

**Enterprise Architecture**

**Dreams/LILO System Assessment**

**Author :** Brian Wilson

**Created :** 7/10/13

**Last update:** 9/9/13 9:40 AM

**Table of Revisions**

|  |  |  |
| --- | --- | --- |
| **Revision Date** | **Author** | **Description** |
| 07-10-2013 | Brian Wilson | Initial draft for kickoff |
| 08-01-2013 | Brian Wilson | Template prepared |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |

#### Table of Contents

Introduction 7

Executive Summary 7

Recommendation Summary 7

Top Priority Action Item Recommendations 7

Assessment Framework 8

Principles 8

Principles Alignment 8

Team 10

Stakeholders 10

Assessment Resources 10

Deliverables 10

Objectives/Constraints 11

Objectives 11

Constraints/Out of Scope 11

High-Level Component Diagram 12

Base Concepts 13

JSF/RF Application Patterns 13

JSF Patterns (Java Server Faces) 13

RF (RichFaces) Patterns 13

Design Implications 13

Dreams Service Patterns 14

Security by Obscurity 14

Large Result Sets 14

Complex, Inefficient Queries 14

Physical Deployment Dependencies 14

LiloUI Service Interaction Patterns 14

Client Driven Service Invocation 14

Long Timeouts 14

RichFaces AJAX Tags 15

Database Access Patterns 15

Analysis 15

Future State Integration Patterns 16

Alternate Technology Platform 16

Refactored User Experience Design 16

Reimagining the Application Scope 16

Mobile First Design Approach 16

Recommendations 17

Technology Framework 17

Overview 17

Analysis 17

Recommendations 17

Upgrade to RichFaces 4.3 17

Implement technology relevancy processes 17

Replace JSF/RF Platform with a more stateless, lighter weight alternative 18

Performance and Stability 18

Analysis 18

Recommendations 18

Establish and Enforce Service Level Agreements 18

Perform analysis of the service invocation scheme 18

Utilize Resource Oriented Architecture for Static or Slow Churning Entities 18

Audit and Remove unnecessary RichFaces AJAX Tags 19

Audit and remove redundant AJAX calls for resources 19

Audit and specify timeouts for all AJAX interactions and SOAP service accesses to fail fast and disable retries after timeout errors 19

Specify SLA's for service interactions and set timeouts accordingly, disable retries for timeout failures 19

Assess Azul for sub-optimization impacts 19

Provide and utilize Service Health Check Endpoints 20

Refactor interactions against workflow resources 20

Utilize a search engine/index (Endeca, SOLR) for search use cases 20

Usability 21

Overview 21

Analysis 21

Recommendation 22

Involve the Usability Engineering Team throughout Development Efforts 22

Require Experienced UE Development Staff on the Delivery Team 22

Use CSS Styled Buttons instead of Graphics 22

Implement the Recommendations from the UE Study from 2012 and address all of the Currently Open UE Defects in the Application Defect Queue 22

Revise the Evaluation of Defects to Make UE Defects High Priority Fixes 23

Revise customer acceptance criteria and customer acceptance processes 23

Refactor workflows across domain/transactional boundaries 23

Scalability 24

Overview 24

Analysis 24

Recommendation 24

Utilize the Resource Oriented Architecture style where possible 24

Utilize WebSphere ExtremeScale to implement session replication 24

Upgrade the JSF version to 2.2 24

Refactor user interfaces using JSF "Stateless Views" to access more static resources 24

Renegotiate sessions on transaction boundaries 25

Continuous Integration 26

Overview 26

Subtopic 26

Analysis 26

Subtopic 26

Recommendation 26

Continuous Availability 27

Overview 27

Subtopic 27

Analysis 27

Subtopic 27

Recommendation 27

Establish smoke tests, configuration validation processes 27

Operational Isolation 28

Analysis 28

Physical Deployment Dependencies 28

Recommendation 28

Appropriately segregate service nodes (utility services, etc.) 28

Isolate, Relocate, Replicate, and Advertise Utility Services 28

Monitoring & Failure Analysis 29

Overview 29

Recommendation 29

Use Centralized Logging Service 29

Review monitoring processes 29

Test Driven Development 30

Overview 30

Analysis 30

Recommendation 30

Review service test cases for coverage and completeness 30

Enforce true test-driven development practices 30

Implement Peer to Peer Code Review Process 30

Services Security, Design, and Response Management 31

Analysis 31

Large Payloads 31

Recommendation 31

Paginate Results 31

Information Architecture 32

Analysis 32

Database Configuration 32

Custom SQL used in the ORM layer for performance purposes 32

Recommendation 33

Implement Active/Active Database Server Configuration 33

Implement Alternative Query Support for Expensive Operations 33

Appendix 34

JEE JSF Reference Architecture 34

Details of the 2012 LILO Usability Study 35

# Introduction

# Executive Summary

At the request of management team, this assessment was requested to determine areas of opportunity for improvement of the LiloUI user experience from both a performance and usability level. It is also intended to cover the Dreams/Lilo application from the perspectives of continuous availability, operational isolation, performance, and stability.

## Recommendation Summary

### Top Priority Action Item Recommendations

* Technology Framework
* Performance/Stability
* Usability
* Continuous Availability
* Operational Isolation
* Monitoring
* Configuration Management
* Test Driven Development
* Services Security, Design, and Response Management
* Information Architecture

# Assessment Framework

## Principles

The Enterprise Principles, as defined and approved the Architecture Review Board, were used as input in the assessment, in order to help determine compliance. The principles are approved by the Architecture Review Board.

The current approved set of Principles can be found here: <https://backlot.disney.com/groups/arb-working-group/projects/architecture-principles/content>

The principles fall into the following categories:

* Enterprise
* Business
* Information
* Solution
* Infrastructure and Security

### Principles Alignment

The following are the current Enterprise Architecture Principles, established by the Architectural Review Board. These principles help provide ‘guard rails’ for system implementations, to ensure consistent approaches to key architectural areas.

This assessment is based on LiloUI current state. Some of the following recommendations are intended to mitigate one or more alignment discrepancies.

| **WDP&R Architecture Principles** | **Name** | **Description Statement** |
| --- | --- | --- |
| **Enterprise Principles** | **Enterprise Architecture  Drives Alignment** | Not Compliant – This initiative is driven from a performance and stability perspective, not from an Enterprise Architecture point of view. |
| **Structured Investment Management Process** | Compliant - Uses structured processes to evaluate and approve investments, technology decisions and execution approach. |
| **Metrics & Measurements** | Not Compliant - Need to provide metrics and measurements for test coverage, code quality, performance, service level, reliability, and usability |
| **Business Principle** | **Business  Architecture** | Not Compliant – Business architecture assessment has not been done as of this writing |
| **Information Principles** | **Information is a Strategic Asset** | Compliant |
| **Compliance with Law** | Compliant |
| **Solution Principles** | **Globalization  Support** | Not Compliant – This is currently a domestic, Cast-facing, US system with minimal globalization requirements, but a primary deployment target is Disneyland Paris, which means that language, translation, and currency support must be provided. |
| **Interoperability** | Not Compliant - While in compliance in spirit, some of the Lilo interactions promote brittleness over interoperability (amount of information passed, etc.) |
| **Technology Independence** | Not Compliant - JSF/RF is a monolithic web framework with a large server side and network footprint that causes very tight coupling between the client and the server. Recommendations to alter workflows and page/session statement management will be complicated by the existence of this architecture. |
| **Scalability** | Not Compliant - JSF session state, at least as written, does not support appropriate scalability. Modern versions of JSF support better scalability, but hat a high refactoring cost. |
| **Auditability** | Not Compliant – Speaking from a pure application management perspective, tracing down and reproducing problems is an issue because of the way that information is logged and tracked. |
| **Configurability** | Not compliant – While Dreams and Lilo are configurable components, the solution design needs to take this configurability into account to manage the configuration deployment process to avoid system issues caused by delivering incorrect configuration information or delivering configuration information into the wrong context (production servers get referenced by test client installations, etc.) |
| **Infrastructure & Security Principles** | **Business Service Availability** | Compliant – While LiloUI components should be refactored to validate and take into account business service availability prior to use, enabling operational isolation. |
| **Security Controls** | Not Compliant – Dreams services are not protected by appropriate security controls. |
| **Modernized Technology** | Not Compliant – The versions of JSF and RichFaces in production are antiquated and do not scale well. These should at least be brought up to current versions. The JSF platform should be considered for replacement. |

## Team

### Stakeholders

This assessment was done at the behest of the LiloUI team in response to concerns about LiloUI performance and usability issues reported in the field.

### Assessment Resources

This assessment has been carried out with the cooperation of the following people and the utilization of the following resources.

|  |  |
| --- | --- |
| **Domain** | **Person or Resource** |
| **Enterprise Architecture** | Brian Wilson |
| **Business Architecture** | Domestic Sales/Fulfillment Standardization Architectural Approach Document |
| **Information Architecture/Data Modeling** | EJ Kraus |
| **Infrastructure Arch** | Manny Patel |
| **Solution Architecture** | Mangesh Patil, Haytham Abushaaban, Frank Blum, Jeff Braun |
| **Security** | Andy Scygiel |
| **Center of Excellence** |  |
| **Usability Engineering** | Jonathan Israel, Tiffany Adams |

## Deliverables

The deliverable for this assessment is this Word document.

# Objectives/Constraints

## Objectives

The following objectives were identified for this assessment:

* Evaluate the client application architecture from a performance and usability perspective
* Recommend short term actions that can be taken to improve the performance and usability of the system
* Recommend long term strategies that should be employed to resolve the performance and usability issues inherent in the system

## Constraints/Out of Scope

The following constraints for this assessment were identified:

* Not a holistic assessment  
    
  This document is not addressing systems outside of LiloUI, although some portions of the assessment delve into the Dreams service layer and the appropriateness of its design for LiloUI use cases.

# High-Level Component Diagram

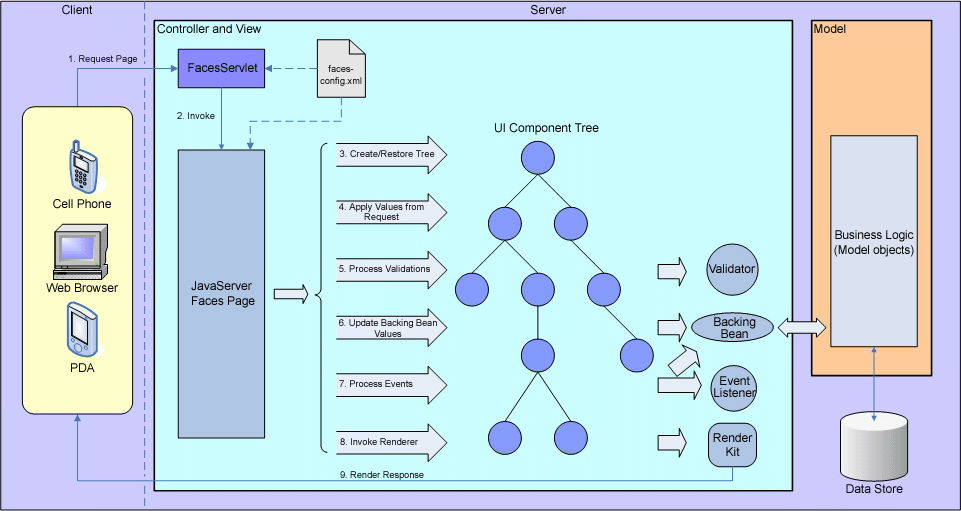
# 

Seen in context of the Sales & Fulfillment Composite, LiloUI provides the user interface component for the front desk and housekeeping functions at Disney resort hotels.

# Base Concepts

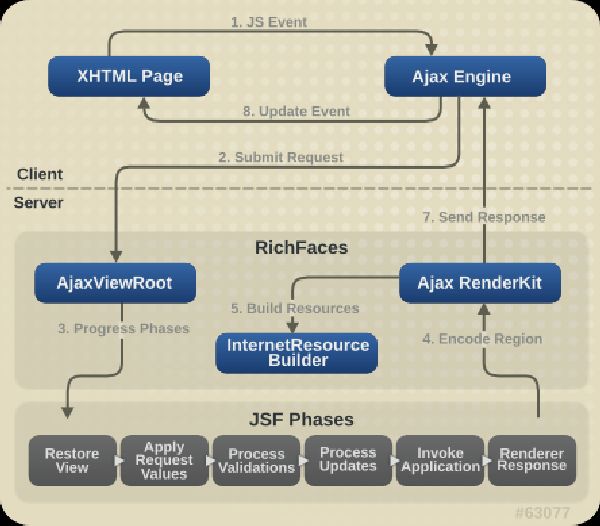
## JSF/RF Application Patterns

### JSF Patterns (Java Server Faces)

JSF is a Java-based web platform that implements the MVC (Model/View/Controller) paradigm completely on the server. It actually maintains a representation of the UI state in the server memory. This creates a high degree of server affinity so that a user must always return to the same server for the duration of the session. If a problem occurs on the server that a user is connected to, their work session is lost and they must begin anew with a fresh connection to another server. This hinders horizontal scalability unless some mechanism can be used to replicate the session state across nodes.

For ordinary web site interactions where a user logs in for a period of time, performs some tasks, and then leaves, the JSF model is workable. However, at the Walt Disney World front desk, the issue is exacerbated by the length of sessions that occur: An operator will sign in when starting their shift and maintain the same session over the course of several working hours. The session state is maintained on the server for this entire duration, often growing to many megabytes in size. This is largely due to the size of the results that are generated by Dreams web services, which then take up residence in the session objects and remain there for the duration.

### RF (RichFaces) Patterns

RichFaces is a Javascript client library that works with JSF to provide AJAX capabilities for the rendered web pages to interact with the server without actually reloading the browser page. It also provides a number of user interface control implementations that simplify the development of rich web-based user interfaces.

RichFaces can be used to populate and maintain the state of controls like lists, combo boxes, tables, and forms. However, the content that gets used by these controls also takes up space in the JSF session and remains there, even when the page that requested it has been unloaded from the browser.

The AJAX capabilities of RichFaces can make an application seem more responsive when used correctly, but if used incorrectly can create blocking issues and long, often unnecessary waits for the user. Indiscriminate pasting of RichFaces tags can cause expensive end runs to the server that may not be required.

### Design Implications

Because of the way that JSF and RichFaces work together, it appears that an "all in one" user interface approach was adopted so that the system state for the user would be held together for the duration of interaction. It does not appear that the end user roles were fully taken into account. Because of this, one page is expected to support multiple business functions. This causes unnecessary data to be populated in the user session, and then maintained over the course of the day. As we will see later, it also has implications on the user experience.

Many of the recommendations you will see in this assessment have to do with reducing the scope of interactions and the size of information payloads in order to reduce the size of the session, its lifetime, and its dedication to one particular access node.

## Dreams Service Patterns

### Security by Obscurity

Dreams services are presented with no security constraints. The only requirement to access Dreams services is for you to know the endpoint and the interface. This has been seen to cause issues as testing activities have been inappropriately pointed at production endpoints. Rather than being refused by the service provider, performance of the production endpoint has been affected.

Waiting on some input from Andy Scygiel to get his assessment of Dreams service security

### Large Result Sets

Dreams services often provide very large result sets, which cause a number of performance issues. The result sets are expensive to prepare and package, and they wind up taking up space in the JSF session. Since these are remote procedure calls, no benefit is derived from the HTTP interactions – the cost is the same for every subsequent request. Because the results are not paginated or otherwise constrained, the size of the session space maintained is the largest possible rather than the smallest reasonable. Server-side optimizations have been used to provide application cache, but in many cases a refactored resource structure would provide smaller, more granular access to entities, and provide them in a more stateless fashion that lets us take advantage of the caching capabilities built into a network-based application.

### Complex, Inefficient Queries

Core Lilo interactions include looking up room availability and searching for reservations. These queries run against a complex database structure, joining many tables and applying complex filtering criteria.

### Physical Deployment Dependencies

In some cases, Dreams services are incorrectly deployed. Utility services reside on application service instances, creating a runtime coupling between what should be unrelated components (for example, account payment services become unavailable when the booking service instance is down because utility services it depends on are deployed on the booking service instance).

## LiloUI Service Interaction Patterns

### Client Driven Service Invocation

Dreams service operations require the client to provide values that then determine what actual service operations are performed, or in what way they are performed. This forces an untoward amount of business logic onto the user interface tier.

### Long Timeouts

Dreams service requests are made with default timeouts (usually 60 seconds if they are not explicitly specified) or with excessively long timeouts (up to 300 seconds). This is because of the size and scope of the service operations themselves. The large result pattern indicates that appropriate service response time was not considered for the interaction, and the long timeout pattern indicates that the service granularity is too coarse. Extending the length of timeouts to accommodate slow services is considered an anti-pattern and should be avoided.

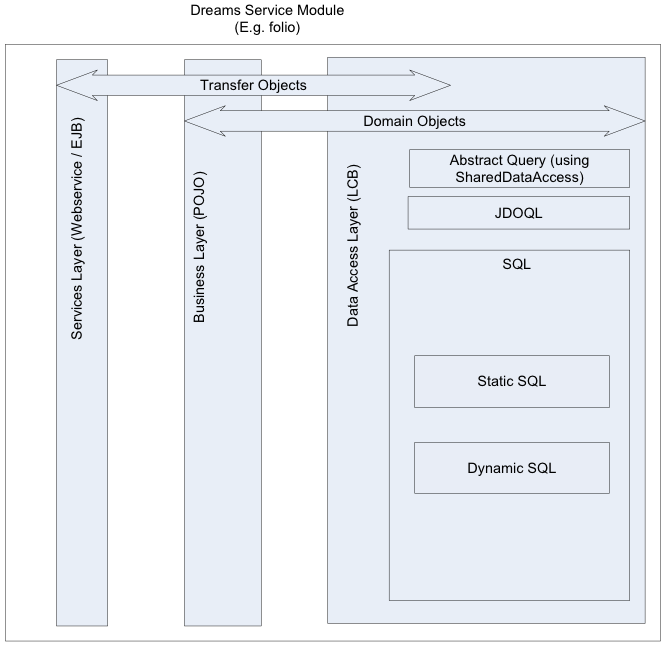
### RichFaces AJAX Tags

RichFaces provides support for AJAX Tags which can perform validation activities, etc., when the state of controls on the web page change. There are appropriate uses for these tags, but in LiloUI it appears these have been included where they shouldn't have been causing unnecessary trips to the server and contributing to perceived performance problems (that application blocks while this AJAX activity is occurring within the form).

## Database Access Patterns

### Analysis

This model from the Dreams documentation illustrates how Dreams services generally access the database using the Transfer Object design pattern:



In the case of room inventory management, in the interest of performance, custom SQL queries are being passed directly to the database through the ORM layer rather than being managed as domain objects. This indicates several issues:

* The database queries are probably too complex
* The database's indexes may not appropriately cover the queries being made
* The database may be too highly normalized for the queries to run efficiently
* The query patterns may not be filtering to a reasonable result set so the result set is too large

## Future State Integration Patterns

### Alternate Technology Platform

The desired future state includes the deprecation of JSF/RF in favor of a standardized web UI development platform that is consistent across the segment, as well as incorporating more of the Resource Oriented Architectural style.

### Refactored User Experience Design

Some of the recommendations in this document speak to refactoring or separating user interfaces into distinct interfaces and flows. For new development, utilizing the new stack is recommended. However, the integration between the new and old components must be considered.

### Reimagining the Application Scope

While it is recommended to address the usability issues in LiloUI today to provide a better user experience for the current user base, reimagining the application scope for DLR and DLP could be an appropriate way to reset the playing field and initiate the move to a common web application development platform without dragging along too much baggage.

### Mobile First Design Approach

For new development, the "Mobile First" design approach is recommended. However, we must be sure to approach this in the right fashion and apply the correct user interface and data access patterns. The current suite of web services is not designed correctly for this use case. The proper set of resources to support the required interactions must be identified, and those resources must be designed in such a way as to support the mobile paradigm appropriately.

# Recommendations

# Technology Framework

## Overview

LiloUI is a web application that runs on a highly stateful platform called Java Server Faces, or JSF. It is running on version 1.6, which was released in 2006. This is a very old version with some known limitations, not the least of which is the way that it handles sessions and retains session information. Long-lived sessions with a very large footprint have been an issue for Lilo UI from the beginning, so modernization is required, if not full out replacement.

## Analysis

The following recommendations are based upon two somewhat divergent objectives:

1. Reduce the pain and operational impact experienced by end users because of the current deficiencies in Dreams and Lilo UI
2. Recreate the property management system, in whole or in part, on a new technology platform that aligns with the segment strategy for rationalizing the number of technology platforms used and supported

These recommendations assume a hybrid approach where new components identified through refactoring are deployed against the newly identified technology platform to improve the system overall and evolve it onto the new platform, while addressing current pain points and taking advantages of the current technology platform through local modernization of it.

## Recommendations

### Upgrade to RichFaces 4.3

RichFaces offers many new capabilities that will be useful in improving the perceived performance of the LiloUI. These include:

* HTTP Server Push: As resources change on the back end, the updated information can be pushed to interested clients, removing the need for backhaul service requests to keep the UI up to date
* Long Poll: An variation on HTTP Server Push with similar benefits, but usable on older browsers that don't support HTML5 as well
* HTML5 Local Storage: This is a cache of information available to the browser, limited only by the size of the local computer's storage, that can be used to store large data sets for fast local access
* CSS…

Upgrading to the current version of RichFaces will address many of the usability and performance issues currently suffered by users of the LiloUI.

### Implement technology relevancy processes

Implementing a technology relevancy process will increase the ability of our development teams to appropriately adopt newer versions of technology as they become available, and to take advantage of the newer capabilities that those versions provide.

This means that we need to bring our technology standards documentation up to date, communicate these standards, and implement a process through which projects are periodically reviewed against the current standards to see if a modernization effort should initiated, based upon the lifecycle state of the component.

Implementing a technology relevancy process will decrease the likelihood that we will arrive at a similar position in the future, with a system that is hobbled by antiquated versions of software components. However, the outcome will also require us to mature our technology governance processes and emerging technology evaluation capabilities.

### Replace JSF/RF Platform with a more stateless, lighter weight alternative

This recommendation aligns with stated objectives to reduce the technology footprint of the segment and rationalize the number of development platforms that are utilized for various purposes. Given the limitations and constraints built into the JSF/RF platform, the long term recommendation is ultimately to move off of it and onto the common platform as identified by GBTS and our segment level partners.

The target platform for this application will be PHP with the Zend framework as has been utilized by WDPRO for MDX and by GBTS for Passport UI.

Redeploying LiloUI on an alternative platform will increase the scalability of the application and make many more development resources available for future enhancements, expansion, and upgrades. There is a cost to this, however, so an incremental move is recommended over a big bang approach.

# Performance and Stability

## Analysis

From an application server perspective, large data sets and large user sessions cause memory management issues and long garbage collection cycles that can stall out a server until garbage collection is completed. This is why the Azul appliance was instrumental in the original LiloUI release. Without it a large increase in server capacity was going to be required to sustain the expected number of users. However, the benefit of the Azul seems to have waned. It is unclear why these earlier benefits are no longer realized.

## Recommendations

### Establish and Enforce Service Level Agreements

Establish Service Level Agreements for application services and implement monitoring components that detect violations, raise alerts, and elicit responses from the support teams. The outcome from this will be faster detection of issues, faster root cause analysis, and ultimately quicker resolution of service performance issues.

Establish Service Level Agreements for client application performance and implement client monitoring components for Javascript and service response performance that detect violations, raise alerts, and elicit responses from the support teams. The outcome from this will be faster detection of issues, faster root cause analysis, and ultimately quicker resolution of service performance issues.

### Perform analysis of the service invocation scheme

Per the Dreams services interaction section above, the assumption is that correcting the interaction model between the application and the services layer could rectify many of the performance issues. This would require some analysis in order to identify and alter those interactions that would provide the most benefit. A key factor in this analysis is the service level agreements specified above. By measuring the frequency with which service calls are made against the frequency of SLA violation will lead us to candidates for refactoring.

The outcome of this activity would be a prioritized list of the most utilized but worst performing services, increasing our ability to concentrate sustainment efforts on refactoring those services that will have the most positive impact on the user experience.

### Utilize Resource Oriented Architecture for Static or Slow Churning Entities

While carte blanche refactoring of Dreams services and Lilo service interactions could be a lengthy and expensive proposition, many of the resources consumed by the runtime application could be provided as REST resources and consumed in concert with the SOAP services.

Utilizing ROA for lookup lists, resort details, etc., would increase the performance of the application from a user's perspective, while reducing the size of the session footprint at the server (the REST resources would be cached at the browser and the HTTP server rather than in the JSF session).

### Audit and Remove unnecessary RichFaces AJAX Tags

Indiscriminate pasting of RichFaces AJAX tags causes unnecessary server round trips for trivial operations such as updating checkboxes. The code should be scoured for these and any that are unnecessary should be removed. If any are allowed to remain, they should be refactored so that they operate asynchronously rather than causing the user interface to block while this processing occurs.

The outcome of this will be increased performance of the application by reducing the number of service calls being made and blocking further action, increased productivity on the part of the operators because they will be able to more quickly fill in the required information to complete transactions and reduce average handling time, and reduced load on the server because of fewer superfluous service requests. The client code will also be simplified.

### Audit and remove redundant AJAX calls for resources

There are locations within the code where lists are refreshed unnecessarily (for example, when selecting a reservation from the list of daily arrivals and marking it as arrived causes the entire list of arrivals to be refreshed). Based upon application interaction studies, these redundant calls should be identified and removed.

Removing these redundant calls will result in a more responsive application, improved user satisfaction with the system, increased effectiveness of the interface, and reduced average handling times at the front desk and other Guest interaction points.

### Audit and specify timeouts for all AJAX interactions and SOAP service accesses to fail fast and disable retries after timeout errors

The default timeout for many application platforms is 90 seconds. In some cases, Dreams/Lilo timeouts have been extended to be as long as 300 seconds. Service requests use server resources, and keep them tied up for the duration of the wait state. If there is a server issue and multiple clients are requesting services and claiming resources, and then all of them start waiting for up to 300 seconds, the odds of the server failing, paging, queuing, or otherwise degrading are vastly increased. Service clients should wait an appropriate length of them before abandoning the request and handling the exception appropriately.

Specifying timeouts and removing retries will increase application responsiveness, reveal issues earlier, reduce the load on IO bound servers that are trying to field these requests, and comply with generally accepted defensive programming practices.

### Specify SLA's for service interactions and set timeouts accordingly, disable retries for timeout failures

If an appropriate SLA has been set for an expected service response and a timeout has occurred, odds are good that trying again immediately will end the same way – with a timeout.

Setting appropriate timeouts based upon reasonable SLA's will increase the speed within which system issues can be identified, reduce the resources used by a server by terminating requests sooner, and reveal inefficient services and processes. Disabling retries also increases the velocity of issue discovery, as automatic retries can bind up a system with invisible redundant calls that block the user but do not advance the process.

### Assess Azul for sub-optimization impacts

When it was first deployed, Azul provided great economies for running the heavyweight LiloUI JSF sessions because it could manage larger session states and handle large garbage collection processes without blocking the server. Part of this benefit comes from the Azul appliance being aware of other processes running within its JVM and utilizing those to perform zero network latency activities. One reason that performance may have degraded may be that processes have been modified to perform additional or redundant activities that traverse network boundaries. Server processes and services that run in Azul should be evaluated to confirm that they are not exiting the Azul memory space to do their work, or that those accesses are minimized.

Assessing the Azul for sub-optimization impacts will result in the identification of errant processes and the opportunity to refactor and correct those processes, ultimately resulting in improved application performance and cleaner application logic processing.

### Provide and utilize Service Health Check Endpoints

A primary contributor to the perception of application instability is failed operations. A user commits a fair amount of effort to setting up a transaction only to have it fail with some kind of error message. This is exacerbated by the use of long timeouts because the client waits a very long time before giving up on an unavailable service, possibly even retrying under the covers, and ultimately failing. The LiloUI should be more proactive, keeping tabs on available servers or services and adjusting the available options based upon those statuses. Health check endpoints should be established on all service nodes that present current service availability status for that node or service entry point. The UI should poll these endpoints periodically during an interaction and disable UI capabilities that depend upon that service being available. This will save the Cast Member from investing too much time in a data collection activity that cannot ultimately be completed.

Utilizing health check endpoints will increase user satisfaction by preventing users from wasting their time on transactions that are ultimately destined to fail, increase application performance because only transactions that can be completed will be attempted, increase the visibility to server or service availability problems through the use of polling or dashboards against the endpoints, and decrease triage time and root cause analysis by providing a first place to look for potential problem sources.

### Refactor interactions against workflow resources

Today, when a LiloUI user logs in, they keep the same session throughout the course of their workday. The session state is maintained at the server during this entire time, and can grow to a significant size. All of the objects associated to each operation remain as part of the session, so even after a transaction such as a check-in has been completed, objects from that transaction remain in the session state.

Key interactions such as check-in and room assignment should be refactored so that they maintain the least amount of state required. For Online Check-in, this was accomplished using a workflow resource which carries references to the other affected entities. With this model, only the workflow resource needs to be maintained in the session state, and it can be passed to participating services who reference it to complete the transactions.

Redesigning interactions for statelessness and latency tolerance will decrease the amount of memory and computing power required on the server to maintain the user session and increase performance for both the client and the server. It will reduce the size and complexity of messages that must be transferred, and it will reduce the tight coupling between the client and the server by reducing the number of objects that must be maintained in the cache.

### Utilize a search engine/index (Endeca, SOLR) for search use cases

One of the primary points of dissatisfaction in LiloUI is in the response time for search queries. This is largely due to the architecture of the database and the size and complexity of queries that must be run to perform searches (strategies for improving database performance appear later in the document). The alternative is to create and maintain a faceted search index that supports the required search use cases in an efficient manner.

Utilizing a search index instead of direct database queries would vastly improve search performance and reduce database server load.

# Usability

## Overview

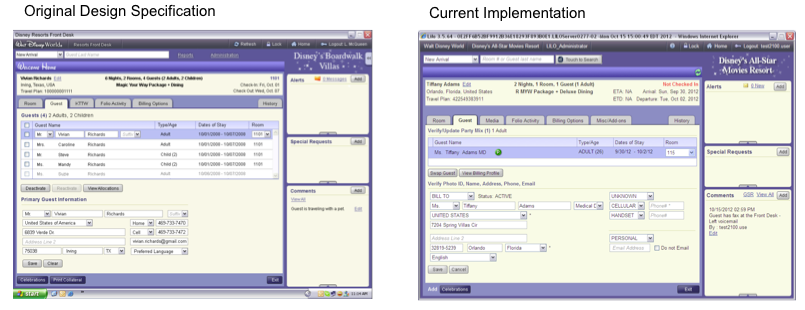
As requested by the Resort Line of Business in 2012, Usability Engineering (UE) was asked to provide an evaluation to capture Cast Member dissatisfaction and improve the efficiency and performance of LILO.

The majority of the issues and user inefficiencies are a result of not adhering to the design standards that the UE team provided in the beginning. Approximately 600 Usability defects were identified and logged against LILO prior to go-live. These defects were later marked as “cosmetic” and remain logged and unfixed to this day.

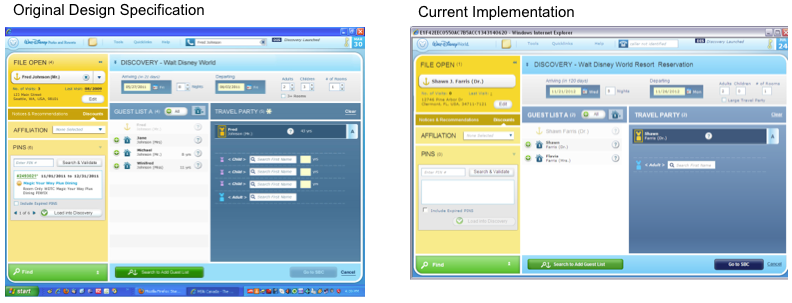
Much work has been done in improving system performance; however there are opportunities for significant gains to productivity through improved usability and efficiency.

## Analysis

In the comparison below, you can see how the LiloUI looked in the original design specification as compared to what was actually delivered. One of the reasons for this is that the UE team was not involved during the development process. The design was handed off, and the developers provided their own interpretation in the result. Many of the existing UE defects currently logged would have been avoided if the specification had been followed.



Here we see another example where the Passport team implemented against the UE specification. However, in this case, the UE team was involved in weekly sessions throughout the development process and were thus able to guide the development toward a complete and usable user interface.



In addition, long waits, a pesky "processing" display, and poor support for keyboard controls delay and frustrate end users, adding to the average handling time for Guest check in and related interactions. Creative front line Cast Members have devised workarounds, but these add complexity, time, and frustration to their daily tasks.

## Recommendation

### Involve the Usability Engineering Team throughout Development Efforts

This is a key recommendation from this effort.

Involving the UE team throughout the development process will increase the quality of the end user experience, reduce the number of user interface bugs reported, and increase the velocity of delivery of the UI.

### Require Experienced UE Development Staff on the Delivery Team

One issue noted with the LiloUI implementation pattern was that the technical staff was deficient in the area of user experience development. They clearly attempted to implement the interface as designed, but the mistakes and omissions noted would most likely have been addressed early on by experienced user experience development resources.

Staffing the team with talented user experience development resources will increase the quality of the delivered UI, increase the velocity with which the UI can be delivered, and improve the quality of communication between the UE and delivery teams via common vocabulary, etc.

### Use CSS Styled Buttons instead of Graphics

In an attempt to replicate the user interface definition, custom image files were created to represent each button and each of its possible states. This becomes a maintenance nightmare, and will only get worse with the inclusion of Disneyland Paris and the internationalization of the UI that will be required.

Using CSS styled buttons will simplify application development and sustainment, reduce the number of image assets that must be maintained, and improve and standardize the appearance of the UI.

### Implement the Recommendations from the UE Study from 2012 and address all of the Currently Open UE Defects in the Application Defect Queue

This recommendation might seem overly broad, but user defects noted in a study that is over a year old should never have remained opened and unaddressed this long.

Addressing all of the currently open defects will increase end user satisfaction with the application and its perceived performance. The users will feel as if their concerns have been heard and rectified, and the relationship between delivery and the end user will be improved.

### Revise the Evaluation of Defects to Make UE Defects High Priority Fixes

It appears that the priority of user experience defects can be arbitrarily changed by the delivery team, flagged as "cosmetic" and shoved to the back of the queue to languish, literally, for years. It is an indicator that there be low pride in ownership for the user interface code and its assets. This is one symptom that may be treated by augmenting staff with UE developers as mentioned above, but ultimately the process for evaluating and prioritizing defects must be repaired.

Revising the process for evaluating defects will decrease the number of defects languishing in the production queue, increase the level of partnership between the business and delivery teams as the front end users see their concerns being taken seriously. It will also turn up underlying bugs more quickly, as a number of UI defects are simply the visual manifestation of underlying service interaction or application logic problems.

### Revise customer acceptance criteria and customer acceptance processes

In reviewing the performance and usability issues present in the Dreams/Lilo system, it becomes apparent that sub-optimal solutions are being approved for delivery. Automated testing for ordinary and expected behaviors such as tab order and form submission based on keyboard input is not being done, or not being done to the right level. Tests should be in place to confirm that the expected user interface actions are present and supported, and user acceptance testing should represent that the users are actually satisfied with the result.

If user interface issues are being unnecessarily deferred then it is hard to imagine that the end users are truly satisfied with the results they are getting. How are these deliverables making it into production only to be received so poorly?

Revising the customer acceptance criteria and customer acceptance processes will increase the amount of participation that end users have in the development process, increase user satisfaction with the completed system, reduce the number of defects that get into the production system, and increase the number issues having to do with the user experience that get resolved in a timely fashion.

### Refactor workflows across domain/transactional boundaries

The current Lilo user interfaces are expected to be used by differing groups of users (Financial, Front Desk), but the UI's are not customized to support the different views that these varying roles would expect to see. One side effect of this is that unnecessary information gets included in queries and in the stored session, depending on the use case (a financial user might not need arrival information, but it still gets fetched and stored in the session). It is recommended that these user interfaces be refactored to support discrete roles and role activities across domain and transactional boundaries.

Refactoring these user interfaces will simplify and increase the effectiveness of the UI's, increase the effectiveness of the users of these UI's, reduce handling time and confusion for key transactions such as resort check-in. It will also improve the velocity of development as changes can be localized to discrete user groups or activities as opposed to making changes that affect broad groups of application clients.

# Scalability

## Overview

In order to provide better application performance, horizontal scalability is desired. We should be able to deploy the LiloUI server components across any number of application server nodes, adding or removing them as required. A user should be able to connect to any server in the farm and be able to process or continue to process the transaction at hand.

## Analysis

LiloUI does not scale well because of the way that JSF works with its heavy server-based session state. This creates a high degree of server affinity, meaning that a user must always return to the same server for subsequent requests once logged in. An outage on the connected server will cause a user to have to reconnect to an alternative server, logging in again, losing any work that was resident in the session state on the original server. This happens to users today.

## Recommendation

### Utilize the Resource Oriented Architecture style where possible

By using the REST style to achieve stateless server interactions where possible, application services can become more stateless. With REST the state travels with the resources rather than having to be maintained at the server, so any node can service a request.

Utilizing ROA reduces session affinity of the application and the session footprint required by the application.

### Utilize WebSphere ExtremeScale to implement session replication

One approach being persued by the team is the use IBM's ExtremeScale technology, which will allow the session state to be seamlessly transferred from one node to another in the event of node failure. Session affinity is managed at the load balancer today. With ExtremeScale you can configure the load balancer to aim all of the traffic at one node in order to take another node offline temporarily, with ExtremeScale transparently refreshing the session on the target node transparently to the user.

For long term success, the application should be refactored so that session affinity and statefulness are minimized or removed. By employing WebSphere ExtremeScale, the application can be made more resilient. ExtremeScale maintains a cache of session information which, when one application node fails and a user is connected to a new one, retrieve the user's existing session state and rehydrate it on the new node.

Utilizing ExtremeScale will reduce, but not eliminate, the negative user impact of server affinity, increasing user satisfaction.

Note that ExtremeScale is not panacea. It is still recommended that we take all possible measures to reduce the size of the session state in order to make ExtremeScale failover processing as fast and as streamlined as possible.

### Upgrade the JSF version to 2.2

As mentioned before, there are many advantages to upgrading to the current release of JSF, not the least of which are the scalability benefits that can be realized. JSF 2.2 offers new capabilities for managing session state and allow for appropriate session size and duration management, as well as taking advantage of more resource oriented approaches to often used but slowly changing data sources.

### Refactor user interfaces using JSF "Stateless Views" to access more static resources

The new version of JSF offers a features known as "Stateless Views". These are JSF views that do not take up space in the user session. This is useful for more static resource (lists of resort names, etc.) that can be accessed without adding them to the server session storage.

Using Stateless Views would reduce the amount of memory used by user sessions, increase performance by utilizing HTTP caching mechanisms for fetching static information resources. It would also make the application more modular, allowing more resources to be reused in additional contexts.

### Renegotiate sessions on transaction boundaries

A recurring theme for LiloUI is that sessions are long lived throughout the workday of a typical user, requiring large amounts of memory to maintain on the server. This is because the session is opened and then kept open until the user eventually logs out, often hours later. There are logical breakpoints within the usage pattern of resort cast member where the session could be closed and reopened – right after a Guest has completed their check-in, for example. Closing the session and starting a new one would release the session to the garbage collector, reclaiming resources and freeing the server for additional requests.

One way to approach this is to use JSF 2.2 "Faces Flows" to provide more self-contained and tightly scoped application interactions which maintain a smaller session scope for a shorter amount of time and releasing resources earlier.

Renegotiating sessions against transaction boundaries would reduce the load on the application server, reduce the size of user sessions, increase the number of user sessions that each application server could support, and increase performance by making more resources available for processing.

# Continuous Integration and Availability

## Recommendation

**Implement a Governed Continuous Integration Platform**

Establish build/dependency system (Maven, although Gradle should be considered)

1. Setup common artifact repositories
   1. Set up SCM -Move all source artifacts for targeted app(s) (code files, rules, property files, etc.)
   2. Configure Nexus with internal reposiotry to control dependency downloads
2. Build and store ‘common’ IDE artifact (e.g., STS) with base configuration (includes plugins for compliance tools in CI that will run as part of IDE) for doing CI on you desktop, like compliance plug-ins, etc.) in Nexus
3. Configure and use Maven
   1. Define shared maven setting.xml file
   2. Define parent pom(s) for ‘like’ (same stack/dev environemtn) projects
      1. Includes ‘shared’ libs, etc,. e.g., tomcat, CXF, Spring items, etc.
      2. Use Maven Enforcer plugin to avoid over-rides in project poms
      3. Define standard compliance plugins, standard acceptance levels for each
4. Maven-ize the target app(s)
   * 1. Define project pom per app
        1. Reference parent pom
        2. Identify project-only dependencies
        3. Define overirdes of parent (only if Necessary!!)
        4. Refine compliance plug-in rules -Start with ‘Guilty till proven innocent approach’, ease as necessary
   1. Refine compliance rules as apps use CI and do builds
5. Establish manual review standards for project poms to spot ‘rogue’ overrides, un-authorized libs, etc.

The implementation of a governed continuous integration platform will increase the quality of the code that is released into production, reduce the number of bad coding practices employed by teams, and increase the repeatability of deployment with less manual planning and intervention.

### Establish smoke tests and configuration validation processes

Many of the stability issues seen with LiloUI and Dreams have to do with server configurations being incorrect and servers being unavailable.

Dreams applications servers and client UI's should be subjected to smoke tests and configuration validation processes.

Establishing smoke tests for server and client configurations will reduce the occurrence of misconfigured applications being activated in production.

# Operational Isolation

## Analysis

### Physical Deployment Dependencies

The Dreams service oriented architecture provides the ability to deploy application components as instances, logically and physically segregated in such a way that they can run independently of one another. This strategy has been utilized for capabilities like Account Payment Services, where touch to pay is made available on a separate instance from the Dreams Res functionality used at the front desk. However, in practice, it appears that not all of the components have been identified and segregated appropriately, creating dependencies between the instances. In today's environment, a Dreams Reservations outage will cause an Account Payment Services outage because Account Payment Services depends on utility service components that are deployed on the Dreams Reservations instance. This also creates a startup dependency as the Account Payment Service has to be brought up later in the sequence than the Dreams Reservation services.

## Recommendation

### Appropriately segregate service nodes (utility services, etc.)

Conduct a service dependency analysis and determine which service components are being used by which application functionality instances. From this analysis, find those common dependent components and move them to shared external nodes, preferably multiple or dynamically allocated instances, so that no one node is absolutely dependent on any other node.

Appropriately segregating service will increase uptime for application components, increase the independence of system components, simplify and shorten the length of deployments and deployment outage windows, and simplify restart sequences.

### Isolate, Relocate, Replicate, and Advertise Utility Services

Utility Services, by definition, should be stateless and standalone, useless regardless of the deployment location or context in which they run. The service invocation patterns for each Dreams module should be analyzed to identify those utility services that are being utilized by each component. The identified Utility Services should be made available and advertised in the ESB service registry and consumers should be updated to consume these Utility Services against an ESB endpoint.

The outcome of this will be increased uptime for all Dreams components, which will be able to run in a more independent fashion; increased operational efficiency because Dreams instances will be able to be restarted in arbitrary sequences; reduced outage times because outages will be localized to affected instances, and; increased availability of Utility Services for all consumers.

It would be good if you could name the specific services – can George get that to you?

# Monitoring & Failure Analysis

## Overview

Issues with Dreams/Lilo monitoring and measuring make it difficult to assess performance in a predictable way (measurements taken in QA environments do not foreshadow the actual performance that will be seen in production), so the impact of improvements is difficult to correlate.

## Recommendation

### Use Centralized Logging Service

DTSS offers a centralized logging service that stores normalized log entries into a Hadoop store for querying later. NGE has implemented an interception-based centralized logging service that performs boundary logging based on interception patterns and automatically logs to the DTSS CLS.

Utilizing interception-based logging and the CLS will increase application performance because logging activities will be handled out of band (the application does not have to wait while log records are formatted and written), reduce disk space utilization on the application server (logs go to a remote location), reduce the monitoring requirement (disk space shortages become less of an issue), and enable more efficient triage and root cause analysis because the logs will be in one place, in a standardized format, and can be queried based upon conversation ID's and other keys that can span transactional or session boundaries.

### Review monitoring processes

While appropriate tools are being used for monitoring the JVM and server processes, the business process of system monitoring may not be defined correctly to support speedy identification and resolution of performance or system constraint issues.

From the client perspective, it appears that no Javascript performance monitoring is being performed, not even in QA environments. A tool should be identified and implemented and folded into the performance evaluation process.

The entire suite of monitoring tools and processes should be evaluated and weighed against the set of exceptions that is being tracked. By changing the mix of monitoring tools, changing the way that they are used and how alerts and notifications are targeted, the precursor conditions that cause the bulk of issues and outages should be made more predictable and preventable.

Reviewing the monitoring process will increase the value of the tooling that we do have by focusing the results on the problem spaces that are currently active; let us identify new tools and targets for monitoring that will increase our ability to diagnose and correct issues and measure performance; and streamline the amount and types of monitoring required to achieve the intended results in runtime performance, triage, and defect correction.

# Test Driven Development

## Overview

A prevailing theme for Dreams and LiloUI is around the difficulty of testing and managing deployed releases. One way to mitigate this is to employ test driven development practices so that full regressions can be performed against altered code bases.

## Recommendation

**Review UI test cases for coverage & completeness**

While some degree of testing is certainly occurring, the repeatability and relevance of the testing has been called into question. This is an indicator that the type of testing or the test coverage is inadequate or incomplete. There should be test use cases for all of the key features of the LiloUI, and as defects are logged against the UI, new test cases should be added to ascertain that the fixes have been made and that they don't accidentally become unmade in a subsequent update. The current set of UI test cases should be reviewed for coverage and completeness to confirm that, once a round of testing is done, users can feel confident that the application will work as expected.

Reviewing the UI test cases for coverage and completeness will increase the quality of the test suite, and identify areas of poor test coverage.

**Use automated tools to perform UI testing**

Manual testing and test scripts are useful tactics for supporting code quality, and they can be very subjective and affected by various external forces. Testing tools should be used to automate the UI testing process. A full suite of UI tests should be defined that covers the primary use cases, and then tools like Selenium should be used to build and automate these tests. We should be able to run a full regression against the UI whenever changes are made. If the tests fail they should reveal areas that need correction, or areas where the usage model has changed significantly, requiring an updated suite of tests.

Using automated tools to do UI testing will increase the quality of the application that goes out to end users, reduce the number of defects that are released to production, increase the speed with which testing can be completed, and increase the quality of UI testing.

### Review service test cases for coverage and completeness

Like the UI, the services that drive Lilo need to be appropriately tested. Test driven development practices are not in use for Dreams services, but they should be. Full regression testing should be possible against the Dreams service base prior to any planned deployment.

Reviewing the service test cases for coverage and completeness will reveal areas where testing is inadequate or missing, and give us a starting inventory for the next recommendation, which is to implement true test-driven development practices.

### Enforce true test-driven development practices

With test driven development, the tests are written first so that they fail, and then code is written to make the test pass. Rerunning these tests as part of regression will reveal when code changes have inadvertently broken existing code.

The following steps are recommended to establish a test driven development environment for Dreams and LiloUI:

1. Define guidelines that identify class ‘types’ to be unit tested (i.e., define exceptions for getters/setter, generated classes, etc.)
2. Establish filters to exclude excepted classes from unit test compliance checking. This includes defining directory structures/packaging rules that facilitate such filtering
3. Generate skeleton unit tests for existing classes in targeted applications
4. Install compliance checking tools as part of the compilation process within the IDE
5. Refine filters, Identify target percentage of compliance
6. For new development, establish checks for the existence of unit test classes before business logic classes, breaking the build if a unit test for a given class is not found, even if overall compliance for the app exceeds the mandated coverage percentage.

Enforcing true test-driven development practices will increase the quality of the services and application code that is released into production, increase the quality of the code deployed, decrease the number of defects that are released, and increase the speed with which defects can be discovered and corrected. It will also enable full regression testing of the application UI and services when required prior to code deployment.

### Implement Peer to Peer Code Review Process

One of the biggest issues seen in the LiloUI code base is simply a cut and paste error. These sorts of mistakes can easily slip past the most carefully crafted set of tests and use cases, but can easily be spotted by another developer who is familiar with the code base.

WDPRO uses Fisheye and Crucible for peer to peer code review. It is recommended that GBTS adopt a similar process and set of tools to that which WDPRO uses.

http://wiki.wdpro.wdig.com/display/Tech/FishEye+and+Crucible

We also have opportunities to utilize the Sonar server to run code quality analysis against the source.

Implementing a peer to peer code review process will increase the quality of code that is deployed, reduce the number of defects that are allowed to be released into production, identify developers who may need additional coaching or education in certain development practices, and generally increase the level of visibility to changes in code across members of the team.

# Services Security, Design, and Response Management

## Analysis

### Large Payloads

One design pattern observed in the Dreams service layer is that very large data sets are returned as the response to a service call. In the LiloUI architecture, this contributes to and exacerbates perceived performance issues. Because the results are large, the queries take a long time to run, which causes clients to run with long wait times. When the result is processed by JSF, this large bulk of information must be parsed, transformed, stored, and maintained in the user session, contributing to the resource consumption and processing requirements on the application server. At the client layer, the old version of JSF in production uses tables for layout, which are notoriously slow when compared with modern CSS. Combine the use of slow table rendering with large result sets, often decorated with unnecessary column value GUIDs, and you wind up with a poorly performing user interface.

We also do not employ pagination for these large result sets. Because the queries are complex and inefficient, fetching a page worth of results is also slow, so users seem to prefer waiting for everything once rather than waiting over and over again for a smaller set of results. Likewise, rather than dynamically update a page of results, the implementation team has chosen to completely refresh the page rather than endure the slow redraw caused by the antiquated client stack.

## Recommendation

**Reduce data payload sizes**

These massive result sets indicate a parity mismatch between the use case and service granularity. The recommendation is to identify the worst offenders from a performance perspective (Daily Arrivals is a good example) and refactor or offer a new service capability that brings back the minimum viable set of information for that use case from a usage perspective. Some of this work is underway as part of the Services Evolution effort – we just need to prioritize that effort to address the most expensive and impactful service offerings first. The user interface will then need to be refactored to use the new result set – but should ultimately simplify the client interaction because the request will be more specific and the response more concise.

Reducing payload sizes will result in more discrete, specific service interfaces, reduced network traffic, reduced load on the application servers, improved accuracy of results (multiple, smaller requests as opposed to infrequent, all-inclusive requests provides fresher, more current information), reduced wait times at the client, and reduced client overhead for transformation and display of information, enabling the dynamic AJAX-style of user interaction rather than brute force page reloads.

It will also enable more capable and correct filtering of results. If a smaller, filtered set can be quickly retrieved, then users will be less likely to perform the lengthy "just give me everything" query that they tend to use today.

### Paginate Results

An appropriately streamlined result set may still be too large for a one page display in the browser. Pagination constraints should be applied to make sure that the client has as little work to do to render a useful UI as possible. Keeping a cache of the result set at the mediation layer will guarantee a speedy response as the user moves from page to page.

The outcome of paginating the results will be reduced page download times, more responsive web pages, and an overall better user experience.

### Service Security

As mentioned in the introduction, Dreams services today are mostly employing the "security by obscurity" approach. There really is no web services security in place. Pretty much anyone can access the services themselves. Database user accounts are used to control access somewhat, but there is no review process in place to affirm that these accesses are still appropriate after the initial configuration. The security team is able to use monitoring tools to determine if usage anomalies occur that might indicate misuse or abuse, but overall the control system around access is relatively lax, prompting the recommendations that follow.

Perform a review of the existing services and determine what user accounts are accessing the services and if those accounts are appropriate for access and removing those for which it is not. This is the spring-cleaning step. Taking this recommendation will reduce the number of active accounts with access to Dreams services, increasing the level of security around the composite. It will also reduce the number of accounts that need to be maintained for the database.

Where it's supportable, Dreams services should be modernized to support shared certificates and service endpoint security. Rather than attempt to boil the ocean, this work can be done as services are revisited for modification. This will increase the security around Dreams services and modernize services in the due process of maintenance.

Confirm that each environment is segregated so that there is no cross-pollination of data or access rights between them. Today the same ID's used to access development environments are also used to access production environments. This has led to issues where misconfigured development servers were able to update the production database, creating huge data integrity problems. Segregated environment permissions would increase the security of each environment involved and mitigate the possibility of configuration issues on development servers polluting production databases.

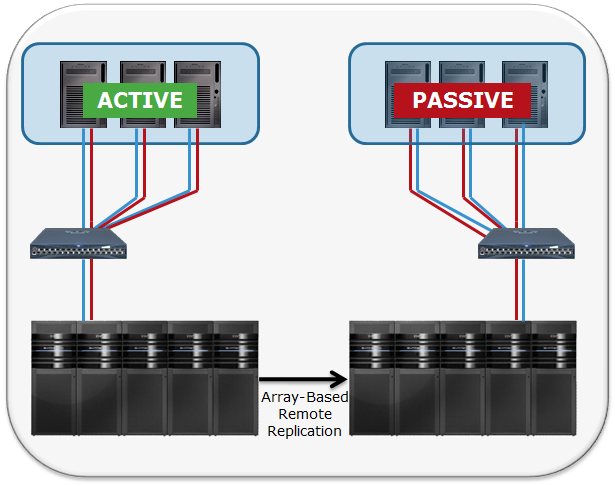
Review the submission and approval process for service access so that appropriate change management controls are applied and access to services is not inappropriately granted. This should also include periodic review of those rights and revocation when they have expired or their purpose has been fulfilled. Only the payment gateway is periodically reviewed at this time. The rest of the Dreams application services should be included in this periodic review. Doing so will reduce the number of superfluous accounts that are kept around, needing to be maintained, and increase the level of security that Dreams services enjoy.

# Information Architecture

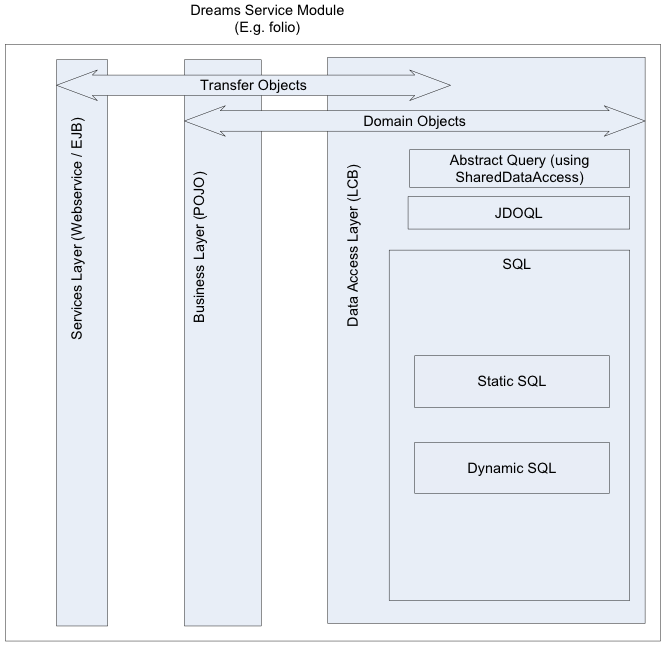
## Analysis

### Database Configuration

The Dreams database uses replication to provide fast backups, but does not use a fault tolerant database architecture to allow fast failover or database instance switching. The current implementation is *active-passive*. This means that some downtime is required to cut over in the case of a database issue. This creates a single point of a failure. If the active node fails, then the system is guaranteed to be down during the period of time it takes to make the passive backup into the primary.



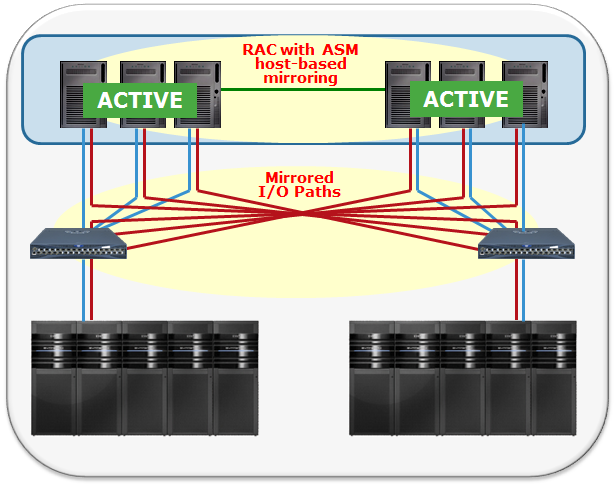
### Custom SQL used in the ORM layer for performance purposes

As implemented today, Dreams services interact with the database via an appropriate data mapping method for the most part. However, for key operations such as availability requests, custom SQL is injected into the ORM. This custom SQL is used for performance reasons, although the results are still being returned slowly. Not only are these queries complex and time consuming, they are run frequently by many operators, which means that every request can impact the application and database server.

## Recommendation

### Implement Active/Active Database Server Configuration

Dreams should implement an active-active database configuration for database replication.

The outcome of this will be increased application up time, improved data quality, and increased flexibility in managing deployments that make changes to the database. 

### Implement Alternative Query Support for Expensive Operations

There are a number of tactics that could be employed to simplify the database queries being submitted and to enable us to remove the custom SQL from the ORM requests.

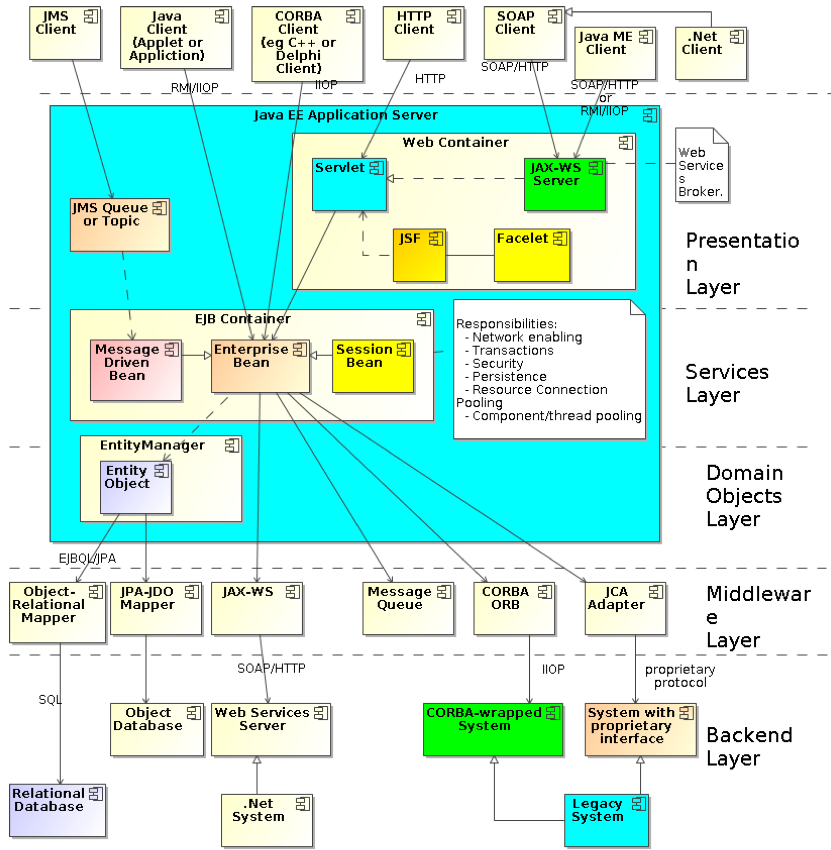
* Employ a distributed object cache – The company has done a DOC analysis and recommended Terracotta as the tool of choice. Some research would be required to understand how best to approach this.
* Use summary tables and EDA – By building summary tables that are designed to support the types of queries most commonly used, the queries and query speed can be optimized greatly. Using event driven architecture to push updates to the summary tables would support using multiple summary tables in disparate locations.
* Flatten the database structure – The third normal form is a fine target for an aesthetically pure database structure, but for a high performance data store denormalization is in order. If the tables can be flattened then the queries can be simplified, reducing the number of tables involved in the join and the number of indexes required to optimize the plan. Using realized views may also show improvements.

The outcome of using alternative query support capabilities will be faster application performance, faster query performance, less complex queries, fewer indexes, and reduced server load. It will also allow the standard ORM and abstraction layer to be utilized in a transparent fashion, reducing database affinity between the application and the information.

# Appendix

## JEE JSF Reference Architecture

Illustrates where JSF fits into the JEE reference architecture



## Details of the 2012 LILO Usability Study

**Usability**

**Overview**

As requested by the Resort Line of Business in 2012, Usability Engineering (UE) was asked to provide an evaluation to capture Cast Member dissatisfaction and improve the efficiency and performance of LILO.

The majority of the issues and user inefficiencies are a result of not adhering to the design standards UE provided pre-go-live. Approximately 600 Usability defects were identified and logged against LILO prior to go-live that were marked as “cosmetic.” These defects that were logged remain unfixed.

Much work has been done in improving system performance; however there are opportunities for significant gains for productivity through improved usability and efficiency. If leadership believes that the application will continue to evolve, then the recommendations below should be used as a guideline to improve the user experience.

**Purpose of the LILO Usability Evaluation**

The LILO User Interface Evaluation and improvement was an exercise that comprised audits and surveys in order to ensure continuous improvement. This exercise evaluated whether the objectives of the design were actually met. At times, there were unintended effects of the design and these must be addressed properly.

The Evaluation Objectives were to assess LILO’s success, collect end-user feedback for the application’s improvement, ensure end-users can accomplish typical tasks in LILO, and recommend UI changes or analysis that may be needed.

**Field Level Observations**

**Pre 3.X Releases**UE Job Shadowswere conducted to observe and document existing LILO functionality and user feedback

* March 7, 2012 & March 15, 2012 Pop Century (4-6 Cast Members)
* March 8, 2012 & March 14, 2012 Coronado Springs (4-6 Cast Members)
* March 9, 2012 Wilderness Lodge (4-6 Cast Members)
* August 28, 2012 Contemporary Resort (4 Cast Members)

**UE Observations of User Characteristics**

Front desk cashiers represent a broad demographic.Front desk cashier is considered an entry level position, and it can be full time, part time, and seasonal work. Ages can range from 17 to 70 (interns, professionals, retirees) although most are in their 20’s.

Front desk cashiers have varying levels of hotel/resort experience from novice to industry veterans. They also have varying levels of computer experience and proficiency. Most new cast members are at a novice level and have lower levels of computer experience and proficiency.

**UE Observations of Working Environment**

The lobby environment for each resort type varies. The noise and traffic levels at value resorts are high (they are noisy with a very busy feel) whereas Deluxe resorts seemed to have a slower pace.

In regards to guest requests and personal attention, Value resorts had a more “assembly line” feel (more functional) while Deluxe resorts had a higher level of personal attention.

There are also differences in personal vs. group reservation traffic; for example, Coronado Springs had a large level of group/convention traffic.

Each resort had a different level of back end support. FSA was either out front at a station or in the backroom. At busier resorts, the FSA could be tied up checking people in**.**

**UE Observations of Work Station Environment**

A typical work station at the resorts has the following characteristics:

There is a small desk area and the cast member works while standing. The monitor is low (the user looks down) and the keyboard sits on the desk at about 36”. The mouse is 12”-18” in behind the keyboard and does not fit straight to the side in some cases (the user has to reach slightly to use the mouse). Instead of a mouse, some resorts have Touchpad keyboards.

Stacks of park information and materials for distribution are typically assembled by cashiers. Some resorts have packets pre-assembled in the back room that cashiers have assembled during slow periods.

Cashiers are flanked by printer and encoder, and they also have a touch device at their work station. Cashiers are placed sporadically across front desk stations, and they also move away from their station at times to perform tasks such as call on a guest in line, check with the back office for help, or look for package or specialty group materials.

FSA support may or may not be physically present in the front**.**

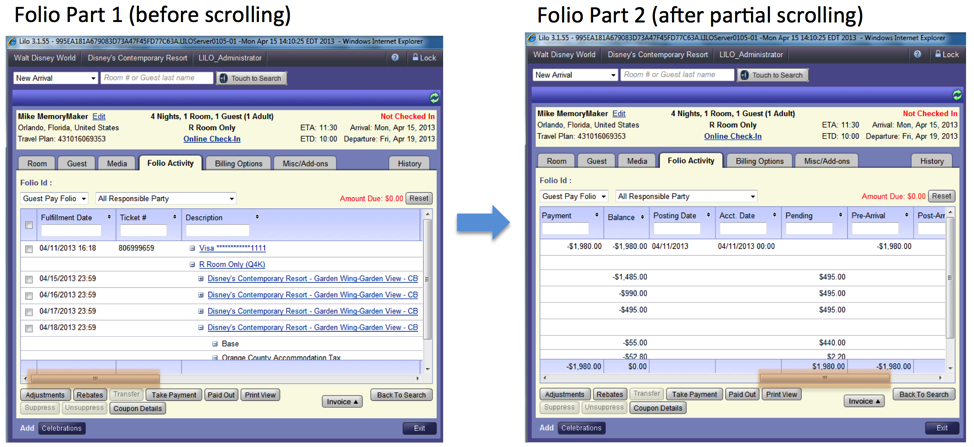
**The Data UE Captured**

Data was captured by observing LILO users, watching for areas where participants experienced delays and gathering information about participant reactions and observations.

Verbal feedback from participants regarding application performance was captured through articulations of any confusing, difficult, or positive experiences, answers to questions, and open discussion.

Introductory and survey information was also collected to learn demographic information and opinions about usability, performance, and general feel.

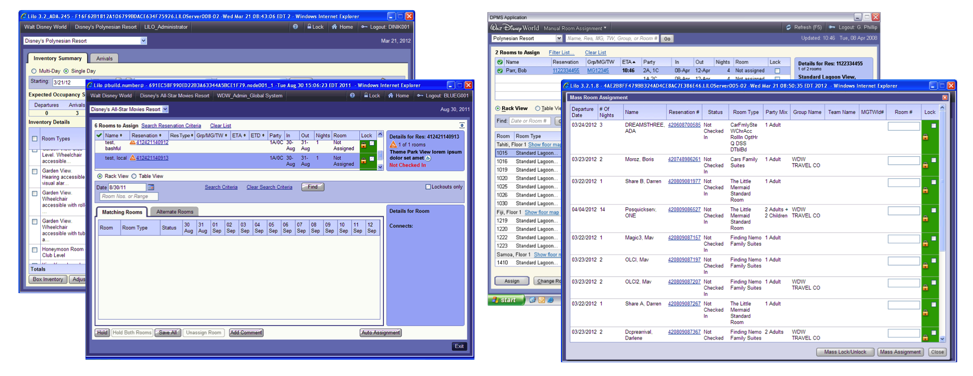
**Lilo Screens**

**Folio Screen  
**

**Identified Concerns**

Against UE recommendations, requirements were to create one folio view for multiple roles (Cashier and Finance). The Font Desk users are overwhelmed with unnecessary information. This causes a lot of horizontal scrolling, preventing the user from referencing the description. Often times the users print the folio to view it more easily and then go back into LILO to make adjustments. Also, excessive number buttons exist and are not grouped appropriately.

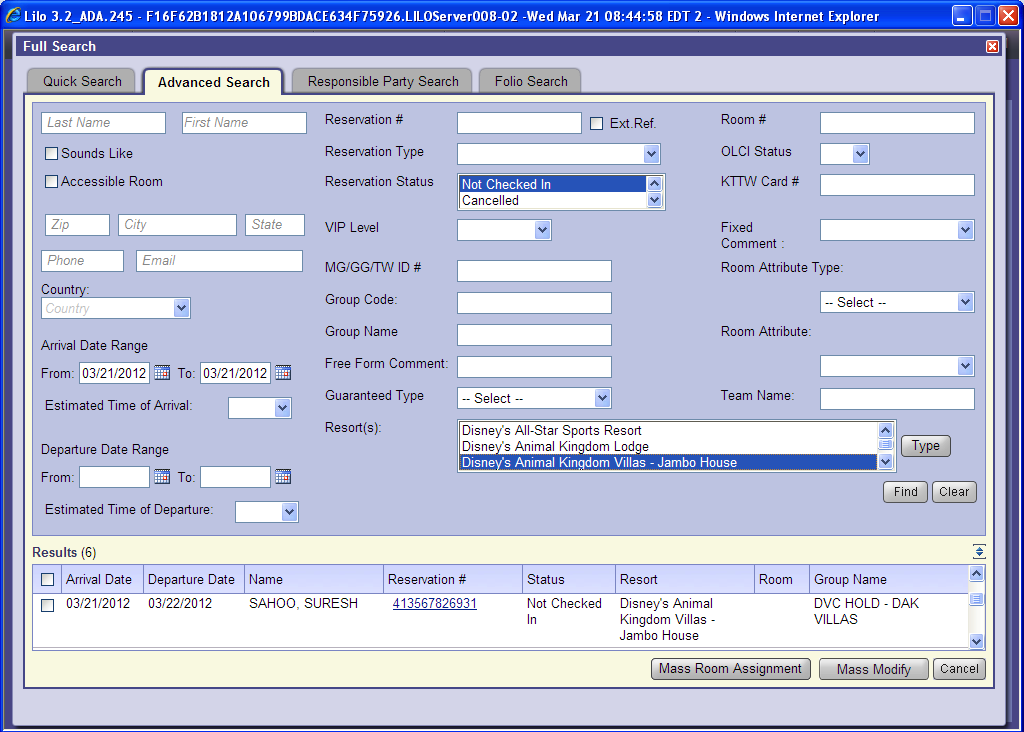
**Room Assignment Screen**



**Identified Concerns**

The original UE Specification included a single LILO Session displaying on dual monitors. The implementation did not support this functionality; therefore users are running multiple versions of LILO to work efficiently. Efficiency is greatly reduced as copy/paste and unnecessary searches are used frequently. The Sort/Filter functionality is not available on Mass Room Assignment, and there are performance issues with page loads.

**Full Search Screen**

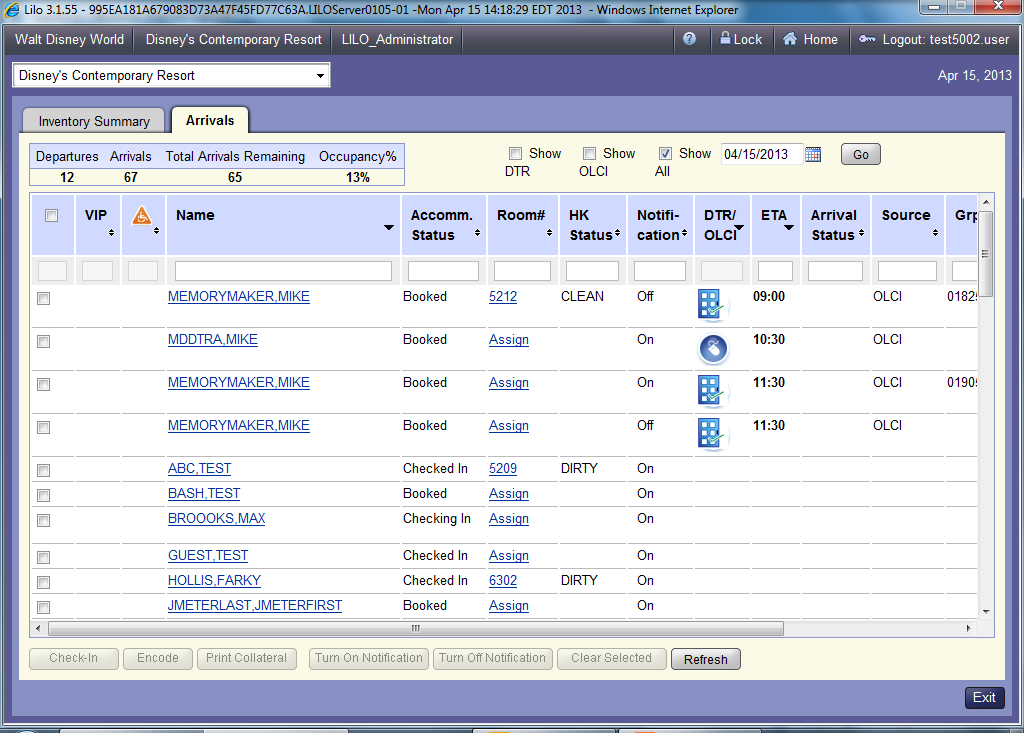


**Identified Concerns**

Acceptable search combinations are not identified or grouped and the user must learn through trial & error. As a result, extra calls are sent to the back end data bases, increasing load time and user frustration. Resort Names do not display in short name which makes the resort section more difficult to read. The use of the list box requires excessive scrolling and the user will only see one result before scrolling is needed; consequently, the user must scroll through the long results set or lose their search criteria by expanding the results set.

As with several other screens in LILO, the tab order is not correct.

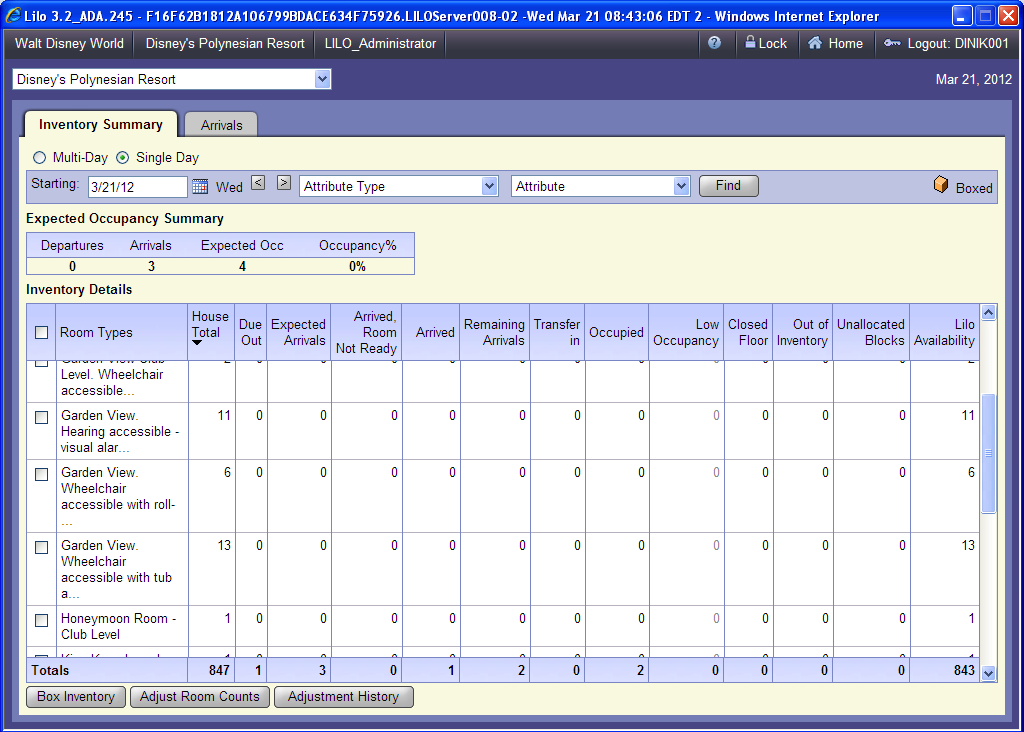
**Arrivals Screen**



**Identified Concerns**

There is unnecessary text wrapping on the Arrivals Screen and the headers are lost when scrolling. The checkboxes seem like a filter, but really a search is preformed again on “Go”. The actual filters are not user friendly. The Headers and Table function are inconsistent with those found elsewhere in LILO as is the refresh. The table needs to be adjusted to be easier to read; the presence of horizontal scrolling makes it difficult to read the table information.

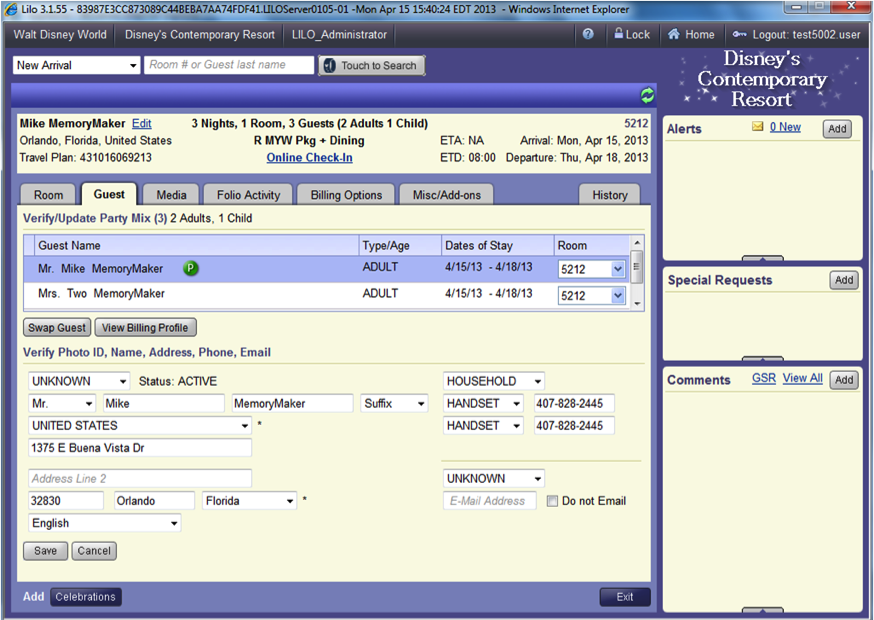
**Inventory Screen**



**Identified Concerns**

The titles of Search Criteria (Starting) do not make sense for a single day view. It is inconsistent in showing a key throughout the application. The headers are inconsistent with those found elsewhere in LILO, the Sort/Filter functionality is limited (search criteria only), and the order of room types could be simplified for easier reading.

**Guest Tab Screen**



**Identified Concerns**

There is inefficient use of the screen, disconnect and user confusion. The user is only able to view a max of 2 guests at a time causing unnecessary scrolling, loss of efficiency and heads down not engaging with guest. The address type is not located with the address and the address block is inconsistent. Also, the Single Phone Type appears for multiple phone numbers.

**Inefficiencies**

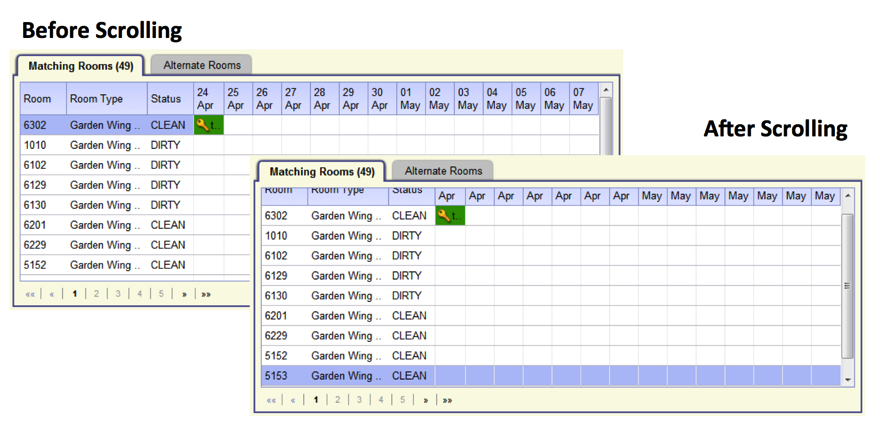
**Keyboard and Keyboard Shortcuts**



**Identified Concerns**

As observed at Disney’s Art of Animation Resort, the new Touchpad Keyboards are difficult to use versus a standard mouse. For example, the number pad on Touchpad Keyboards is not in the same position as typical keyboards. Users cannot use the “Enter” key to perform a default or selected function, and therefore some users only use the mouse or touchpad. There are also no keyboard shortcuts on the Touchpad Keyboard.

**Scrolling**

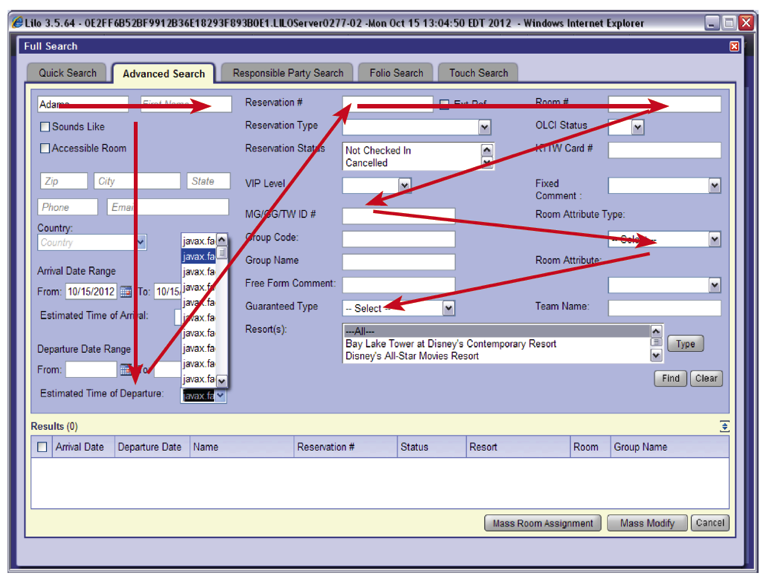


**Identified Concerns**

Scrolling is causing the Cast Members to be inefficient and slow at time critical tasks. The unnecessary large amount of scrolling is creating loss of context and slowing transactions. The scrolling is causing a “Heads down” usage, resulting in the user not interacting with the Guest. There is also horizontal scrolling throughout the application - studies show horizontal scrolling is extremely detrimental to time critical tasks. The Business was made aware during the design process that horizontal scrolling would be detrimental to the usability of the application. Alternative designs for the screens were suggested.

It is recommended to prevent the screen from scrolling both horizontally and vertically. The help fix the issue, navigation and action buttons should always be visible. Objects within the screen (lists, informational blocks) can scroll vertically, however in tabular displays, column headings should not scroll. There should be no horizontal scrolling.

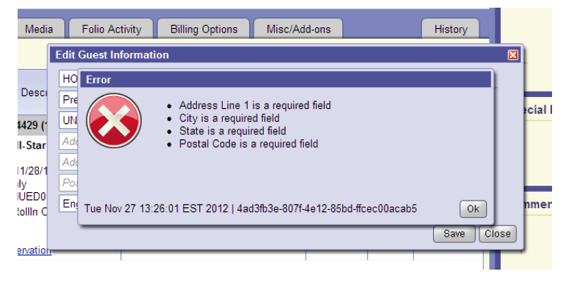
**Tab Order**



**Identified Concerns**

The global tab order is confusing in areas where keyboard access improves user efficiency.

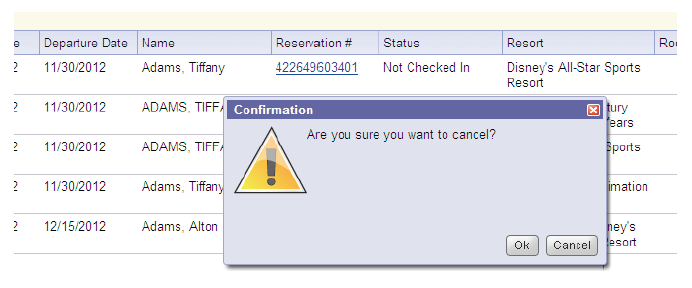
**UI Field Level Validation**



**Identified Concerns**

UI Field Level Validation does not exist; therefore, user feedback is not received until the user tries to move forward which is frustrating and inefficient. This also impacts performance because of the unnecessary calls to the service.

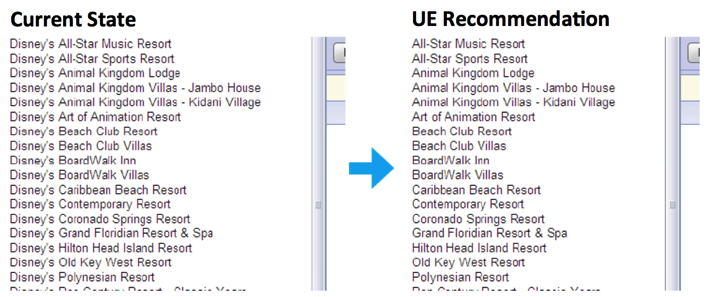
**Error Messages**



**Identified Concerns**

Because different individuals created the Error Messages, there is user confusion and inconsistencies around error message text and button options, as seen in the above example.

**Display Resort Short Name**

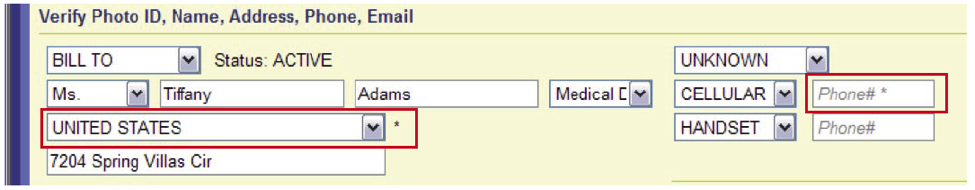


**Identified Concerns**

There is difficulty in identifying a particular resort when repetitive text is used.

**Inconsistencies**

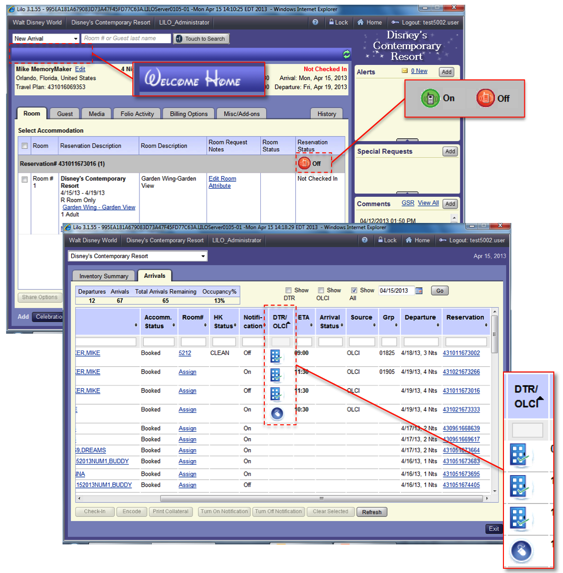
**Required Fields**



**Identified Concerns**

There is an inconsistent use of metaphor used to indicate required fields.

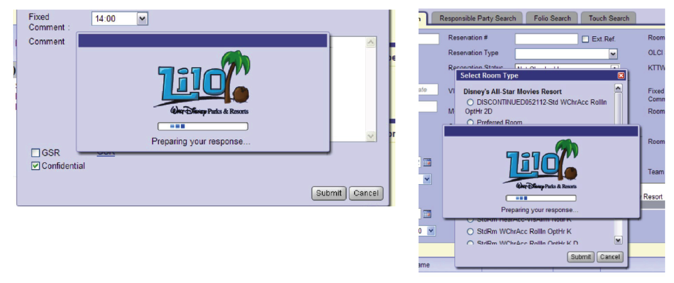
**Iconography**



**Identified Concerns**

As changes were made in LILO that the UE team was not involved in, new icons were created that do not follow UE Standards. Inconsistent use of iconography between LILO and other applications causes user confusion. Some icons are used as a control instead of a visual indication or status. Other icons were designed without considering Color Blind Cast Members.

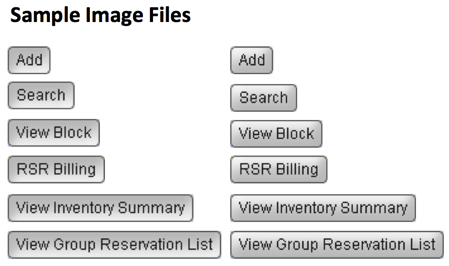
**Processing Window**



**Identified Concerns**

Excessive use of the processing window causes user frustration and an increased wait time to interact with the UI. The processing window is appearing on selection of controls such as radio buttons, checkboxes or cancelation of an overlay. The selection of these controls is preforming unnecessary calls to validate user actions. This issue is happening globally throughout the LILO application on almost every screen, causing the perception that the UI is running slow.

**Button Image Files**



**Identified Concerns**

There is unnecessary image use. The LILO UI should consist of HTML/CSS generated and styled buttons that resize dynamically based on the text needed. Instead, each button in the application is comprised of its own set of flattened image files that include its text label and different button states. The use of this method causes latency through the additional load time for each screen. Possible accessibility concerns due to the fact screen readers (such as JAWS) will not be able to read the button text label. The use of flattened button images prevents the application from supporting Internationalization.

**Usability Engineering Recommendation for Next Steps**

In order to address user efficiency, the Usability Engineering team recommends working with the business and taking UE actions for each prioritized area.

Next steps would include working with the business to identify pain points, identifying why multiple sessions are being used, and prioritizing the areas to address.

For each prioritized area, the UE team recommends conducting field observations and assessing original design versus implementation and how it is used today. It is also important to identify inefficiencies and inconsistencies, and to document the recommended solution to these issues. For implementation, it is recommended to work with the UI development team and release management.