CodingInfinity

Benchmark Service Architectural Requirements Documentation

Git:
`https://github.com/CodingInfinity/Benchmark-Service-Documentation`
GitHub Organization: `https://github.com/CodingInfinity`

The Client:
Ms Vreda Pieterse
Department of Computer Science
University Of Pretoria

The Team:
Andrew Broekman *11089777*
Brenton Watt *14032644*
Fabio Loreggian *14040426*
Reinhardt Cromhout *14009936*

September 2016

Contents

List of Figures

Chapter 1

Software Architecture Overview

Figure 1.1 shows a high-level overview of the software architecture at first granularity with the benchmarking system being based on a message bus architecture. The figure further shows the core architectural components of the system.

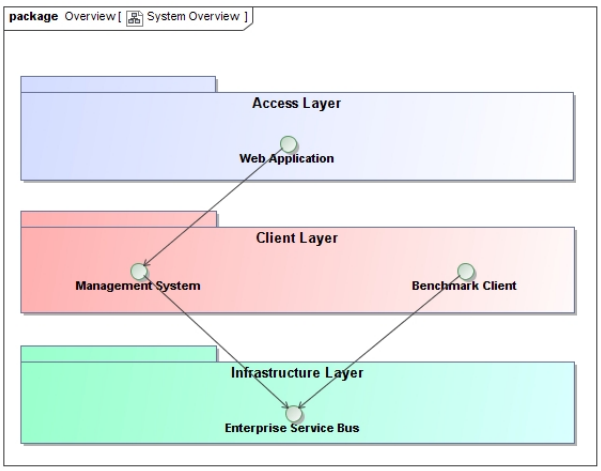

Figure 1.1: A high-level overview of the software architecture for the Benchmarking Service at first granularity level

1.1 Management System

Figure 1.2 shows a high-level overview of the management service software architecture at second granularity level with the service being based on a layered architecture. The figure further shows the core architectural components of the subsystem.

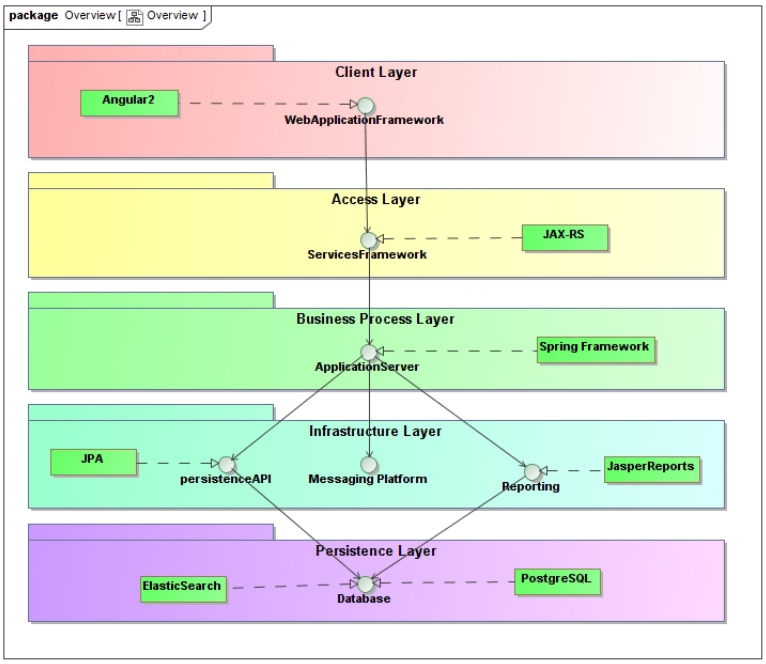

Figure 1.2: A high-level overview of the software architecture for the Management Service at second granularity

1.2 Benchmark Monitor

Figure 1.3 shows a high-level overview of the benchmark service software architecture at second granularity level with the service being based on a monolithic architecture. The figure further shows the core architectural components of the subsystem.

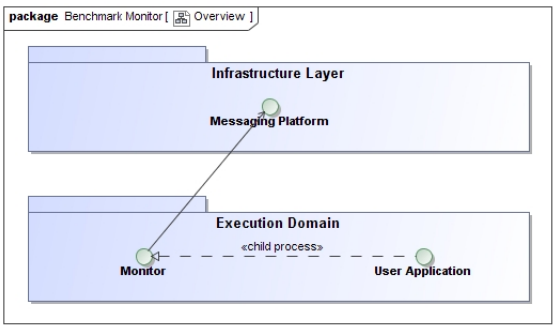

Figure 1.3: A high-level overview of the software architecture for the Management Service at second granularity

Chapter 2

Overall Software Architecture

This section specifies the software architecture requirements and the software architecture design for the first level of granularity - the system as a whole. The output will be the high level software architecture components, the infrastructure between them and the tactics used at the first level of granularity to realize the quality requirements for the system. Subsequent sections will focus on the software architecture requirements and design for these high-level architectural components. Note that many of the architectural requirements (particularly the quality requirements) will be propagated down to lower level architectural components.

2.1 Architecture Requirements

This section discusses the software architecture requirements around the required software infrastructure within which the application functionality is to be developed. The purpose of this infrastructure is to address the non-functional requirements. In particular the architecture requirements specify.

- the architectural responsibilities which need to be addressed
- the access and integration requirements for the system
- the quality requirements
- the architecture constraints specified by the client

2.1.1 Access and Integration Requirements

Access Channels

At the first level of granularity we require one system access channel to facilitate communication between the management and benchmark client service.

The benchmark client application or so-called *monitor* application will upon completion of the benchmarking jobs, need to report the results back to the management

system, which will need to process and persistent the results. To accomplish this communication between these disparate systems, a messaging platform will be used to facilitate communication.

2.1.2 Quality Requirements

This section will specify the quality requirements at the highest level of granularity. These requirements will propagate throughout the entire system into every other component. Quality requirements that are specific to certain parts of the system will be discussed under the respective sections.

Flexibility

The system should utilize dependency injection, open standard protocols and open source library implementation's to allow the user to switch out any realization of a technology with another.The system should use best software engineering practices as far as possible, to allow the system to be easily expanded upon by any member of the greater public community as this project is to be released as an open source project.

Maintainability

Among the most important quality requirements for the system is maintainability. It should be easy to maintain the system in the future. To this end

- Future developers should be able to easily understand the system,
- The technologies chosen for the system can be reasonably expected to be available for a long time
- Developers should be able to easily and relatively quickly
  - Change aspects of the functionality the system provides, and
  - Add new functionality to the system.
- Configure a new monitor node to be able to fetch tasks and post its results to the system.

Scalability

As this system is envisaged to be used in academia in very localized context, a need for scalability doesn't exist.

Reliability

The system must be reliable in terms of messages exchanged on the message platform as we would want to avoid a situation where users results get lost in the system and require users to possible run an expensive experiment a second time.

Further more the results, experiment specifications and user uploaded artifacts should be reliably sorted and retrieved, as these specifications all need to be declared in academic research.

Security

As this system is a prototype, security is not a major concern. However security should be kept in mind, as this is a software engineering best practice. However full isolation on the monitor nodes and network isolation is outside of the scope for this prototype.

A security requirement specified by the client is that of least privilege on the system as a whole. In this regard, the monitor nodes are only allowed to communicate to the message platform, the management back end is only allowed to communicate with the message platform and receive connections from the web interface.

Integrability

The chosen message infrastructure and lower level components namely the management system and monitor application should be integrable over a common interface. Note however, as we are aiming for a loosely-coupled system, the management system will rather communicate with the monitor application than integrate with it.

Deployability

The system must be buildable from source and build scripts only. The system must be deployable

- on Linux servers,
- in environments using different message brokers for the enterprise message platform.

The system should ultimately be packaged as a series of Docker images which are deployable as Docker containers installed on virtual or physical Linux servers.

2.1.3 Architectural Responsibilities

The architectural responsibilities of the messaging platform are shown in Figure 2.1

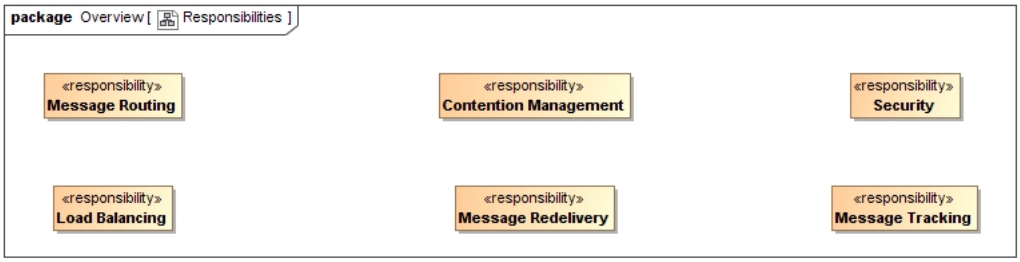

Figure 2.1: The architectural responsibilities of the Messaging Platform

2.1.4 Architecture Constraints

The chosen architecture was determined to best fulfill the non-functional requirements for the system as well as the requirements set forth by the client.

- The architecture must be deployable on Linux servers
- All libraries, frameworks, programming languages and any other material of sort utilized within this project must be open-source.
- All libraries, frameworks, programming languages and any other material of sort used must be supported by open standards, have an active and vibrant support community and should have an active release cycle.

2.1.5 Architecture Design

2.1.5.1 Frameworks and Technologies

Messaging Broker As the benchmark service is based on an messaging architecture, the choice of which message platform to use is a critical decision. It was determined that the Apache ActiveMQ will be used as the reference implementation.

Apache ActiveMQ has a vibrant support community, supports various programming languages, and has a very active release cycle, with the latest stable release being version 5.13.1 released in February 2016, at this time of this writing.

Features of Apache ActiveMQ includes:

- Spring Framework support
- Support J2EE servers
- Supports pluggable transport protocols
- Designed for high performance clustering and client-server communication
- Full support for Enterprise Integration Patterns

Other implementations considered:

- Apache Apollo
- Apache Qpid
- Pivotal RabbitMQ

2.1.5.2 Concepts and Constraints for Application Components

ActiveMQ provides an implementation of a messaging architecture used to provide integration between disparate business services.

Messaging architectures are commonly used to decouple business services from one another while providing reliable communication between these disparate systems.

Chapter 3

Management System Software Architecture

This section specifies the software architecture requirements and the software architecture design at the second level of granularity - the management service of the benchmarking system. The management service will be responsible for all administration of user management, repository management, experiment management and reporting. The management service will communicate with the client monitor applications through the messaging platform. The reader is referred to figure 1.2 for an overview of the management platform architecture and requirements.

3.1 Overall Software Architecture

3.1.1 Architecture Requirements

This section discusses the software architecture requirements around the back end management system infrastructure. The back end management system will be responsible for the delegation of jobs to cluster nodes, persisting of results, user management and allowing users to derive value from reporting. In particular the architecture requirements at the second level of granularity is specified.

3.1.1.1 Access and Integration Requirements

Access Channels The access channels can be divided into two broad categories based on the user type

- Human Access Channel
- System Access Channel

Human Access Channel The system will be providing a REST based access channel to be used by human users via the HTML 5/JavaScript Single Page Application Web Interface.

System Access Channel The system will expose a further access channel to be used by the so-called "Benchmark Client" or "monitor" application. This access channel will be utilizing a message bus architecture to deliver messages between the client and the management back end system.

This access channel will however not be accessible to end users.

Integration Channels The various integration channels of the benchmarking system

- Integration with a persistence provider.
- Integration with the human access channels, namely the web interface.
- Integration with the message platform architecture.

The integration with the persistence provider is required as we need to persist the measurements obtained with the benchmarking client. Further more as per the client's request, test data must also be persisted as it is envision that a repository of test data will be build.

In order to make the results from the benchmarking tests useful, users will need some way to interact and manipulate the data in order to be able to derive value from this data. To enable users to interact with the data, a web interface will be provided.

The final integration required by the management system is that of integration with the message platform. In order to better decouple the monitor and management systems, a messaging architecture was introduced. The management system will process the results from a queue structure managed by the messaging system.

Once a program is uploaded via the a human access channel, it will be deployed to an autonomous monitor node upon which the benchmarking will commence. The reason for utilizing a messaging architecture is assist in making nodes autonomous in order to meet the security quality requirements as provided in section 2.1.2.

3.1.1.2 Quality Requirements

The quality requirement are the requirements around the quality attributes of the systems and the services it provides. This includes requirements like maintainability, flexibility, extensibility, performance, scalability, security, auditability, usability and testability requirements.

Authentication The system needs to support a simple registration and authentication framework which will determine what each user can do based on their authority level.

Flexibility Persistence architectures and reporting infrastructures are rapidly evolving as can be seen from the rapid growth of NoSQL databases, semantic knowledge repositories and big data stores. In this context it is important that the application functionality is not locked into any specific persistence technology and that one is able to easily modify the persistence provider and reporting framework.

Testability All services offered by the system must be testable through

1. automated unit tests testing components in isolation using mock objects, and
2. automated integration tests where components are integrated within the actual environment.

In either case, these functional tests should verify that

- the service is provided if all pre-conditions are met (i.e. that no exception is raised)
- the correct exception is thrown when the corresponding pre-condition is violated.
- that all post-conditions hold true once the service has been provided.

In addition to functional testing, the quality requirements should also be tested.

3.1.2 Architecture Design

This section specifies the software architecture design for the second level of granularity. It includes the allocation of architectural responsibilities to architectural components, any tactics which should be used at the current level of granularity to address quality requirements,

3.1.2.1 Architectural Responsibilities and Components

Figure 3.1 shows the allocation of architectural responsibilities to architectural components.

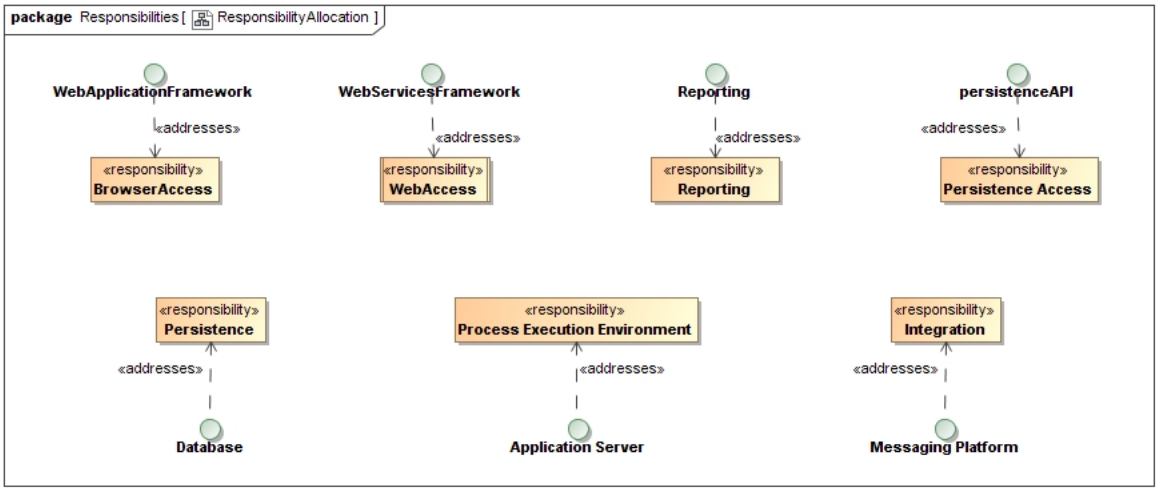

Figure 3.1: The abstract architectural components to which the architectural responsibilities are assigned.

3.2 Database

The database is a persistence provider providing long term, organized storage of the data which will be generated through the application. Various types of database exist, which are mainly categorized based on the approach with which the provider in questions represents and stores the data internally. Some of the types of databases which where considered for this project include

- Relational Databases e.g. MySQL, PostgreSQL
- Document-oriented Databases e.g. MongoDB, Couchbase
- Graph-based Databases Neo4j, OrientDB

In choosing a persistence provider, it is important to consider the data as well as the relationship between the data to ensure the correct type of provider is selected.

The data to be generated by the benchmarking service will have a very hierarchical structure between data elements, making the data well suited for a SQL-based persistence provider. For large BLOB objects, a document-based persistence provider will be used.

3.2.1 Architecture Requirements

3.2.1.1 Quality Requirements

Scalability The database selected must allow for horizontal scaling of the infrastructure, which means that the database should be able to scale across different hosts to

form a combined and intelligent cluster. Optional support of distribution the database across providers will be beneficial for future expansion.

Performance As the database will receive a higher ratio of reading to writing operations per sec, this must be kept in mind when selecting the required database. The database should also be able to deliver on strict Service Level Agreements (SLA), especially on availability.

Reliability As the database will be used to build a repository of knowledge, in the form of historical benchmark results and test data sets, it is of the utmost importance that the selected database store is able to preserve data reliably even in the event of a possible disaster. The one again highlights the need for a database store which is able to function across data centers.

Integrability The chosen database should be supported by the chosen persistence API realization in such a way, that from the business logic, there are no database specific code. The chosen persistence API was chosen for this exact reason as to support the swapping out of the underlying database store without the need to change any higher level code. As not all databases are supported by the persistence API, this places an inherit technological constraint on the database stores which can be utilized in the project.

3.2.2 Architecture Design

3.2.2.1 Architectural Responsibilities, Components and Realization

The architectural responsibilities, components and concrete realization of these responsibilities are shown in Figure 4.2

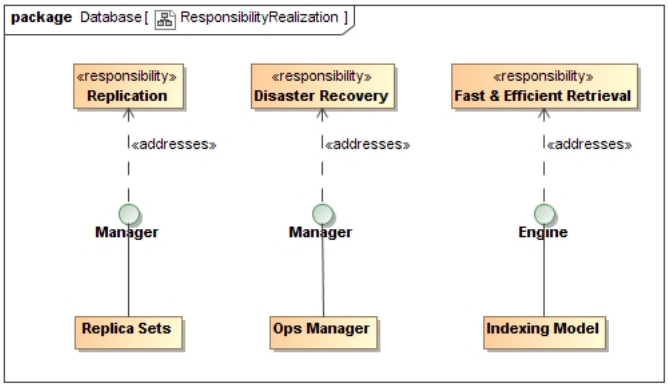

Figure 3.2: The components within both PostgreSQL and ElasticSearch addressing the architectural responsibilities of the database

3.2.2.2 Tactics

As this system is too be used in academic context's with limited ICT resources. we require some tactics which should be used by the database to address the quality requirements:

- *Scaling out resources* to allow for flexible and cheap improvement in performance and scalability.
- *Efficient Persistence* to allow for the quick and efficient retrieval of data from the provider aiding in performance improvement.
- *Efficient Storage Usage* to minimize the required infrastructure needed by the service.
- *Spread Load* in order to fulfill the required scalability and performance requirements as set out above.
- *Removing single points of failure* to ensure data is protected against disasters as the client will be building a knowledge repository.

3.2.2.3 Frameworks and Technologies

The databases chosen for the project is a PostgreSQL and ElasticSearch as they fulfill the required quality and technological requirements as set out in this section.

3.3 Persistence API

The persistence API provides abstracted access to a persistence provider while remaining decoupled from the underlying technology, including but not limited to database, SQL version and transaction management, whilst employing a range of software engineering tactics to concretely address required quality requirements required for the persistence domain.

3.3.1 Architecture Requirements

The architectural requirements for the persistence API include the refined quality requirements and architectural requirements listed below. The architectural constraints for this lower level components are the same as for the system as whole, as referred to in section 2.1.4 with further extensions as specified in section **??**.

3.3.1.1 Quality Requirements

Flexibility The provided persistence API should be able to adapt to the rapidly evolving persistence architecture domain, especially in terms of the different methodologies of storing data such as relational and NoSQL data stores. It is further import that the persistence layer is not locked to any specific persistence technology.

Maintainability The used persistence API should be in a mature stage of the software development life cycle as to guard against a rapidly evolving changing API. The chosen persistence API should be an open standard with multiple realization as to guard against realization technologies be abandoned. This will allow in future an easier switch to another persistence API implementation if required for the long term maintenance and use of the project.

Scalability The chosen persistence API should be able to allow for future scaling of the infrastructure either horizontally or vertically with a preference for horizontal scaling.

Performance The persistence API should allow for the use of certain architectural tactics to increase performance. Specifically the following tactics should be supported to some extent

- Object Caching
- Connection Pooling
- Thread Provisioning
- Scheduling

3.3.2 Architecture Design

3.3.3 Architectural Responsibilities, Components and Realization

The architectural components of the persistence API are shown in Figure 3.3

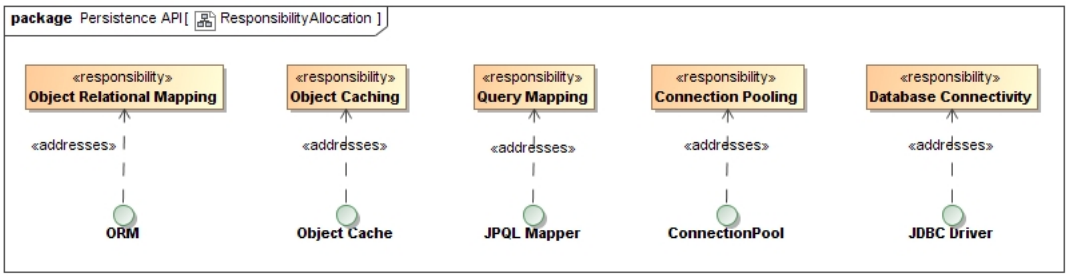

Figure 3.3: The abstract components to which the architectural responsibilities are assigned.

3.3.3.1 Tactics

The persistence API implement the following tactics:

- *Object Relational Mapping* to reduce code bulk, improve maintainability and allow for decoupling from the persistence provider.
- *Query Mapping* from queries across a graph of Java objects onto the database queries used in the selected database technology and provider.
- *Object caching* to improve scalability and performance.

3.3.3.2 Frameworks and Technologies

JPA 2.1 will be used as the persistence API as it is a widely supported public standard with multiple implemenentations, with most implmenetations supporting both relational and NoSQL persistence stores. The chosen concrete implementation used will be Hibernate.

- *Object Relational Mapping* including the mapping of relationships between objects via a provided ORM implementation such as Hibernate or EclipseLink.
- *Query Mapping* from object-oriented queries across the domain object graph to queries for a specific database provider.
- *Object caching* within the persistence context with in-memory or NoSQL database based caching allow for the fulfillment of the performance, flexibility and scalability quality requirements.
- *Connection Pooling* is provided through a JCA connector based implementation of a JDBC driver.

Queries will be specified as Spring Data JPA queries, thereby reducing boilerplate code which ease future maintenance and development, as the queries are not specific to any underlying query language e.g. SQL or any underlying persistence technology such as relational or NoSQL providers.

3.3.3.2.1 Concepts and Constraints for Application Components The application concepts within the persistence domain include

- *Domain objects* which host long-living state objects, and is realized in the Java architecture as Plain Old Java Objects (POJO's) which doesn't contain any business logic.
- *Queries across object graph of domain objects* through which the required information of state in the domain objects is retrieved, modified and removed.

3.4 Web Services Framework

The web services framework is used to expose business services in a technology-neutral way over some network, which in most cases is the public Internet. Wrapping business services in a technology-neutral layer allows one to decouple the front-end technologies, specifically the user interface technologies from the back-end technologies, while simultaneously allowing for the decoupling of back-end services from one another, in effect communication between disparate applications. This decoupling provides one with the ability to vary either the front-end technologies and back-end technologies independently from one another. Furthermore this allows one to write back-end services in the most appropriate technology stack and then have seamless communication between these individual components.

3.4.1 Architecture Requirements

3.4.1.1 Access and Integration Requirements

The web service framework serves as the bridge between the client and back end systems, allowing one to communicate with the other. For this reason it is important the access channel to be used between the client and back end systems should utilize common and standard compliant protocols to ensure the greatest amount of integration can be achieved to provide users with maximum value.

3.4.1.2 Quality Requirements

Maintainability The web services framework is concerned with wrapping business logic, thereby allowing one to categorize this as so called "plumbing code" which should be as far as possible be removed from the actual code. This code is normally applied by the use of annotations in the Java context or by weaving the code into existing business code using aspects.

Using the above mentioned approach allows one more easily to maintain the code base.

Integrability The framework should enable one to expose business services in a technology neutral way to allow for easy integration with other independent business services and systems. The chosen technology neutral format should be supported by the business services and system that require integration.

3.4.2 Architecture Design

3.4.2.1 Architectural Responsibilities, Components and Realization

The concrete components addressing the required responsibilities are shown in Figure 3.4.

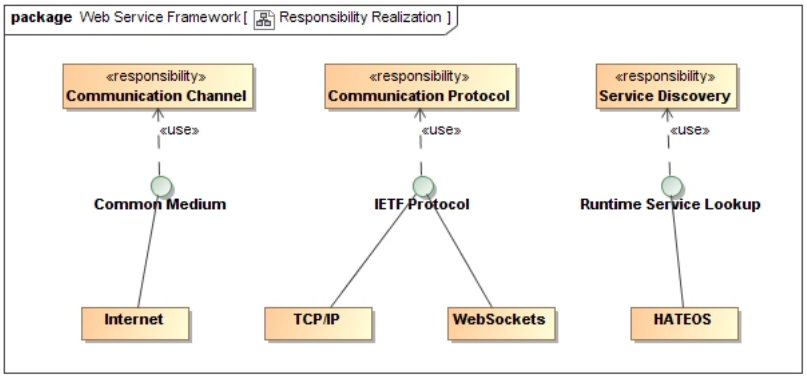

Figure 3.4: The components within Jersey addressing the architectural responsibilities of the Web Services Framework

3.4.2.2 Tactics

The Web Services Framework implement the following tactics:

- *Support Communication Channels* The web services framework should support the communication channel to be used between the client and server as well as between between business services. With regards to this project the communication channels will consists of a standards based network connection between all parties, with the communication channel not necessarily being uniform between all parties. The most likely communication channel between the client and server will be the Internet network with a internet network between business services.

- *Support Standard Communication Protocols* The web service framework should support standards based communication protocols as this will ensure the highest change of ensuring full integrability between client and other business services. All parties should at least support one of the standards the selected web services framework supports, thereby ensuring all parties will have successful communication.

3.4.2.3 Frameworks and Technologies

We decide upon a light, text-based communication standard, namely REST, which will be used to bridge the communication between the client and back end systems. Various frameworks exists, however the Java programming language exposes a vendor neutral REST wrapping API referred to as JAX-RS. The advantage of using a vendor neutral API is that there are various realizations of these API.

The API assist in reducing boilerplate code by weaving in code with the use of annotations, which assists in achieving required quality requirements namely maintainability and flexibility.

For the project in question, we will be using the Jersey reference implementation from Sun of the JAX-RS API standard.

3.5 Web Application Framework

This section specifies the software architecture requirements and the software architecture design for web application framework.

3.5.1 Architecture Requirements

The web application framework provides the software architecture for the software providing browser based access to human users.

3.5.1.1 Access and Integration Requirements

Access Channel Requirements The web application framework will address the human access channel requirement referred to in section 3.1.1.1. As this implementation of the system is a prototype, we will only be supporting the latest Google Chrome web browser. The web application is however HTML5 compliant.

Integration Channel Requirements The web application must integrate with the management system in order to allow users to derive value. However, the web application will integrate with the business layer through the web services layer, allowing one to decouple the business and presentation layers from one another.

3.5.1.2 Quality Requirements

Usability It is important that users using the system feel as though the system delivers what they require, when they require it.

Maintainability The system as a whole should be easily expandable, for this reason a modern JavaScript framework such as Angular2 was chosen to assist developers to maintain the system much easier, allowing the code base to mature with time without losing strict control and structure.

3.5.2 Architecture Design

This section specifies the software architecture design for a third level granularity component, namely the web interface. The architecture design include the allocation of architectural responsibilities to architectural components, as well as tactics used to realize these responsibilities under the current level of granularity to address stated quality requirements.

3.5.3 Architectural Responsibilities, Components and Realization

The architectural components of the Web Application Framework are shown in Figure 3.5

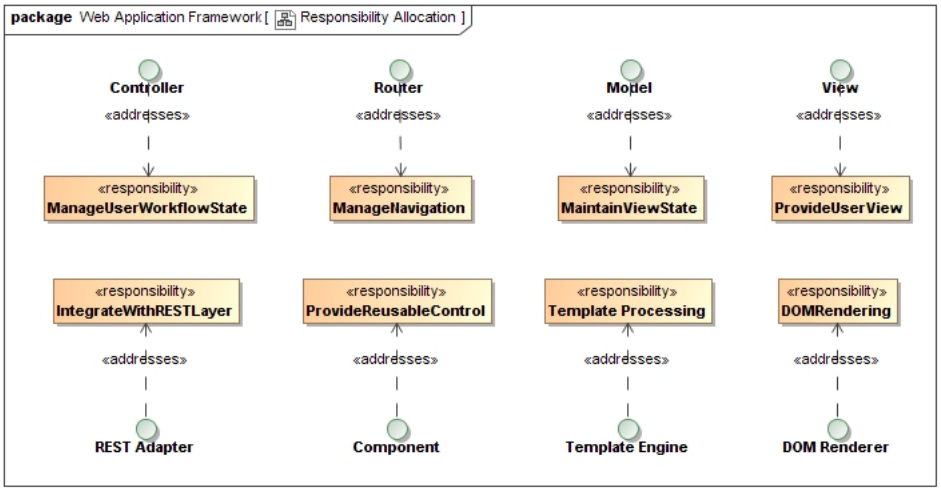

Figure 3.5: The abstract components to which the Web Application Framework responsibilities are assigned.

3.5.3.1 Tactics

Tactics Angular2 uses in order to address the quality requirements should include:

- caching of pre-generated and pre-populated HTML pages for performance
- virtual DOM for in-memory updates, incremental builds and efficient diffing based on differentiation between static and dynamic DOM elements

3.5.3.2 Frameworks and Technologies

The web framework the project will be utilizing will be based on Angular2 which is a JavaScript framework developed by Google Inc. Angular2 is an open-source web application framework which promises a one-way flow of information between model, controller and view. This promise is meant to increase performance and decoupling of the underlying code thereby assisting with maintainability.

Core reasons for using Angular2 include:

- Framework requires a defined structure which aids in maintainability.
- The framework is maintained not only by a community of users but also by Google, which aids in maintainability due to the constant upkeep of the code base and documentation.

- Angular2 supports very good integration with RESTful API services.

Other frameworks that were considered;

- ReactJS
- Ember.js
- Backbone.js

3.6 Reporting

The Reporting Module provides functionality to generate and compare reports of single/or multiple benchmarks, with the option to export them as csv, whilst still complying with the quality requirements defined for the system.

3.6.1 Architecture Requirements

3.6.1.1 Quality Requirements

Flexibility The reporting module should be flexible enough to add different elements, such as graphs, charts and tables. The color scheme should also be easily changed.

Performance The performance of the reports should as minimalistic as possible, such that the page is responsive as possible.

Reliability Each report must be reliable with the data it represents, as this is what the user will be referring to.

3.6.2 Architecture Design

3.6.2.1 Architectural Responsibilities, Components and Realization

The architectural components of the Reporting Module are shown in Figure 3.6

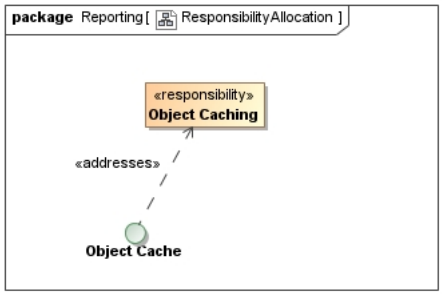

Figure 3.6: The abstract components to which the architectural responsibilities are assigned.

3.6.2.2 Tactics

The Reporting module should implement the following tactics:

- *Report caching* to improve scalability and performance.

3.6.2.3 Frameworks and Technologies

The frameworks and technologies used for reporting will consist of JavaScript, HTML and CSS. With the exporting of reports using a JavaScript library called jsPDF.

We have considered using the jasper reports to generate reports, but it does not provide the full functionality that we are looking for. Such as exporting the graphics of the reports.

Chapter 4

Benchmark Monitor Software Architecture

This section specifies the software architecture requirements and the software architecture design at the second level of granularity - the benchmark monitor application. The monitor will be responsible for benchmarking a user application.

4.1 Monitor

4.1.1 Architecture Requirements

This section discusses the software architecture requirements around the back end benchmark system infrastructure. The monitor will be responsible for retrieving a job specification from the queue, building, compiling and benchmarking a user application based on the job specification. A key concept in the design of the monitor nodes was to ensure monitor nodes remain autonomous to allow for the addition and removal of monitor nodes. In particular the architecture requirements at the second level of granularity is specified.

Figure 4.1 show the a high-level infrastructure view of the Benchmark Monitor service.

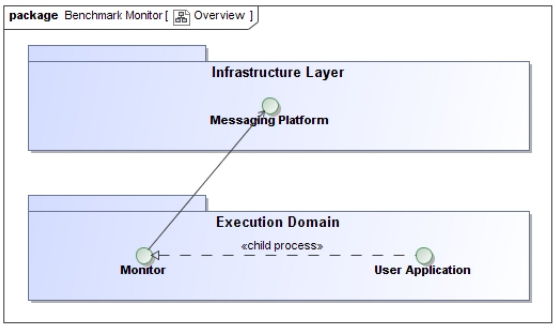

Figure 4.1: A high-level overview of the software architecture for the Benchmark Service

4.1.1.1 Access and Integration Requirements

Access Channels No human access channels are present with the monitor application. A system access channel however is present, allowing the monitor node to communicate with the back end management system through the messaging platform. The messaging platform provides a highly decoupled system, allowing nodes to come and go as necessary. To assist the administrator to monitor the nodes, the management platform implements a monitoring system where nodes can send heartbeat messages to alert the management system of the individuals status. Message can indicate if a node is online and waiting to retrieve jobs, if a host is going to start a benchmark or if a node is shutting down.

4.1.1.2 Quality Requirements

No additional quality requirements are imposed on the system other than that defined in section .

4.1.2 Architecture Design

4.1.2.1 Architectural Responsibilities, Components and Realization

The architectural responsibilities, components and concrete realization of these responsibilities are shown in Figure 4.2

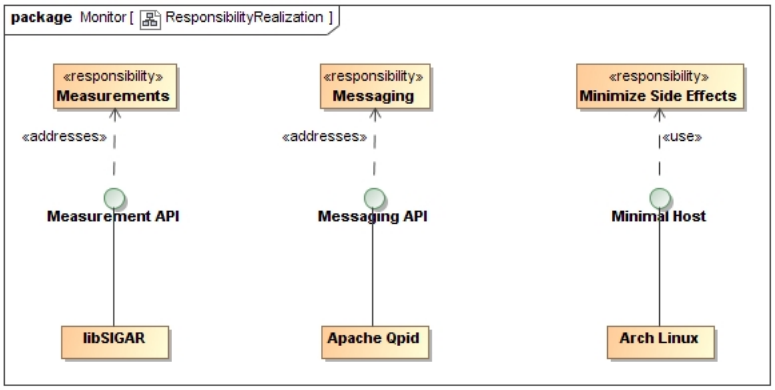

Figure 4.2: The components with which the architecture responsibilities within the monitor is realized

4.1.2.2 Tactics

As the monitor node must ensure accurate benchmarking, it is important to keep the monitor node small and eliminate side effects as far as possible. Tactics to address the quality requirements include:

- *Accurate measurements* must be made when benchmarking the user applications. For this reason an existing library that is used by the greater academic community will be used to make the measurements.
- *Minimizing side effects* will be done by setting up a minimal host operating system, as to minimize, threads running, interrupts and paging. To accomplish this, a minimal Linux host will be setup, will all but the bare components required to accomplish the task of benchmarking.
- *Communicate synchronously* with the queue, to ensure no side effects is introduced. Communication with the queue will only occur when no benchmarking is being done on the host.

4.1.2.3 Frameworks and Technologies

libSIGAR As we require accurate measurements, it is very important to use a library that has been battle-tested in an academic environment. After extensive research, we noticed that the libSIGAR library is use in multiple research projects, is discussed in academic papers and seem to be the current library used to do benchmarking with.

Apache Qpid and Proton As C++ doesn't have native support for communication with a message broker, we require a library that provides support for the AMQPv1

protocol. Apache Qpid together with Apache Proton, providers support to C++ to allow communication with an AMQP broker.

YAML CPP To assist the administrator of the benchmarking service in monitoring node, while keeping nodes autonomous and ephemeral, we will allow the operator of a benchmarking node to configure a YAML file with information related to the node which he/she is operating. This information will then be forwarded to the management system by way of the messaging platform, which will allow the administrator of the benchmarking system to monitor nodes.