Data Dash

Team Blue Moose Design Document

Team Manager
Ethan Miller
Team Members:
Corwin Burdick
David Dzgoev
Ajibola Famuyibo
Ethan Fine
Harry Juda
Sean Morris
Abdurrahman Munir
Dakota Orion
Chris Raff
Jonathan Shum

Table of Contents

1.0 Overview

1.1 Document Overview

The goal of this document is to outline the organization of the system at a high level, and to explain the choices in structure and choice of technology. To this effect, it will present and explain how data flows between the various components of the system both in words and with diagrams. The six following sections in this document explain the design of our system. Section 2 explains in-depth our architectural design, clearly outlining users, databases, and modules. Section 3 explores the purposes of the modules in our system. Section 4 lays out the APIs that we will use and how modules will make use of them. Section 5 contains a list of the frameworks and 3rd party softwares that we will utilize. Section 6 expands on Section 5 and explains our reasoning for choosing each framework. Section 7 provides a glossary for reference.

1.2 System Overview

The goal of this project is to provide a GUI for liberty mutual to interact with their database in a way that is more user-friendly and less error-prone than manual interaction. The GUI will be accessible to the user as a web application. This web application will be built from a system of modules, the main six being the Web UI, Web Translator, Mediator, Internal Data Access Layer, Authentication Data Access Layer, and Liberty Mutual Data Access Layer. The Web UI communicates with the Mediator through the Web Translator, and the Mediator facilitates interactions between the libraries.

2.0 Architectural and High Level Design

2.1 Architectural View

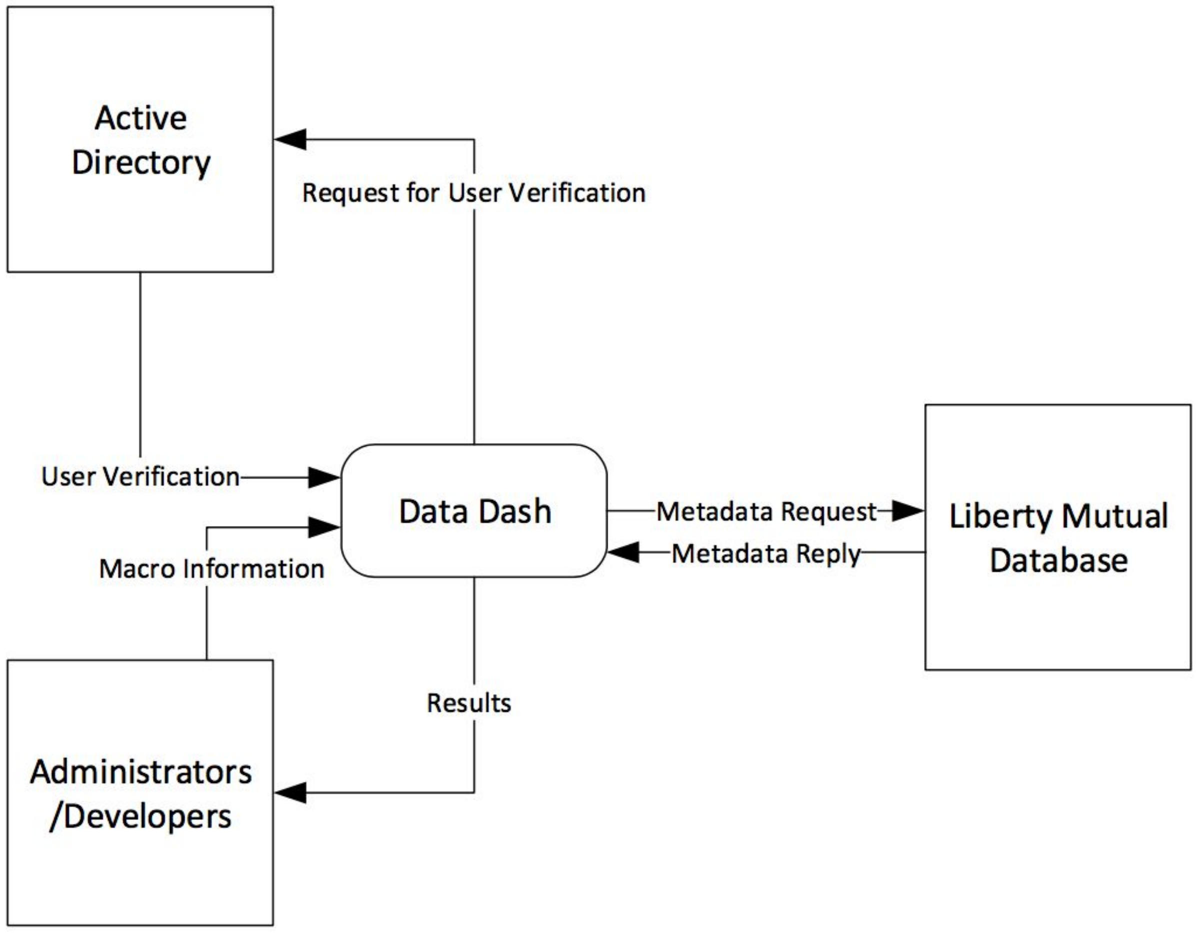
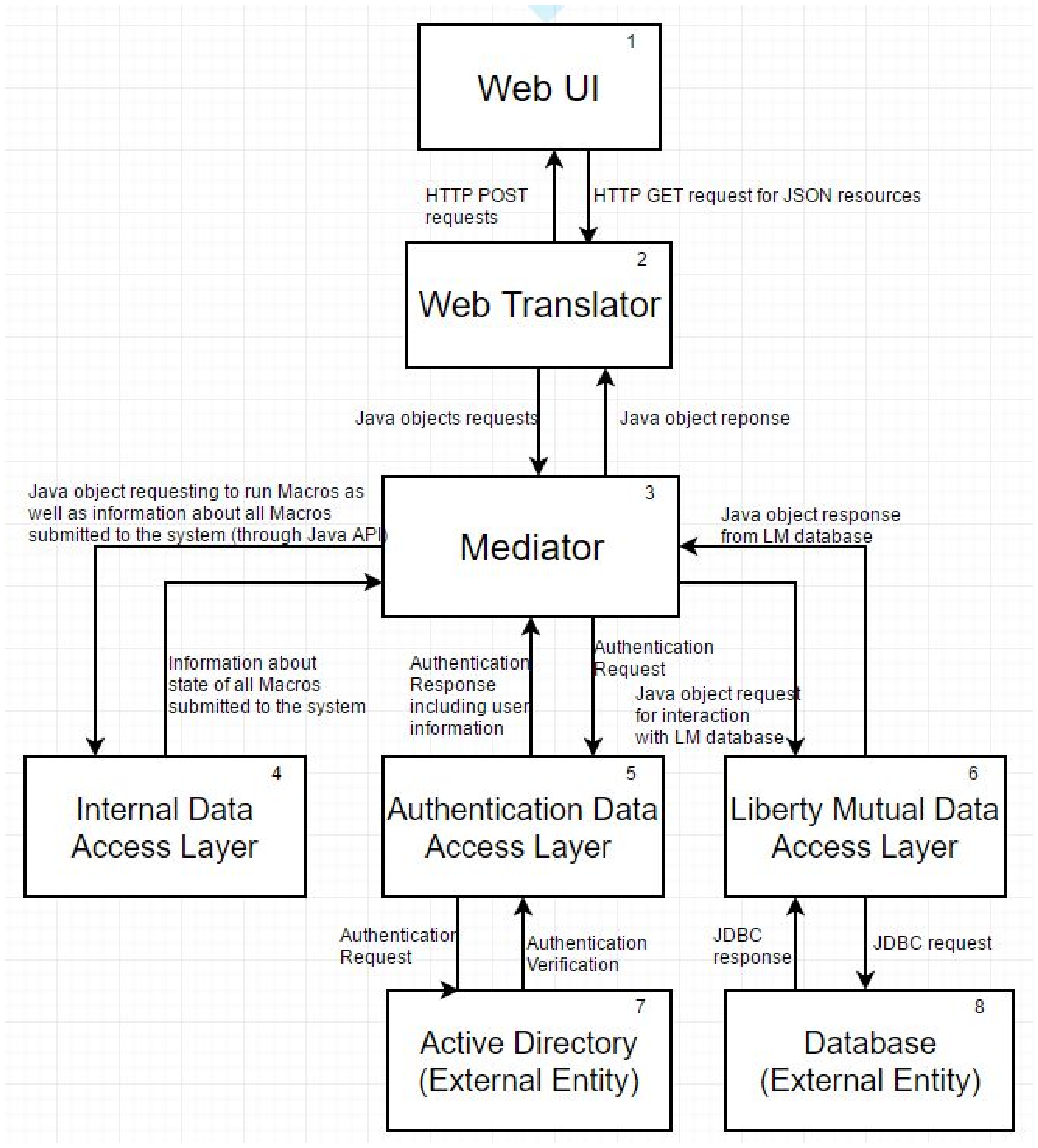


Figure 2.1 - Context Level DFD

2.2 High Level View



1
Web UI
HTTP POST requests
HTTP GET request for JSON resources
2
Web Translator
Java objects requests
Java object reponse
3
Mediator
Java object requesting to run Macros as well as information about all Macros submitted to the system (through Java API)
Java object response from LM database
Information about state of all Macros submitted to the system
Authentication Response including user information
Authentication Request
Java object request for interaction with LM database
4
Internal Data Access Layer
5
Authentication Data Access Layer
6
Liberty Mutual Data Access Layer
Authentication Request
Authentication Verification
JDBC response
JDBC request
7
Active Directory (External Entity)
8
Database (External Entity)

Figure 2.2 - Module System Diagram

1. Web UI, 2. Web Translator, 3. Mediator, 4. Internal Data Access Layer,
5. Authentication Data Access Layer , 6. Liberty Mutual Data Access Layer, 7. Active Directory,
8. Database

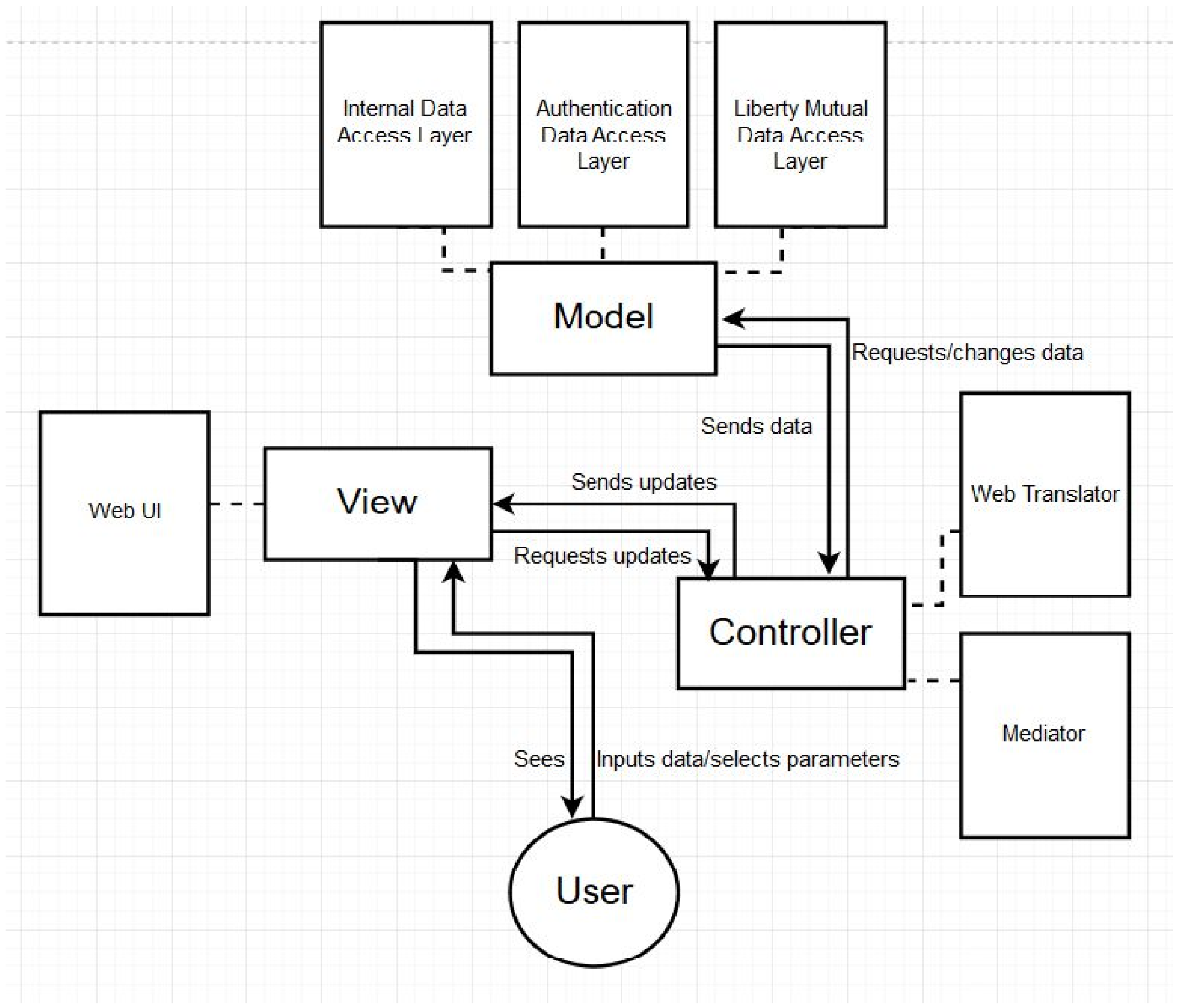

Internal Data Access Layer
Authentication Data Access Layer
Liberty Mutual Data Access Layer
Model
Requests/changes data
Sends data
Web UI
View
Sends updates
Requests updates
Web Translator
Controller
Mediator
Sees
Inputs data/selects parameters
User

Figure 2.3 - Model View Controller Diagram as it relates to our Modules.

The architectural view shown in figure 2.2 shows exactly how data is sent within our system and how different components of the system interact, and the second diagram shown in figure 2.3 is a higher level abstraction of the system in terms of the model view paradigm and its interactions with users.

2.2.1 Application Design & Interaction

The Data Dash application is split into 6 modules that interact with each other only through well defined APIs. The modules are divided into multiple levels. The levels, from highest to lowest, are the UI level, the translation level, the mediation level, and the data access level. Each module can only request resources from the modules one level lower than it. We designed one module for the UI level, the translation level, and the mediation level, and 3 modules for the data access level. However, the system is designed so that arbitrarily many modules can fit in at the translation level and UI level without any change at the other levels.

As indicated in Figure 2.3, the Web UI corresponds to the View. The Web Translator and Mediator correspond to the Controller. The Internal Data Access Layer, Authentication Data Access Layer, and Liberty Mutual Data Access Layer correspond to the Model. The traditional Model View Controller paradigm involves the Model, View, and Controller all interacting with each other albeit in a structured way. Our system imposes additional constraints on how our modules interact with each other. Each module can only interact with modules in the level directly above or below it. For example, our Web UI never directly interacts with our Data Access Layer. In addition, communication is always initiated from top to bottom - modules at the lower level will never initiate communication with modules at a higher level, they just respond to communication from the module at the higher level. That is to say, for example, the Internal Data Access Layer will never initiate an interaction with the Web Translator. For more information about levels, see section 3.

3.0 Module Level Description

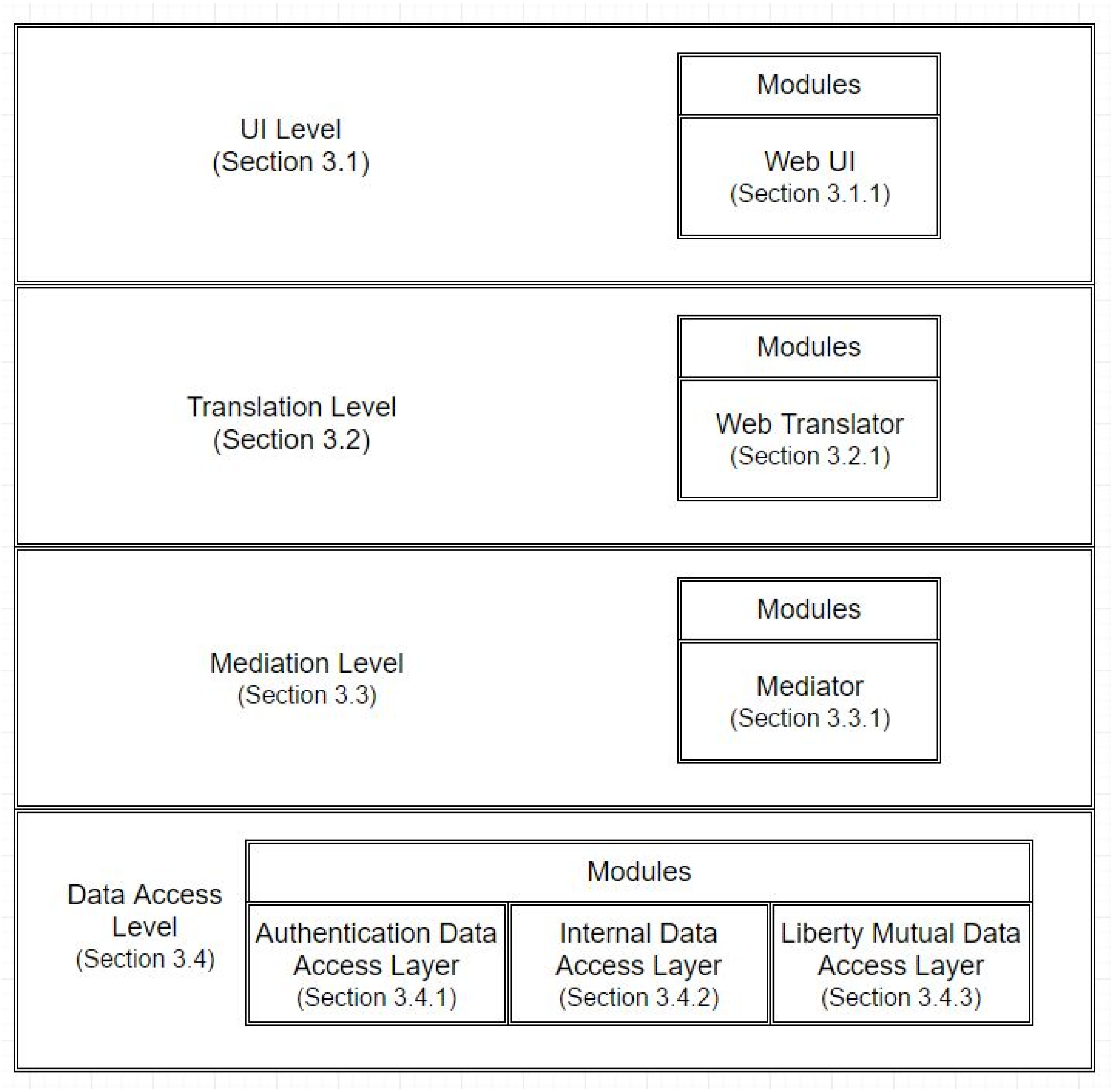


Figure 3.1 - Levels Diagram

Our system is divided into levels based on how far removed functionality is from the user, and then further divided into modules.
A level is a grouping of modules. Modules at a given level can access functionality of modules at the level directly below their level, and provide services to modules at the level directly above their level.

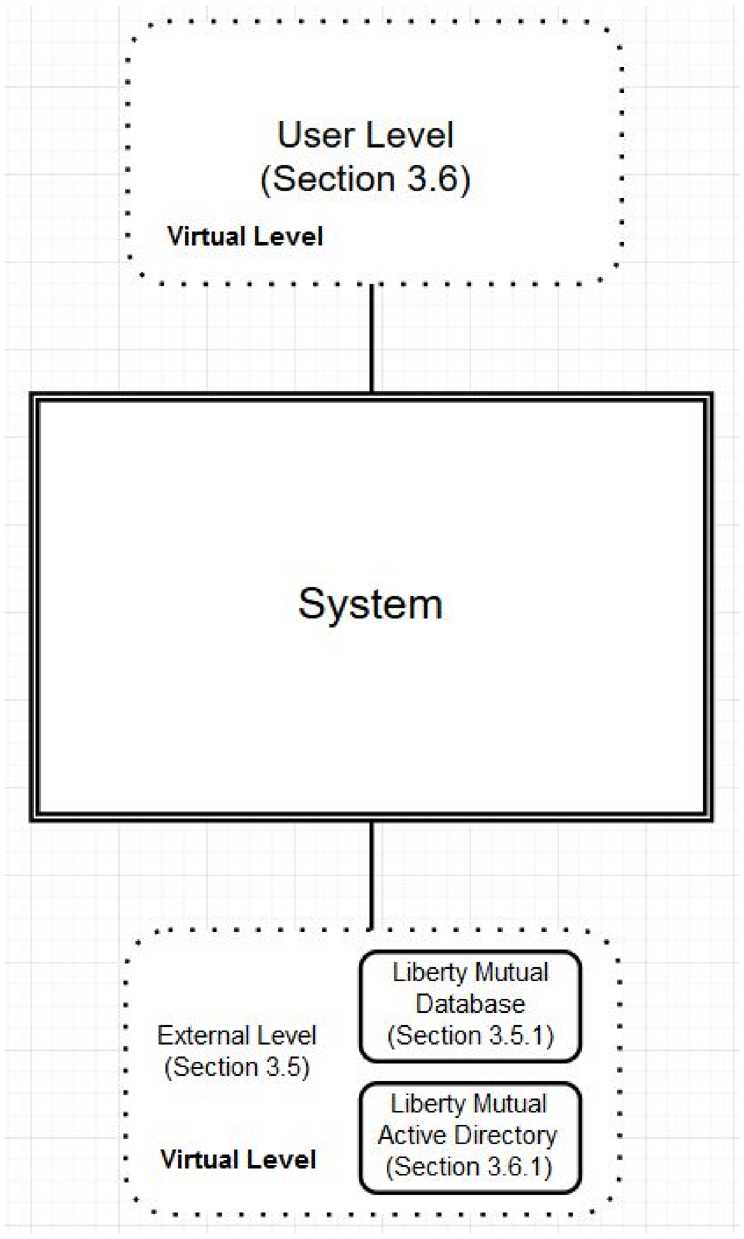

User Level
(Section 3.6)
Virtual Level
System
External Level
(Section 3.5)
Virtual Level
Liberty Mutual Database (Section 3.5.1)
Liberty Mutual Active Directory (Section 3.6.1)

Figure 3.2 - Virtual Levels Diagram

There are also virtual levels, which are groupings of entities not part of our system. See Figure 3.2 as well as sections 3.5 and 3.6.
A module is a grouping of system functionality which interacts with other modules through well defined APIs. See section 4.0 for those APIs.
We have a particular naming convention for modules. Modules at the UI Level will end with the word “UI”. Modules at the Translation Level will end with the word “Translator”. Modules at the Data Access Level will end with the words “Data Access Layer”. None of these words have any

particular meaning outside of naming convention, but the naming convention allows someone to immediately recognize what level a module is at.

3.1 UI Level

Modules at the UI level are responsible for exposing system functionality to users of the system. A module at the UI level must communicate with one of the modules at the Translation level, using an API defined for that particular module.

3.1.1 The Web UI

(See figure 2.2 #1)
The only UI we will be providing will be the Web UI module. The Web UI module will consist of a set of files served statically by the Web Translation module. A web browser will be able to interpret the files and run the module on the user's computer. After the initial static files are served, further requests for resources will be made to the Web Translation module using JSON objects through a restful interface.

3.2 The Translation Level

Modules at the Translation Level are responsible for exposing the systems functionality via a particular API. No real system logic is being performed at the Translation Level, simply a translation of requests into java objects, and a translation of java objects into responses, such that the requests are producible by the UI Level, and the responses are readable by the UI Level. After translating requests to java objects, the Translation Level sends the requests to the mediator's java API to fulfill them.

3.2.1 The Web Translator

(See figure 2.2 #2)
The only translator we will be providing will be the Web Translator Module. The Web Translator module will consist of a spark-java web server which will provide the Web UI with three services:

1. When receiving an HTTP GET request for a static file that is part of the Web UI, the Web Translator will serve that static file.
2. When receiving a HTTP GET request for anything else, the Web Translator will serve the static file index.html, which will represent the root of the Web UI and from there browsers will request all other needed files.
3. The Web Translator will respond to HTTP POST requests according to its API. Through this API it will display all of the systems functionality.

In order to provide the third service, the web translator will translate JSON objects into java objects and make use of the Mediator's API. It is notable that the third service can also be used by any future UI layer so long as that UI layer is sending http requests as its method of communication.

3.3 The Mediation Level

Unlike every other level, there should only ever be one module at the Mediation Level. The job of the module at the Mediation level is to expose the entire system's functionality through a purely java API, so all modules at the Translational Level can access it in exactly the same way. In order to fulfill those requests, the Module must make use of the services provided by the Data Access Level.

3.3.1 The Mediator

(See figure 2.2 #3)
The mediator will implement the Mediation Level API, making use of the API's of the Data Access Level.The Mediator will, on a best effort basis, be configurable in the following ways: AllowSkipPeerReview = True/False : when set to false requests to skip peer review will be rejected.

3.4 The Data Access Level

All persistent and/or external data is accessed through a module at the Data Access Level. Each module has its own responsibilities, and its own API. They each provide a purely java API to be used by the Mediator.

3.4.1 The Authentication Data Access Layer

(See figure 2.2 #5)
The Authentication Data Access Layer(ADAL) is responsible for validating user credentials, and providing necessary user information to the mediator. It also is responsible for producing authentication tokens, and validating those tokens. The ADAL will communicate with Active Directory in order to validate login information, and will use a private key encryption library to generate and validate tokens. (see figure 2.2)

The ADAL will, on a best effort basis, be configurable in the following ways:

1. GroupName = String: The name of the Active Directory group that grants access to the system.
2. ExpirationTime= int: The time, in hours, that authentication tokens are valid for, before the user must login again.

3.4.2 The Internal Data Access Layer

(See figure 2.2 #4)
The Internal Data Access Layer (IDAL) is responsible for storing and keeping track of all macros submitted to the system, both the ones awaiting peer review, and those that have already been

run. It gives the mediator access through a purely java API.
The IDAL will persist data using an embedded HyperSql database.

3.4.3 The Liberty Data Access Layer

(See figure 2.2 #6)
The Liberty Data Access Layer (LibDAL) s responsible for all interaction with Liberty Mutual's data. In our design, all interaction with Liberty data will be done over JDBC connection to the Liberty Mutual SQL database, however future requirements like runname creation, which require communication with Liberty data not in the database, should also be implemented in the LibDAL.
Because the LibDAL must execute the macros, all macro definitions come from the LibDAL. The LibDAL is responsible for providing a list of all macros it can execute, and for giving instructions on what sorts of data the parameters for the macros can be. The LibDAL is also responsible for providing Liberty Data to the mediator, and for the actual execution of macros.
The LibDAL will definitely be configurable in the following ways:

1. Host = JDBC:etc : The url used to connect to the liberty mutual database.
2. Username = String: The username used to connect to the liberty mutual database.
3. Password = String: The password used to connect to the liberty mutual database.

The LibDAL will, on a best effort basis, be configurable in the following ways:

1. OptimizeSpeed = true/false : when set to true the LibDAL will avoid expensive database operations when possible, especially by avoiding giving the “List” response to queries about parameter possibilities
2. AlwaysReportFailures = true/false : when set to true the LibDAL will block when executing macros, and report runtime failures of macros.

3.5 The External Level

While not technically a level of our system, being external, our design includes an External Level as a virtual level below the Data Access Level, to show that the external entities are service providers to the Data Access Level in the same manner that every level in our system is a service provider to the level above it.
There are two external entities we are concerned with.

3.5.1 The Liberty Mutual Database

(See figure 2.2 #8)
The Liberty Mutual Database is an entity external to our system accessed only by the LibDAL. The Liberty Mutual Metadata Database serves the LibDAL by executing queries and providing results. We expect and require the Liberty Mutual Metadata Database to be accessible over JDBC and execute arbitrary SQL.

3.5.2 The Liberty Mutual Active Directory

(See figure 2.2 #7)
The Liberty Mutual Active Directory is an entity external to our system that is accessed only by the ADAL. The Liberty Mutual Active Directory serves the ADAL, by authenticating usernames and passwords, and providing data about users, at the very least a first and last name. We expect and require our application to be allowed to access Active Directory over LDAP.

3.6 The User Level

While not technically a level of our system, our design includes a user level as a virtual level above the UI level to show that modules at the UI level are service providers to end users of the system in the same manner as modules at one level are service providers to the level above them. All access to the system by end users will be through a module at the UI level. (Except, technically, the initial get request to retrieve the Web UI Module from the Web Translator).

4.0 APIs

Each module has its own API. Any module implementing the exact same API can be easily interchanged without needing to change any of the other modules.

4.1 The Web Translator API

The web translator receives requests in the form of HTTP POST requests with a JSON object in the body, and sends responses with a JSON object as the body.
Every Response json object has a line for status. The status can be one of the following strings:

1. SUCCESS: the request was successfully carried out.
2. INTERNAL_ERROR: the request failed, because of an error.
3. AUTHENTICATION_ERROR: the authentication token could not be decoded.
4. AUTHENTICATION_EXPIRATION: the authentication token was expired.
5. BAD_REQUEST: Something was structurally wrong with the request. The translator was able to convert it to a java object, but the java object had incorrect list sizes or invalid values.
6. TRANSLATION_FAILURE: Something was structurally wrong with the request. The translator was unable to translate it into a java object.
7. FAILURE: The request could not be carried out, but it was not because of an error.

If status is anything other than SUCCESS, then the response is considered an error. In this case, regardless of the request type, the Web Translator has responded with a JSON object like this:

```
{
        status: String
        reason: String
}
```

Path and Description	Parameters	Return
1. GET / Renders Index.html, the root of the Web UI module	None	None
2. GET /<filename> Retrieves the static file from the server	None	None

3. POST /login
 	
Submission of user name and password to Active Directory to see if combination is valid. If correct, redirect to logged in homepage, otherwise return to login page with an error message.
	{
	username: String,
	password: String
	}
		{
		status: String,
		authentication: String,
		fname: String,
		lname: String
		}

4. POST /macro
 	
Gets the list of macros that can be run.
	{
	authentication: String,
	}
		{
		macroList: [
		macro:{
		macroID: String,
		macroName: String,
		macroDescripion: String,
		parametersIDs:[
		String
		],
		parametersNames:[
		String
		]
		}
		]
		}

5. POST /macro/runMacro
 	
Sends request to run selected macro with parameters.
	{
	authentication: String,
	macroType: String,
	parameters: [
	String
	],
	skipReview: boolean
	}
		{
		status: String,
		message: String
		}

6. POST
/macro/viewHistory
 	
Sends request to view history of macros that have been run in the past.
	{
	authentication: String,
	startDate: String,
	endDate: String
	}
		{
		macroList: [
		macro:{
		uniqueID: String,
		creatorFname: String,
		creatorLname: String,
		reviewerFname: String,
		reviewerLname: String,
		wasPeerReviewed: boolean,
		runDate: String,
		creationDate: String,
		macroType: String,
		Parameters:[
		String
		],
		originalParameters:[
		String
		]
		}
		]
		}

7. POST /macro/failures
 	
Send request to view all macros that have failed.
	{
	authentication: String,
	startDate: String,
	endDate: String
	}
		{
		macroList: [
		macro:{
		uniqueID: String,
		creatorFname: String,
		creatorLname: String,
		reviewerFname: String,
		reviewerLname: String,
		wasPeerReviewed: boolean,
		runDate: String,
		creationDate: String,
		macroType: String,
		Parameters:[
		String
		],
		originalParameters:[
		String
		]
		}
		]
		}

8. POST /step/running	{ authentication: String }	{ steps:[driverStep: { driverStepDetailID: String, auditID:String, driverStepID: String, appName: String, runName:String, groupNumber: String, runOrderNumber: String, runStatusCode: String, errorProccessNumber: String, sessionStartDateTime: String, sessionEndDateTime: String, runStartDateTime: String, runEndDateTime: String, createDateTime: String, lastModifiedDateTime: String }] }

9. POST /step/past
 	
Sends request to view steps that were run during a specified time period.
	{
	authentication: String,
	startDate: String,
	endDate: String
	}
		{
		steps:[
		driverStep: {
		
		driverStepDetailID: String,
		auditID:String,
		driverStepID: String,
		appName: String,
		runName:String,
		groupNumber: String,
		runOrderNumber: String,
		runStatusCode: String,
		errorProccessNumber: String,
		sessionStartDateTime: String,
		sessionEndDateTime: String,
		runStartDateTime: String,
		runEndDateTime: String,
		createDateTime: String,
		lastModifiedDateTime: String
		}
		]
		}

10. POST /step/average
 	
Sends request to view runtime average of a step over a specified time period.
	{
	authentication: String,
	startDate: String,
	endDate: String,
	stepID: String
	}
		{
		time: String
		}

11. POST /journal
 	
Sends request to view the journal of changes.
	{
	authentication: String,
	startDate: String,
	endDate: String
	}
		{
		macroList: [
		macro:{
		uniqueID: String,
		creatorFname: String,
		creatorLname: String,
		reviewerFname: String,
		reviewerLname: String,
		wasPeerReviewed: boolean,
		runDate: String,
		creationDate: String,
		macroType: String
		Parameters:[
		String
		],
		originalParameters:[
		String
		]
		}
		]
		}

12. POST /peerreview
 	
Gets list of macros which need peer review.
	{
	authentication: String,
	}
		{
		macros: [
		macro:{
		uniqueID: String,
		creatorFname: String,
		creatorLname: String,
		reviewerFname: String,
		reviewerLname: String,
		wasPeerReviewed: boolean,
		runDate: String,
		creationDate: String,
		macroType: String,
		Parameters:[
		String
		],
		originalParameters:[
		String
		]
		}
		
		]
		}

13. POST /peerreview/review
 	
Review a pending macro.
	{
	authentication: String,
	macroID: String,
	parameters:[
	String
	]
	}
		{
		status: String,
		message: String
		}

Table 4.0 - Method Descriptions

All dates are Strings containing an integer representing the number of milliseconds since the beginning of 1970.

4.2 The Java Module API

Each of the java modules we will design will implement a well defined interface.

1. The Web Translation module (and any future Translation Level modules) will implement Translator interface to allow the program to start and stop translation services.
2. The Mediator module will implement the MediatorInterface to allow translation level modules access to the system through a java API.
3. The ADAL will implement the ADALInterface, to give the mediator a simple java API to access authentication data
4. The IDAL will implement the IDALInterface, to give the mediator a simple java API to access internal persistent data.
5. The LibDAL will implement the LibDALInterface, to give the mediator a simple java API to access Liberty Mutual data.

5.0 Third Party Software

5.1 Third Party Software Documentation

5.1.1 - **Java** - general-purpose, few dependencies, all group members have previous experience.
5.1.2 - **ReactJS** - A JavaScript library for building user interfaces.
5.1.3 - **React-Bootstrap** - popular front-end framework rebuilt for React, for views and design. We use it as our primary framework for the Web UI.
5.1.4 - **Maven** - software project management and comprehension tool to manage a project's build.
5.1.5 - **Spark** - A micro framework for creating web applications in Java. We use it for the Web Translator.
5.1.6 - **Json** - Lightweight data-interchange format for passing info between server and client
5.1.7 - **Typesafe Config** - Configuration library for JVM languages.
5.1.8 - **HyperSQL** - SQL database that can be embedded in java applications. We use it as an embedded database in the IDAL.
5.1.9 - **Querydsl** - A framework for creating type-safe SQL-like queries for various backends in Java. We use it to talk to the Liberty Mutual Database over JDBC in the LibDAL.
5.1.10 - **Microsoft Azure Active Directory Library for Java** - Allows for fast and easy integration of Active Directory services into applications.

5.2 Why Our Frameworks/Libraries Were Chosen

5.2.1 - Java
We chose to use Java as our main language because all of the group members have previous experience in it. In addition, Java offers a very general purpose solution with few dependencies.

5.2.2 - ReactJS
The use of react components and the virtual DOM allows a single web page to easily contain all views of the frontend and all necessary routing. This allows the frontend to be entirely self contained, and prevents the backend from needing to provide additional routes for every view. Both of these things were important factors in our group picking ReactJS.

5.2.3 - React-Bootstrap
We picked Bootstrap because it is the most widely-used CSS framework, and React-Bootstrap takes advantage of React to make the code much cleaner.

5.2.4 - Maven
We picked Maven because it allows us to easily manage dependencies and build configurations.

5.2.5 - Spark Java
We pick Spark because it is a very simple web framework for Java that lets us easily create REST API's and increase development speed. We will be using this for our web server and in the web translator module.

5.2.6 - JSON
We picked JSON because it is a very lightweight and easy to use data format that we will be using to send data between the client and server.

5.2.7 - Typesafe Config
We picked Typesafe Config because it allows Data Dash to be configurable with a JSON-like configuration language. In particular, Typesafe Config allows us to alter what database the LibDAL points at and the name of the active directory group the ADAL uses without changes to code.

5.2.8 - HyperSQL
HyperSQL is an SQL database, we chose it because it can be embedded in Java applications, which is a very important feature for this project.

5.2.9 - Querydsl
We picked Querydsl because it is a library which allows SQL commands to be written using API calls and methods instead of query strings. We picked it because it allows us to query SQL databases using Java instead of having to write SQL AND Java. This also allows for typesafe queries and easier error checking at compile time instead of runtime.

5.2.10 - Microsoft Azure Active Directory Library for Java
Liberty Mutual has specified that this project should use Active Directory (AD) for authentication of its users. Microsoft's Azure library for Active Directory allows us to easily interact with Active Directory.

6.0 Architectural Rationale

6.1 Why Our Architecture Was Chosen

The architecture underlying our web application is in the RESTful (Representational State Transfer) style. The reason for choosing this style is that it provides a simple yet full-featured way for the client and server of our system to interact with each other through HTTP requests (get/post/put/delete).

We chose the MVC (Model View Controller) software design paradigm for developing our system. The main reason for this decision is that this design pattern essentially hides the actual logic of the application from the user and vice versa. It allows us to separate the system into modules that are further separated into hierarchical levels with the UI Level on top and the Data Access Level at the bottom.

The advantage of our level based design is that our modules only have to interact with the modules directly above and directly below them in the hierarchy. For example the UI level will never directly need to access the Data Access Level. This allows us to define interfaces in each of the modules for interacting with other modules. The benefit of this is that removing any particular module does not alter or break anything in any other module. This provides the ability to dynamically and easily remove and add modules at will.

The Web Translation module exists because it abstracts the rest of the system away from the UI. For example if a client wanted to use a different internally developed Web UI module, they could remove the existing one and design the new one around fulfilling the interface required by the Web Translation Module. The Mediator module provides uniform access with the rest of the system through a Java api allowing for modules at the Translation Level to be added without changing any other modules. The Authentication Data Access Layer allows the mediator to access Active Directory through a Java api. This is advantageous because it makes how authentication is handled by the system modular, which allows for more flexibility and easier testing. The Internal Data Access Later grants the mediator access to the system's internal data through a simple Java api, which grants the system the ability to swap or edit the way internal data is accessed without breaking any other components of the system. Finally, the Liberty Mutual Data Access Layer allows the mediator to access the Liberty Mutual data through a simple Java api which allows it to be changed or swapped out easily.

7.0 Glossary

ADAL	Authentication Data Access Layer.
Authentication Token	Used to allow persistent login and encode user information. Sent with every request requiring login. Becomes invalid after a certain period of time post creation.
HTTP GET (GET)	Request that returns a file in this application the returns will be primarily static files if it successfully find the file
HTTP POST (POST)	Request for additional information from the client.
IDAL	Internal Data Access Layer.
JDBC	Application programing interface defining how a client may access a database.
JSON objects	www.json.org/
LDAP	Core protocol that is supported by Active Directory, as described in RFC 2251 (LDAPv3) and RFC 1777 (LDAPv2).
Level	Level refers to what a module can communicate with. Each module is at a level, Modules can only make requests to other modules that are one level below themselves.
Module	A module is a grouping of system functionality that interacts with other modules through clearly defined APIs.
LibDAL	Liberty Data Access Layer.

SQL	(Structured Query Language) is a special-purpose programming language designed for managing data held in a relational database management system (RDBMS), or for stream processing in a relational data stream management system (RDSMS).
User	A user of the system who accesses the system through a UI.
External Entity	Any entity external to the system that the system interacts with.

Table 7.0 Glossary