



Hyper Perform
Architectural Requirements Specification

Organisation: <https://github.com/Hyperperform>

Developers:

Claudio Da Silva *14205892*

Rohan Chhipa *14188377*

Avinash Singh *14043778*

Jason Gordon *14405025*

Updated September 11, 2016

Contents

1	Software Architecture Scope	3
2	Overall Software Architecture	5
2.1	Quality Requirements	5
2.1.1	Flexibility	5
2.1.2	Maintainability	5
2.1.3	Integrability	5
2.1.4	Scalability	6
2.1.5	Reliability	6
2.1.6	Security	6
2.1.7	Auditability	6
2.1.8	Usability	6
2.1.9	Testability	7
2.1.10	Deployability	7
2.2	Architectural Constraints	8
2.3	Integration and Access Channels	9
3	Architectural Design	10
3.1	Architectural Tactics	10
3.2	Architectural Patterns	10
3.3	Reference Architectures	10
4	Technologies	11
4.1	Back end	11
4.2	Front end	11
4.3	Persistence	11
4.4	Deployment tools	11

1 Software Architecture Scope

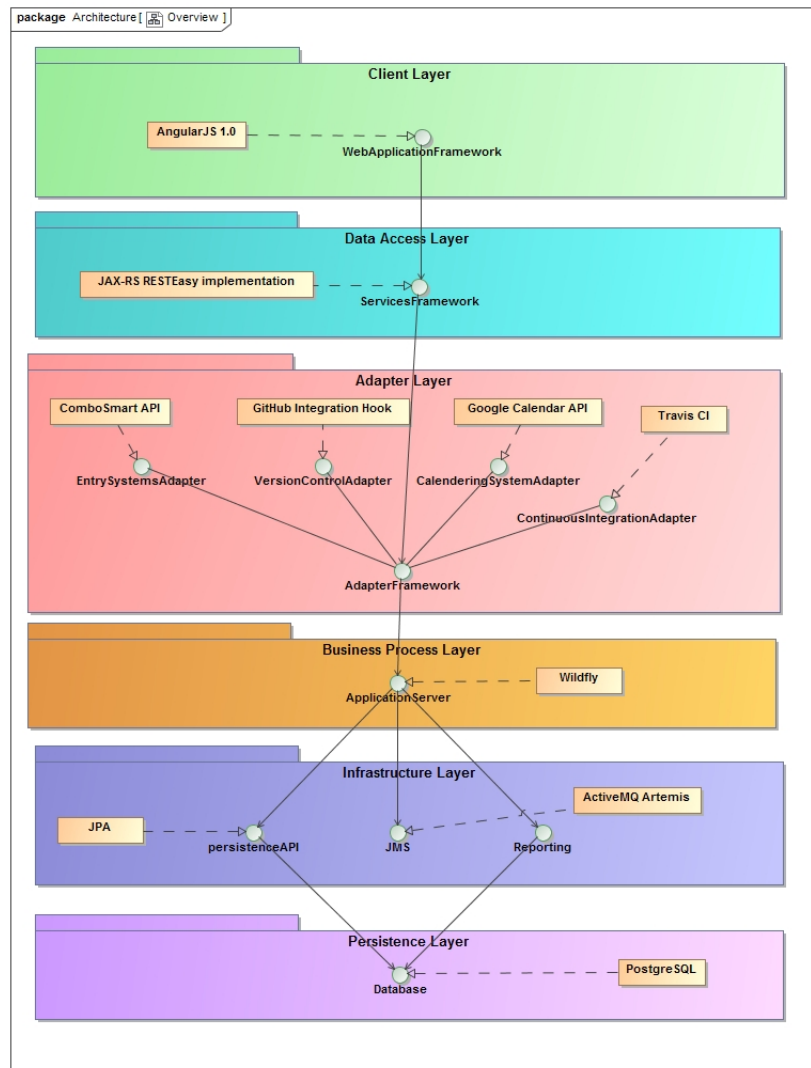


Figure 1: A high level break-down of the architecture to be used

An explanation of some of the adapters follow:

- The EntrySystemsAdapter, provides an interface on to which various entry systems such as card readers, biometric scanners and the like, can map.
- The VersionControlAdapter, provides an interface on to which various version control systems such as GitHub, BitBucket and GitLab can map.
- The CalenderingSystemAdaoter, provides an interface for various calendering systems such as Google Calender, or Outlook calender can map onto.
- The ContinuousIntegrationAdapter, provides an interface on to which various Continuous Integration systems, such as Travis or Jenkins, can map onto.

Extended release adapters will include:

- The BugSystemAdapter, which should provide an interface for bug tracking systems such as Waffle.io or Trello to map on to.
- The WearablesAdapter, which should provide an interface for wearable devices such as Fitbits and Samsung Gears can map on to.

Further information will be provided under integrations.

2 Overall Software Architecture

2.1 Quality Requirements

The following section will focus on the quality requirements of the overall system at it's highest level of granularity, and all the requirements defined are defined for all components of the system.

2.1.1 Flexibility

The most important element of the system is its flexibility. It has been designed in such a way that extra integrations can easily be added. The main focus of the software is to source data from various integrations that change with the current market trends and business environment. Thus the need for additional integrations and replacement of default integrations will arise, allowing for a system fully embracing the concept of Internet of Things.

The system is decoupled in the way that it is not limited to specific architectures, thus it should not be locked down to a specific provider for either the application server or the database.

2.1.2 Maintainability

Another very important part of this system is the maintainability of the system, to the extent that the system should be easy for future contributors to change, maintain and further develop.

- Future developers should be able to easily understand the system, and modify it as needed.
- The technologies chosen for the system can be reasonably expected to be available for a long time as well as remain open source and available. However due to the front-end being completely decoupled from the back-end it would be easy to change the front-end technology without affecting the back-end functionality.
- The system should be able to be maintained without the need to make it unavailable for said maintenance, following the always-up methodology.

2.1.3 Integrability

The system should allow for easy future integration requirements by providing access to it's services using public standards. All external systems should have a preprocessor interface on which to communicate and map on to.

2.1.4 Scalability

The system should theoretically be usable in large industries of thousands of employees. Thus the system should be able to handle the load and grow as needed. To accomplish this we make use of the message bus architecture.

2.1.5 Reliability

The system should be reliable in the fact that many events will be constantly captured, and in the event a system is not able to currently accept an event, that event must not disappear, but must instead remain on a message queue until the system is able to handle it.

2.1.6 Security

This is not the main focus of the system however for any system this requirement is important. The following security measures have been put in place:

- Proper authentication is to be preformed in order to gain access to confidential user information from the database.
- The messages and the message queue as a whole has been protected from attack from outside systems. The messaging queue is on the server which is protected by SSL encryption.
- Outside components can only gain access through the message channel, and only when authorised to do so.

2.1.7 Auditability

Logging should occur for:

- Events captured into the queue.
- Access to the users personal information.

Logs should contain at least for all different logs:

- An id pertaining to user performing the task.
- A description of the activity happening.
- The time at which this activity happened.

2.1.8 Usability

The system has been created to be intuitive and easy to use, the system should not be difficult for a user to learn or interact with. Some things that have been emphasised are:

- Useful and helpful error messages, as well as client side validation as far as possible.
- Labels and hints where possible to guide users.

2.1.9 Testability

The system must be testable through Continuous Integration using:

- Automated isolated tests using mocks
- Automated integration tests using the actual realizations

Tests should ensure that all pre and post-conditions as well as class invariants are properly met.

2.1.10 Deployability

- The system must be built using only the build tool and scripts.
- The system should be deployable on cross platform environments.
- The system should be deployable in environments with different databases and message brokers.
- The system is eventually to be Dockerised for easy distribution.

2.2 Architectural Constraints

The following constraints are non-negotiable and must be met for the system to properly meet client requirements:

- The system should be platform independent, allowing for docker to handle the environment setup.
- All software used in the creation of this project, including frameworks, libraries and application servers should be strictly open source, as well as preferably well supported and long term.
- The web interface should run correctly on all Webkit enabled browsers.

2.3 Integration and Access Channels

In terms of access channels, the project as it stands focuses on two main forms of access:

- The web-client component, which will make use of REST calls in order to interact with the back end server. The web client is written in AngularJS 1.0, and interacts with the JavaEE server using the Jersey implementation of REST wrapping.

With regards to integration, the system has the following adapters on which to map external integrations on to :

- The EntrySystemsAdapter, provides an interface on to which various entry systems such as card readers, biometric scanners and the like can map.
- The VersionControlAdapter, provides an interface on to which various version control systems such as GitHub, BitBucket and GitLab can map.
- The CalenderingSystemAdaoter, provides an interface for various calendering systems such as Google Calender, or Outlook calender can map on to.
- The ContinuousIntegrationAdapter, provides an interface on to which various Continuous Integration systems, such as Travis or Jenkins, can map onto.
- Similarly the events that are retrieved will come through the REST wrapping designed specifically for the events. Each integration has its own REST URL to which it sends events.

Thus since the events come through the REST URL's we don't have to continuously poll these systems. Instead the separate systems send the events when ready. Once the events are received they are processed and placed on a messaging queue.

3 Architectural Design

3.1 Architectural Tactics

At the system level of granularity, that being the highest level of granularity, we do not specify any architectural tactics in order to concretely address the quality requirements for the system.

3.2 Architectural Patterns

Some of the patterns used at the system level include:

- The layered pattern, which limits access of component within one layer to components which are either in the same or the next lower level layer. A great strength it has is that it can be relatively easily replaced. In particular, one can add further access channels without changing any lower layers (A critical component of this system) within the software system and changing the persistence provider would only require changing the persistence API.
- The Message Bus pattern, an enterprise integration pattern which allows for safe and efficient handling of events via messages, using a response and request queue, one can ensure that events and messages never get lost even if part of a system fails, thus allowing a greater fail safe in the system.

3.3 Reference Architectures

The key architecture at the system level is the Enterprise Service Bus. In order to properly network all devices in a proper IOT (Internet of Things) manner, one would implement a messaging queue system. We chose ActiveMQ Artemis as our realization of this pattern, as it integrates quite nicely with JavaEE, especially the WildFly server we are running, and has decent support, as well as many features we can make use of.

4 Technologies

4.1 Back end

- WildFly as the JavaEE container and server.
- JUnit as a unit testing framework along with Spring.
- AspectJ as an Aspect Oriented Programming approach.
- RESTEasy implementation of JAX-RS for REST wrapping.
- ActiveMQ Artemis as the message broker for the system.
- Esper for complex event processing (CEP).

4.2 Front end

- HTML5, CSS/SCSS and SASS, Bootstrap and JQuery.
- AngularJS 1.0 is used to access the REST services.
- ChartJS to create appealing charts on the front-end.

4.3 Persistence

- PostgreSQL for the database.
- JPA as an Object Relational Mapper.

4.4 Deployment tools

- Docker as a shipping container.
- TravisCI as a Continuous Integration tool.