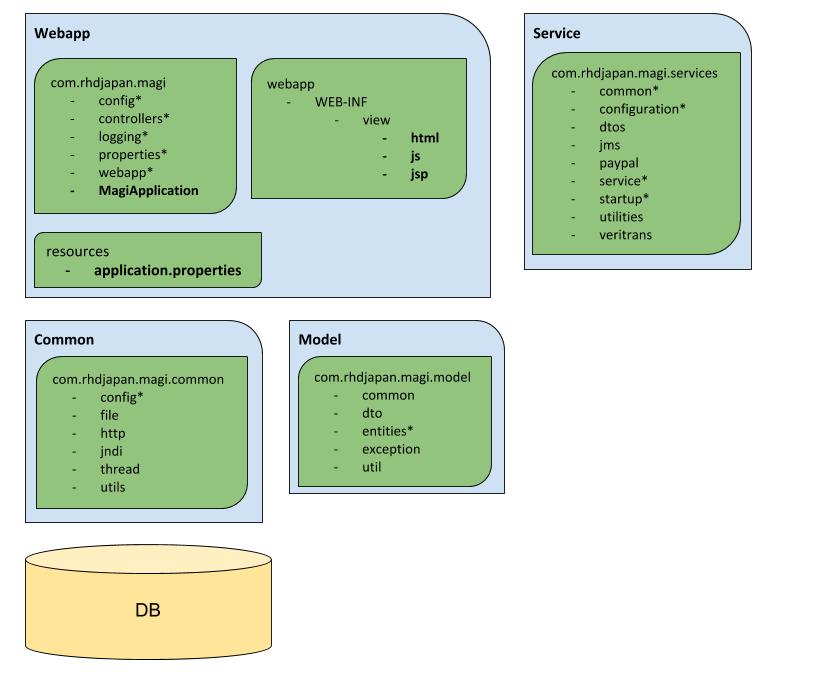
# **Magi Application/Move to SpringBoot**

**(V 2.0, Oct. 11, 2019)**

This document is an ongoing document and subject to change. For previous versions refer to the document’s Version History

# **State of Application (Oct 11, 2019)**

### **Spring Boot POC (\* = contains Spring resources (Configurations, Services, Repositories, etc))**



This Proof of Concept can do the following so far

* Handle Login/Authentication with Spring Security. Allows use of UserDetailsService provided by us
  + DaoAuthenticationManager and MD5 PasswordEncoder
* Handle rendering of pages to Tomcat via SpringBoot defaults
  + Tomcat handlers are directed to JSP via **application.properties**
* Use SpringBoot property classes to strongly-typed properties to be used
  + Allows the handling of properties via @Autowired configuration instead of @Value, and reduces the dependency on Build/Env/BuildPropertyUtils, multiple specified **\*.properties** files.

### **SpringBoot Issues and Resolutions**

The following is a rolling record of the issues that were found during installation and hookup of SpringBoot. Problems are listed in **chronological order**, and some older problems are likely not applicable.

* **Problem:** Maven was unable to build a WAR file and use it for a reference for Spring-Boot
  + **Solution:** Was an issue with regards to the building of the SpringBoot application. *webapp-overlay-main* was set to be a WAR , and the POM was modified
* **Problem:** LoggerFactory is not a Logback LoggerContext but Logback is on the classpath. Either remove Logback or the competing implementation (JDK14LoggerFactory loaded from file:/slf4j-jdk14-1.7.28.jar). If you are using WebLogic you will need to add 'org.slf4j' to prefer-application-packages in WEB-INF/weblogic.xml: org.slf4j.impl.JDK14LoggerFactory
  + **Solution:** This was due to a clash in the Log4J JARs and the JARS provided by SpringBoot *web-starter* dependency. The solution was to exclude the logback-classic artifact from both *spring-boot-starter-web* and *spring-boot-starter* JARs
* **Problem:** 'dataSource' not configuring correctly, causing an issue in Flyway configuration
  + **Solution:** added properties to **application.properties** to allow hibernate properties to be loaded by Spring.
* **Problem:** Issue finding the database by name put in the property file. Root cause was FlywaySqlException: Unable to obtain connection from database: Unknown database 'dev\_magi\_entites'
  + **Solution:** making sure the *spring.datasource.url* property was correctly named to the right URL
* **Problem:** Flyway was having configuration issues out of the box. The root cause was FlywayException: No value provided for placeholder expressions: ${schemaPrefix}. Check your configuration!
  + **Solution:** added *spring.flyway* properties to **application.properties**
* **Problem:** FlywayException: Found more than one migration with version 1.1.0.0
  + **Solution:** adding/modification of *spring.flyway* properties to **application.properties**
* **Problem:** More Flyway! IllegalStateException: Cannot find migrations location in: [db/migration/test\_magi\_entities] (please add migrations or check your Flyway configuration)
  + **Solution:** moving of the migration scripts under src/main/resources (this was when we had a separate module for SpringBoot. These files already existed in webapp-overlay-main)
* **Problem:** Hibernate collision issue. Root cause was MappingException: Repeated column in mapping for entity: com.rhdjapan.magi.model.entities.part.PartStatus column: dtype (should be mapped with insert="false" update="false")
  + **Solution:** For data type Status in Magi, we made use of @DiscriminatorValue to help Hibernate deal with inherited column.
* **Problem:** Hibernate complained about being unable to find the DB. Root cause was NoSuchBeanDefinitionException: No bean named 'RHDManagementPlatformPU' available
  + **Solution:** Moved to the default name for the DB, provided by Hibernate/SpringBoot. For this, we also modified the Services to stop checking for the entity manager, and use the one autowired.
    - (**Note: this will likely need to be remedied, as the POC showed that the services might not actually have the EM properly wired into them. This will be resolved as we move services to where the DAO legitimately is**)
    - Made modifications to AppConfig, allowing it to set up the entityManagerFactory
      * **Note: need to determine if this is necessary or if we can use “out of the box” spring configuration**
* **Problem:** A component required a bean of type 'Utilities.TicketRHD' that could not be found.
  + **Solution:** The class needed to be moved to the appropriate place, or wired to be a bean. Given how many resources it uses, it is now a @Component
* **Problem:** ClassNotFoundException: Could not load requested class com.rhdjapan.magi.core.interceptor.HibernateInterceptor
  + **Solution:** moved the pointer to the interceptor’s new location in *com.rhdjapan.magi.services.utilities.interceptors.HibernateInterceptor*
* **Problem:** java.lang.IllegalArgumentException: Could not resolve placeholder 'activeMQ.serverUrl' in value "${activeMQ.serverUrl}"
  + **Solution:** Fixed to use ActiveMQ properties built up in SpringBoot
* **Problem:** No Persistence provider for EntityManager named RHDManagementPlatformPU
  + **Solution:** Moved to using Hibernate out of the box. Removed “RHDManagmentPlatformPU” determination for DB setup
* **Problem:** SEVERE: Error while initialising Veritrans properties file: java.nio.file.NoSuchFileException: classpath:3GPSMDK-local.properties
  + **Solution:** added Veritrans properties to **application.properties**, and allowed VeritransConfig to handle the setup of the property file locations.
* **Problem:** Cause: Could not connect to broker URL: ssl://virtlocal.rhdjapan.com:61617. Reason: javax.net.ssl.SSLHandshakeException: java.security.cert.CertificateException: No name matching virtlocal.rhdjapan.com found
  + **Solution:** Added SSL properties to **application.properties**
    - **Note: This is currently not working, as SSL seems to block the 8888 port on Tomcat when SpringBoot tries to boot up.**
* **Problem:** java.lang.NullPointerException; com.rhdjapan.magi.common.utils.common.GenericPropertiesUtil.loadProperties(GenericPropertiesUtil.java:16)
  + **Solution:** This was due to trying to read off of property files that are no longer being used or can be found. By moving these to MagiProperties and calling that instead, it was able to find the flags or values it needed
    - **Note:** This needs to be continued to roll into Magi, as there are still many more being called. Properties should not require a DB/Property file/class loading.
* **Problem:** Unable to enable security, and allow finding of the userDetailsServiceImpl
  + **Solution:** Configured security setup in SecurityConfig.
    - Also modified login.jsp to call a LoginController which can call the correctly configured Spring Security userDetailsService
* **Problem:** Forwarding off of login was failing silently
  + FINE: Forwarding to [/WEB-INF/view/jsp/index.jsp] Oct 09, 2019 1:56:35 PM org.springframework.web.servlet.FrameworkServlet logResult FINE: Completed 200 OK
  + **Solution:** Moving the required files under */WEB-INF/view/jsp* helps for Spring to find them
* **Problem:** NPE .NullPointerException at com.rhdjapan.magi.services.service.user.UserServiceImpl.getUser(UserServiceImpl.java:60) at org.apache.jsp.dashboard\_005fview.reminder\_jsp.\_jspService(reminder\_jsp.java:210)
  + **Solution:** This was pointing at the “bad” entity manager (the one being pointed to by many services atm). Moving it to point to the right service allows the call to work correctly.
    - **This is likely the next step to get things working correctly**

### **Issues fixed/addressed since Version 1.0**

* Removed dependency off Glassfish entirely
  + Can now use Tomcat instead and let SpringBoot handle Magi startup
  + This will necessitate the removal of some files and properties that are Glassfish dependencies
* With Reminder, have a POC for Application Stack
  + JSP calls service, which is a simplistic implementation of Generic Service and DAO which is hooked immediately by Spring and Hibernate
  + Allows for other services to follow a similar suit
  + Allows for the extension of using Spring Data built-in features, Hibernate Criteria
* Cleared out/reduced dependencies on property files
  + [Build|Env|Global|Generic]PropertiesUtil can slowly be phased out, as we can let Spring handle different profiles and data via **application.properties**

### **Issues found/still open**

* **DAO is likely broken**
  + The entity manager in the various services do not seem to be working correctly now. This is likely due to having prior hookups handled by the XML files (which are now obsolete)
  + The services will need to be hooked into the proper service
* **JSP servlets will need to be corrected**
  + Locations for SpringBoot to find the JSP files will need to be moved around, so rendering can be done.
  + Right now , JSP files are calling services directly. We will need to have them call controllers via API calls down the road
    - This will allow us to down the road, to separate the UI from the backend, and allow us to move off of JSP (React, pure JS, etc)
    - **Note: the hooking of JSPs into controllers is a long-term goal, likely not possible until other areas are done**
* **Continue services move to “common” pattern**
  + Reminder and User are currently using the MVC pattern we want to encourage. Going forward we will need to slowly move the services to the same pattern, allowing for a huge reduction in specified code
* **SSL/ActiveMQ configuration needs to be figured out**
  + SSL settings have been added, but once they are on, Tomcat fails to start. The root cause is “Port is in use”. Need to figure out how to resolve this
* **Moving JSPs to call Controllers, which means we will need a controller layer**
  + Possibly a new module to manage controllers and allow them to call the services respectively.
  + Controllers will need to return data the JSPs can use. Right now, the JSP files call the services directly, and use the raw Java objects. Per example in reminder.jsp

List<Reminder> list = reminder.getReminders();

%>

<div id="reminderContainer">

<% for (Reminder rem : list) {%>

<div class="reminder">

<span class="deadline">(<%=dateFormat.format(rem.getEndDate())%>)</span>

<span class="title"><%=rem.getTitle()%> :</span> <%=rem.getDescription()%>

</div>

<% } if (list.isEmpty()) { %>

<br/> <%}%>

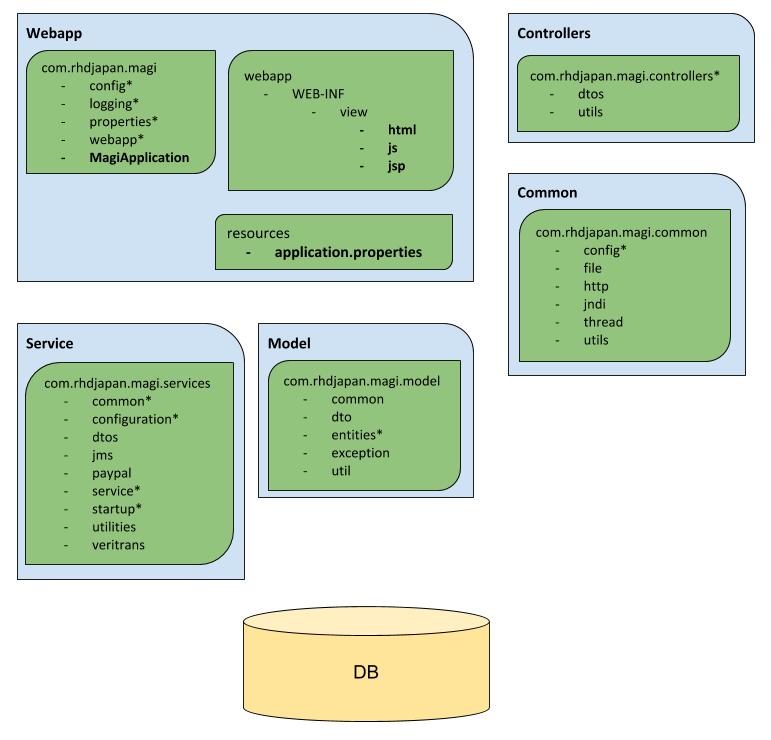
* [IO|RHD]Config files will need to be reduced
  + Called upon to handle debugging for Log4J, and will need to be hooked into a different area so that logging can continue to be done by Spring
* Reduce need for multiple Configurations
  + While we need profiles for local/dev/prod, configuration files/classes need to be reduced to a single spot

# **Next Steps for Magi**

### **Priorities (In descending order)**

1. **Fix DAO/Services**
   1. Move Services, one by one, under the same package where User and Reminder Services are already. Apply Generic[Service|DAO] to Services and allow Spring to handle DB
   2. Modify the current services if possible.
      1. Make use of Spring’s CRUD/JPA Repository interfaces, which have built-in features to allow for most standard operation.
   3. For services that need more “complex” queries, make use of Hibernate Criteria/Spring Data powers
2. **Move JSP files into locations preferable to SpringBoot, and ensure that any links are working**
3. **Hook RHD/IOConfig classes**
   1. Move IOConfig to possible service, and create common Log configuration/bean
   2. Move RHDConfid to possible service
4. **Move Configuration to Common**
5. **Move DTOs to appropriate place**
6. **Move (or Create) DAO objects**
   1. Hook DAO code to return DTOs to Services, Services can call down with DAOs
7. **Create Controllers Package**
   1. Create new module, allowing Magi to make use of controllers
   2. JSPs should call controllers and get the data back from them
   3. JSPs will need to move from Java objects to raw data (JSON/etc)

### **Magi Application after Next Sprint (Potential/WIP)**

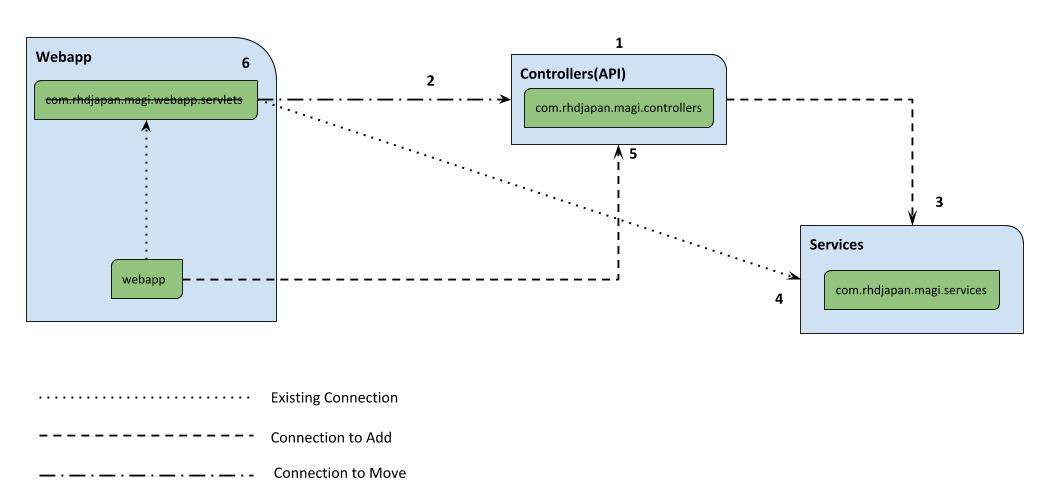


### **Concepts/Considerations**

* Controllers will need to be made, and tested to be call the services
  + This will allow us to determine Controllers are working
* Configuration and DTOs need to be consolidated to a primary location
  + DTOs
    - On the controller layer, it could make sense as to dictate what to send back to the caller
    - On the model layer, the data could be directly manipulated into DAO -> DTO/ DTO -> DAO. This would allow for everything **BUT** the model to be concerned with just the DTO
      * Model seems to be the strong contender
    - Need to also determine DTOs in the system and make sure proper DTOs are in the right spots (Service/Model Layers seem likely as the majority of data is currently there, but could be moved)
  + Configuration
    - Including \*Properties, along with \*Config class files for this consideration
    - Configurations should be in a single location, need to determine the best place
      * Proposal is to put it under *Common* module, to allow for them to be picked up by Spring, yet usable throughout the system.
* More common configuration and organization will likely be needed
  + Moving configuration, and moving dto files will help “clean” up organization
  + Common code for Logging, should be moved with configuration under *Common*
  + Common Utilities should be leveraged more, with the large swath of Utils under *Common* investigated and determine where to use them and to utilize more Java 8/Spring/etc.

# 

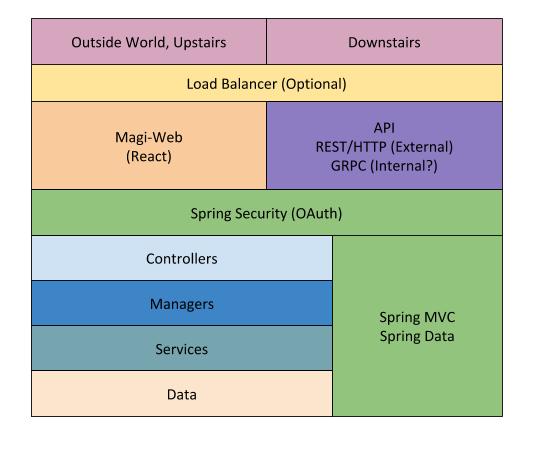
# **Migration to Removal of Servlets and use of Controllers**

****

**Breakdown**

1. Create a new module: Controllers (or to follow naming consistency, core-module-controllers)
   1. Should utilize SB and standard CRUD Interface, create an Abstract Controller to extend
2. Transfer Servlet logic to new Controllers
   1. To help with transitions it’s recommended to apply the [Strangler Pattern](https://docs.microsoft.com/en-us/azure/architecture/patterns/strangler)
   2. This should also allow us to split the work and to help spread the load out
   3. Reduces testing and Crucible patches
3. By transfer of logic, should also move any calls to the services appropriately.
4. Any and all other logical connections made from the servlets (or any packages left over) should be moved to call the controllers instead
   1. This will have to be a case-by-case basis, but my initial opinion is that any DTO/Utility logic can be moved to the controller layer.
5. Webapps should not have to do much in terms of change, but should now be pointing towards the APIs now in the controller layer
6. Remove the Servlets package once its become obsolete

# **Application Stack for Magi 2.0**

****

**Breakdown**

* Magi is used in 2 distinct different ways at the moment
  + HR/Upstairs uses it for mainly managing orders and users
  + Downstairs uses it for updating orders, making stickers
* A load balancer could be seen down the road for devhost1 but as of now feels completely optional
* Magi-Web
  + New “front-end” for Magi. Hoping to use React to create components and make use of modern framework for UI
* API
  + Used by the Magi-Web for communication to lower layers, as well as being able to be called by the downstairs that is using only a “part” of the web application
  + Could also be used for batch jobs or to signal them to be activated
* Spring Security
  + Adopts OAuth for modern security ideal
    - This might cause some security migrations that will need to be addressed (invalidating passwords for users, resetting logins, etc)
* Controllers
  + Core-module-controllers layer, holds the controllers listening for API calls
* Managers
  + Layer for logic managers
    - Managers can be useful for those operations that require multiple services to be called or used, and to help moderate operation order as well as make upper level decisions that lower levels shouldn’t concern themselves with
* Services
  + Core-module-services layer, holds the Service code
    - Business and lower level decisions should be made here
* Data
  + DB layer
    - DAO/Entity layer
* Spring MVC/Spring Data
  + Spring support for the back-end layers