical hardware or the number of instances of any layer. It is recommended that at each location there should be at least two Transaction Engines. The Transaction Engine and Storage Manager can be installed on the same physical machine, separate machines, or in the cloud.

NuoDB also offers another component, the Broker. The Broker balances requests across multiple Transaction Engines.



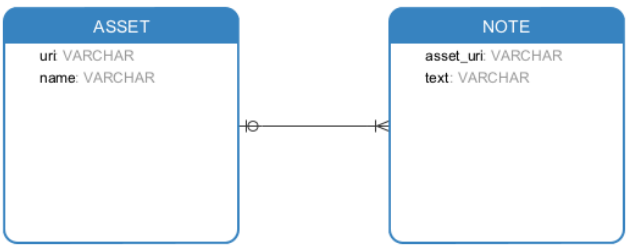
Class Diagram

The sample app follows the MVC architecture pattern minus the view. There is an interface for the Asset DAO to allow additional implementations. The current implementation is in-memory storage using a thread-safe Hashtable.

The REST API endpoints are defined in the constructor of the AssetController. There are routes defined for GET, POST, PUT, and DELETE.

The POST and PUT method passed data in body in JSON format. All APIs return data in JSON format.

The application was tested using JUnit integration and unit tests. The GET, POST, and PUT methods were tested with Apache Bench. Apache Bench does not support DELETE.



Entity Relationship Diagram

The data model is quite simple, two tables ASSET and NOTE. There is a foreign key relationship between asset_uri and uri. The uri column on Asset is a unique index.

Web Services APIs

/asset{uri}

parameter	value	description
uri	*string*	the uri of the asset

GET
Returns a specific asset identified by the uri.

http://localhost:4567/asset/myorg://users/tswift

available response representations:

- 200

  Example

  ```
  {
          "uri":"myorg://users/twsift",
          "name":"Taylor Swift"
  }
  ```

- 400 {"message":"No asset with uri myorg://users/tswift found"}

Returns a full JSON representation of an asset

DELETE
Deletes a specific asset identified by the uri. Returns the deleted asset.

http://localhost:4567/asset/myorg://users/tswift

available response representations:

- 200

  Example

  ```
  {
          "uri":"myorg://users/twsift",
          "name":"Taylor Swift"
  }
  ```

- 400 {"message":"No asset with uri myorg://users/tswift found"}

Returns a full JSON representation of an asset

/asset

GET
Returns all assets

http://localhost:4567/asset

available response representations:

- 200

  Example

```
[
   {
                "uri":"myorg://users/test2",
                "name":"Test2"
   },
   {
                "uri":"myorg://users/test1",
                "name":"Test1","notes":["Note1","Note2"]
   }
]
```

- 400 {"message":"":"<exception message>"}

Returns a full JSON representation of a list of assets

POST
Creates an asset. Returns the new asset.

acceptable request representation:

- application/json

  Example

```
{
      "uri" : "myorg://users/test",
      "name": "Test"
}
```

available response representations:

- 200

  Example

```
{
          "uri":"myorg://users/test",
          "name":"Test "
}
```

- 400 {"message":"<exception message"}

Returns a full JSON representation of an asset

PUT
Updates an asset. Returns the updated asset.

acceptable request representation:

- application/json

  Example

  ```
  {
        "uri" : "myorg://users/test",
        "name": "Test",
        "notes": ["Note1","Note2"]
  }
  ```

available response representations:

- 200

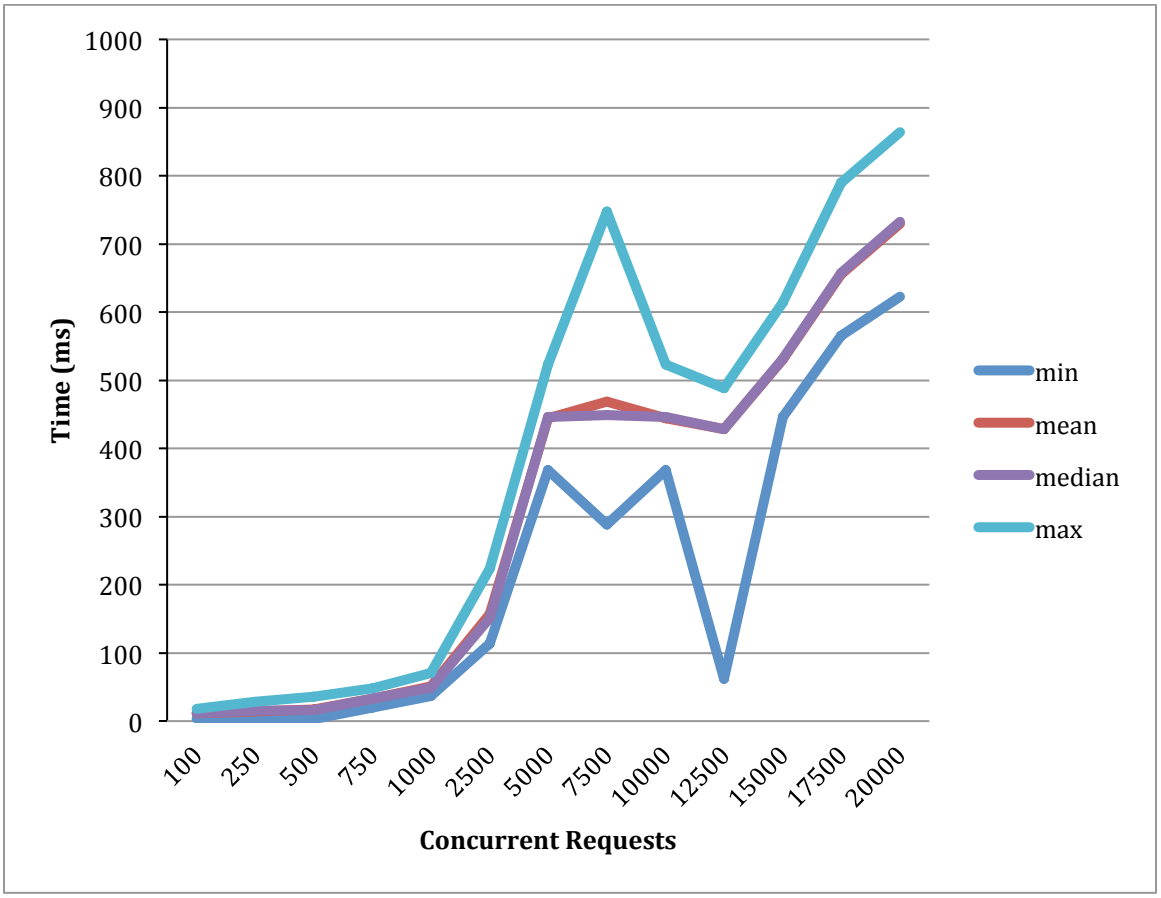
  Example

  ```
  {
              "uri":"myorg://users/test",
              "name":"Test",
              "notes":["Note1","Note2"]
  }
  ```

- 400 {"message":"<exception message"}

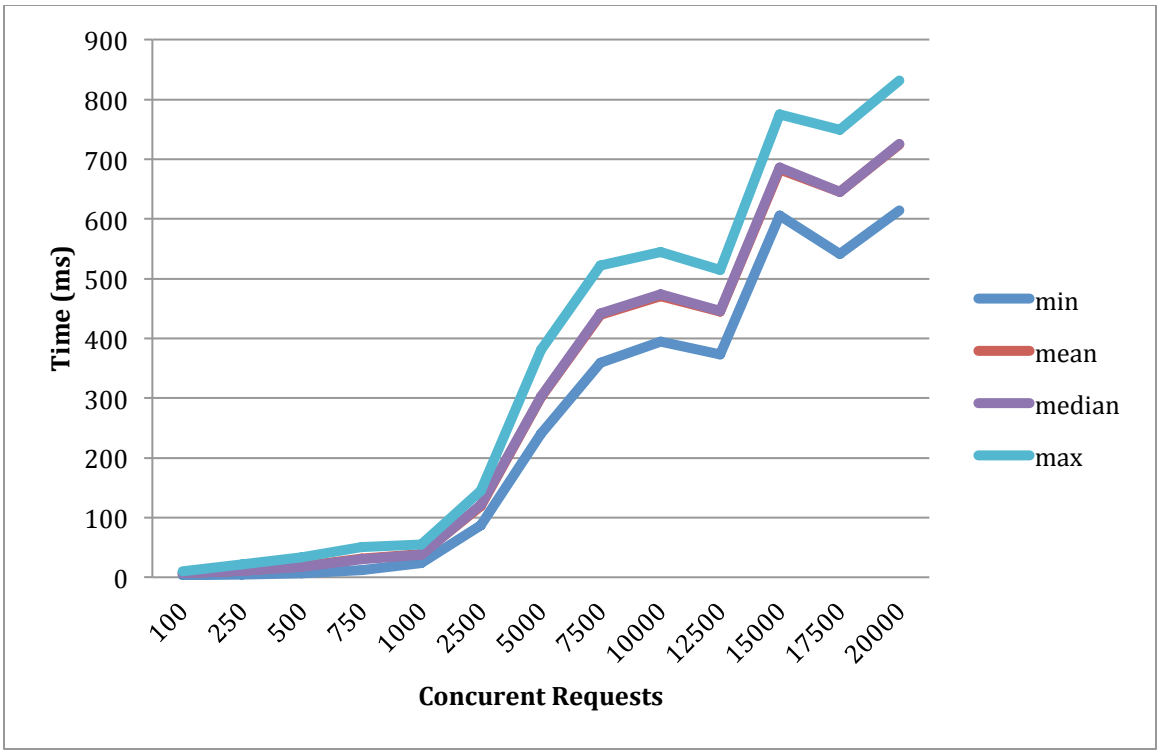
Returns a full JSON representation of an asset

Stress Testing



Sample App Stress test – read

The read stress test was run using Apache Benchmark. The read test ran concurrent requests from 100 to 20,000, each request made 100 GET API calls. The concurrency values are in the chart above. The mean values are below 100 milliseconds for concurrencies below 2500. Above 2500 concurrent requests, the timings become a bit erratic, however, all values remained sub-second.



Sample App Stress test - writes

The write test was also run with Apache Bench. Concurrent requests were run from 100 to 20000. Each request made 100 POST API calls. The graph shows good numbers for concurrent requests below 2500 with call times less than 100 milliseconds. The write test has about the same turning point as the read test.

Given that storage implementation is an in-memory Hashtable, I would suspect the slow down the Spark framework and route mapping, the GSON mapping, or the testing environment.

When trying to run the stress test with the PUT method, the GSON mapper was throwing ConcurrentModificationException when parsing the Notes list. Due to time constraints the stress test was switched the POST method. This outstanding issue requires either the GSON mapper debugging or switching to a different JSON parser.

Conclusion

Thank you the providing this test project. I enjoyed the challenge and learning/researching new technologies. The project gave me a specific problem to research. In my research I came across NuoDB, which I want to take a deeper dive and Spark framework, which lived up to its claim.