LaundryChamp
Design Document
02.08.2016

Kerby Hughes
Sarang Mongia
John Peterson
Harsh Pandey
Team 15
Index

● Purpose 2
  ○ Functional Requirements
  ○ Non Functional Requirements

● Design Outline 6
  ○ Components
  ○ High-Level Overview
  ○ State/Activity Diagram

● Design Issues 11
  ○ Functional Issues
  ○ Non Functional Issues

● Design Detail 14
  ○ Data Classes Diagram
  ○ Data Classes Description
  ○ Sequence Diagrams
  ○ UI Mockups
  ○ Database Design
  ○ API routes
Purpose

Residents of multi-unit housing, such as students or apartment tenants, must coordinate their use of the shared laundry facilities with the schedules of all the other users in these domiciles. There is demand for a system that provides information and communication tools to help optimize their time using the laundry facilities. Washers and dryers are simple to operate, but residents cannot effectively share this resource without good information about machine availability. To gather this information requires residents to physically visit the laundry facilities without guarantee of any machine availability. This is cumbersome, especially in the common case that the facilities are located far away from their room or apartment.

Alliance Laundry provides WashAlert, which is a stock web application that their customers can host to make public websites with laundry facility availability information. In our research, these sites are not well designed and lack important features. There have been other mobile apps that provide laundry facility availability information, but these apps focus on only a single site, such as a single college campus.

LaundryChamp will take the raw data provided by these public systems and incorporate it into a more usable mobile application. We think we can provide a far superior product to the stock solution. It will be a more general solution that can work with any WashAlert facility to provide a better experience for their users without commissioning a custom application.

Functional Requirements

1. Users can check the availability of washers/dryers and set reminders
   
   As a user,
   
   a. I want to see if there are any washers/dryers available right now.
   
   b. I want to be able to be notified when there are ‘x’ number of washers/dryers available.
   
   c. I want to be able to be notified when there are ‘x’ number of minutes left before my washer/dryer is finished.
   
   d. I want to see how long it will be until a washer is available.
   
   e. I want to be notified when my laundry is done.
f. I want to be able to choose between an alarm/automatic email/mobile notification to remind me when my laundry is done.

g. I would like to set weekly reminders on my phone to do my laundry.

2. Users can get usage statistics for every location

   As a user,
   a. I want to see which day is best to do my laundry by seeing past usage statistics for various days of the week.
   b. I would like to be automatically recommended days and times to do my laundry.

3. Users can share information regarding lost items

   As a user,
   a. I want to let other users know about lost items of clothing.
   b. I want to be able to contact the person who posted the information about a lost item.
   c. I want to be able to mark lost items as claimed if they belong to me and I have managed to get them back.

4. Users should have location specific features

   As a user,
   a. I should be able to raise requests for refunds.
   b. I want to know if the laundry room provides detergents and other supplies.
   c. I want to know the location and the layout of the laundry services in my residence hall or apartment.
   d. I want to be notified regarding scheduled maintenance.
Non Functional Requirements

1. **Client Requirements**
   
   As a developer,
   
   a. I want the application to be able to be used on an Android phone.

2. **Server Requirements**
   
   As a developer,
   
   a. I would like the server to be able to save the WashAlert data to a local database.
   
   b. I may want the server to be able to save user state and preferences to a local database.
   
   c. I want there to be both a development and production server.
   
   d. I want there to be both a development and production database.
   
   e. I want to be able to test the client against both the development and production servers.

3. **Design Requirements**
   
   As a developer,
   
   a. I want both the client and server applications to share a set of class models to be sent and received via the API.
   
   b. I want there to be build scripts to deploy the server to both development and production environments directly from the Git repository.

4. **Performance Requirements**
   
   As a developer,
   
   a. I want the server to be able to handle enough concurrent users to support a university residence population.
b. I want the server to not create excessive requests to the public WashAlert websites.

c. I want the Android application to be responsive to all user requests.

d. I want both the client and server to gracefully handle errors and the failure of other components that they interact with.

e. I want the server design to scale to support additional load without requiring any change to the API or client code.

5. Appearance Requirements

As a developer,

a. I want the application to be aesthetically pleasing and easy to use.

6. Security Requirements

As a developer,

a. I want any user data that we collect, such as camera photos, location data, or phone numbers, to be either anonymized or properly secured on the user’s device before being transmitted or otherwise handled by our system.
Design Outline

Our project is a system that provides laundry availability data to all users of a laundry facility. Our implementation will use a Client-Server model. The server will access data stored in a database and allow access to multiple concurrent Android clients via a web API. While the main architecture is Client-Server, we will use a modular service architecture to populate the database with information.

1. **Android Client**
   a. The Android client, running on each user’s phone, will be the interface to our system.
   b. The Android client will retrieve and send data to our web server via HTTP requests, passing data in JSON format.
   c. The data retrieved from the server will be parsed, formatted, and presented to the user to display information such as laundry machine availability.

2. **API Server**
   a. The API server will provide the abstraction layer on top of our database to make it easy for clients to request and send information.
   b. The API server will accept HTTP requests using REST URI conventions.
   c. The API server will query the database to retrieve the requested information.
   d. The API server will return the data from the database in JSON form as part of the HTTP response.

3. **Database**
   a. The database will store all of the data used in LaundryChamp.
   b. As the database provides persistent storage, the API server and other modules can be stopped, started, and duplicated to meet client demand without loss of data.
   c. Typical data stored in the database will be relationships between laundry machines and their location, and time data related to a laundry machine’s status.
4. **Parse module**
   a. The parser will be another module in our system which requests and reads WashAlert web pages and stores the relevant data from these pages to our database.
   b. The parser will act independently of the other modules in the system.

5. **Statistics module**
   a. The statistics module will take information from the database, analyze it, and store its results back into the database. This data will then be available for consumption by the API server.
   b. The statistics module will act independently of the other modules in the system.
High Level Overview of the system

The Android Client and Web API Server form a many-to-one client-server architecture to serve a large number of concurrent users. When the Server needs to fulfill a request from a Client, it queries the Database for information and sends it back to the Client. When the Client asks the Server to store some information, the Server will write that data to the Database. The Parse Module and Statistics Module form a backend service architecture which provide the bulk of the data which the Server can send to Clients. The Parse Module populates that Database with WashAlert data gathered from WashAlert websites. The Statistics Module analyzes data from the Database and saves its analysis back to the Database. These Modules can perform their work independently as needed in order to keep the data current.
Protocol, Service, and Hosting Overview

The Android Client, API Server, Parse Module, and WashAlert websites all communicate over HTTP. The REST API that we chose is based on the standard HTTP request types, such as GET, POST, PUT, and DELETE. We use these HTTP requests for communication, and pass data in a standard JSON format. The backend service architecture, made up of the API Server, Parse Module, Statistics Module, and the mySQL database hosted in RDS all communicate via mySQL queries, and data is returned from the database as table rows. Although the AWS modules are independent, they are contained in Amazon's virtual private network, while the Android Client and WashAlert websites make and respond to requests over public endpoints.
Activity/ State Diagram
Design Issues

Functional Issues:

1. Do users need to login to use our service?
   - Option 1: Allow the creation of login id using Facebook, Google+ etc
   - Option 2: No login id creation
   - Option 3: Create a username and password that is unique to our service
   Decision: We chose not to provide an option to create a user id as we felt that there was no apparent need for users to do so. They can use all the features and services without going through the trouble of logging in. We will store their preferences locally so that they don't have to choose their residence and location every time.

2. When should claimed items be removed from the lost/found page?
   - Option 1: After a fixed period of time
   - Option 2: Immediately after it's marked as claimed
   Decision: We plan to remove a claimed after a fixed period of time to take care of the unlikely event in which someone accidently claims an item that does not belong to him/her. By keeping the claimed item in the list, the original owner will have a chance to contact the person who posted the item and claim it himself/herself.

3. How should the Residences and laundry locations be displayed?
   - Option 1: List all the residences separately and let the locations be contained within the residences
   - Option 2: List only the locations independent of the residences
   Decision: We plan on implementing option 1 as there are several residences that have multiple locations within them. For example, Purdue University has several dorms that offer laundry services. Therefore, intuitively it makes more sense for the locations to be a sub-list contained within a residence.
4. How should the laundry machine status be displayed?
   - Option 1: The status should be color coded and displayed as an image
   - Option 2: The status of every individual machine should be displayed next to it
   - Option 3: Implement both the features

Decision: We have decided to implement both the features as it will let enhance the overall user experience. Even though this may require a little extra effort from our end, this will eventually be beneficial for the users.

Non-Functional Issues

1. How are we going to host our backend services?
   - Option 1: Amazon Web Services
   - Option 2: Virtual Private Server providers such as Digital Ocean.

Decision: We chose Amazon Web Services for hosting our project’s backend components because the Elastic Beanstalk product provides an abstraction over a bare linux server environment that allows us to easily deploy a wide variety of applications without needing to perform server maintenance tasks.

2. Which languages are appropriate for implementing our backend services?
   - Option 1: C#
   - Option 2: C++
   - Option 3: Go

Decision: Although we had previous experience with C++ and C#, we chose the Go programming language for building our API server, parse module, and statistics module because it can easily be hosted in AWS, and it has built-in standard libraries designed for implementing web servers, parsers, and other server components.
3. What type of database is most appropriate for our data?
   Option 1: Relational databases such as Microsoft SQL or mySQL
   Option 2: A NoSQL database
   Decision: We will use mySQL to implement our database because of its ubiquity and ease of implementation with Go. mySQL is a relational database which we think will be appropriate for our data sets.

4. How can our WashAlert data be kept current without overwhelming the public WashAlert webpages with requests as the number of clients grow?
   Option 1: Refresh automatically after a regular interval of time (ex. 2 minutes)
   Option 2: Let the user decide when he wants to refresh the data
   Option 3: Refresh the data only when the user opens the app.
   Decision: We have decided to let the application refresh its data automatically at a regular interval time. We may eventually give the user an option to refresh it whenever he/she wants to do it. But in our initial sprints, we do not plan to do this as we do not plan to overwhelm the WashAlert webpage.
Design Details

Data Class Level-Design

Residence
- name : String
- logo : Image
- locations: List<Location>

Machine
- type : MachineType
- id : String
- status : MachineStatus
- timeRemaining : Time

<<Enumeration>>
- dryer
- washer

<<Enumeration>>
- available
- endOfCycle
- inUse
- notOnline
- outOfOrder

Contact
- name : String
- address : String
- email : String
- phoneNumber : String

<<Enumeration>>
- claimed
- unclaimed

Location
- name : String
- address : String
- logo : Image
- contactInfo : Contact
- machines : List<Machine>
- lostItems : List<LostItem>

LostItem
- status : ItemStatus
- itemDate : Date
- picture : Image
- description : String
- contactInfo : Contact
Description of Data Classes and their Interactions

Our data class design is based on an understanding of the types of objects in our domain, and how they will be represented in our relational database. They therefore also represent the “Model” classes that our Java client will use, the “structs” that our GoLang API Server, Parse Module, and Statistics module will use, as well as the structure of our API routes and the JSON structures that will be passed between the API Server and the client.

Residence:
- Represents a WashAlert installation.
- This is the highest level in the hierarchy that a user can select.
- Contains identifying data about the site, such as the College’s logo or name of the Apartment complex.
- Created when a WashAlert website is first parsed by our system.

Location:
- Represents a room or floor in a residence where a user does their laundry.
- This will be the main interaction point for a user, as they will generally always use the same laundry location for their laundry needs.
- Created when a WashAlert website is first parsed by our system.

Machine:
- Represents an individual washer or dryer
- Users use this class’s information to check if a machine is available or in use, and may request notifications about time or status updates specific to a machine.
- Created and updated when a WashAlert website is parsed by our system.

Contact:
- Identifying information provided by users
- When part of a location, a Contact instance may refer to the RA or maintenance contact information
● When part of a LostItem, a Contact instance may represent the information a user provides as a way for other users to contact them if their items are found.

● Created for a Location when a WashAlert website is first parsed by our system.

● Created for a LostItem when a user saves a LostItem instance.

**LostItem:**

● Represents information about clothes or supplies that a user wants to let other users know about.
● May contain a photo or Contact information
● Created by users and tied to a specific Location.

**MachineType:**

● Used to identify the type of machine (washer or dryer).
● Created when a WashAlert website is parsed by our system

**MachineStatus:**

● Used to identify the availability of a machine.
● Status examples include “Available”, “In Use”, “Payment In Progress”, “Out of Order”, etc.
● Created when a WashAlert website is parsed by our system

**ItemStatus:**

● Used to identify whether a LostItem has been claimed.
● Created or updated when a user creates or modifies a LostItem instance.
Sequence Diagrams

● Sequence of events when the user first starts the application

![Sequence Diagram](image-url)
Sequence of events when user adds a lost item

- User opens the app and goes to the lost/found activity

  - Application displays a list off all the lost items and some of the recently claimed items

- User uploads a photo of a lost item

  - Application sends the image to the Server

  - Server sends the file to the database

  - Server returns a list of lost items

  - Application displays an updated list of lost/recently claimed items
Sequence of events when user requests to be notified when a machine becomes available

Actor

User selects a location

Application displays a list of available and In Use machines

If all machines are in use, then User requests to be notified upon the availability of a machine

Application stores the user request locally on the device

Requests an updated list of machines

If a machine has become available, then the application notifies the user

Returns an updated list of machines

Requests an updated list of machines

Returns an updated list of machines

Server

Database
UI Mockups

- List of residences
List of locations

<table>
<thead>
<tr>
<th>Location</th>
<th>Washers</th>
<th>Dryers</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cary Hall West</td>
<td>0/10</td>
<td>0/10</td>
</tr>
<tr>
<td>Earhart Hall</td>
<td>0/10</td>
<td>0/10</td>
</tr>
<tr>
<td>Harreion Hall</td>
<td>0/10</td>
<td>0/10</td>
</tr>
<tr>
<td>Hawkins Hall</td>
<td>0/10</td>
<td>0/10</td>
</tr>
<tr>
<td>Hillenbrand Hall</td>
<td>0/10</td>
<td>0/10</td>
</tr>
<tr>
<td>McCutcheon Hall</td>
<td>0/10</td>
<td>0/10</td>
</tr>
<tr>
<td>Meredith Northwest</td>
<td>0/10</td>
<td>0/10</td>
</tr>
</tbody>
</table>
A sample location
Sample usage statistics for a location

Hawkins Hall

Lost/Found | Machines | Statistics

Weekly statistical graph

Detailed usage statistics

Busiest period
Saturday 9-12 PM

Idle period
Wednesday 3-6 PM

Max washers available - 14
Tuesday 6-9 PM

Max dryers available - 10
Sample lost/found activity for a location

- Hawkins Hall

Lost/Found | Machines | Statistics
---|---|---
Lost item 1 description | Contact info | Claimed

Lost item 2 description | Contact info | Claimed

Lost item 3 description | Contact info | Claimed

Lost item 4 description | Contact info | Claimed

Lost item 5 description | Contact info | Claimed

- Checked if the item is claimed

- Add new Lost item
● Notification regarding availability of washer/dryer
Database Design

We are using a relational database implemented with MySQL which will be hosted in the Amazon RDS service. There will be a table in the database for each of the data classes that we identified in our domain. The rows of each table are indexed with a unique Primary Key. Relationships between the tables are created using these Primary Key - Foreign Key constraints. We will implement Views, triggers, and stored procedures as needed to make interacting with the database from the service modules more efficient.
API Routes

The API routes are the URIs that are made available by the API server and they define the communication contract between the client and the server in our client-server architecture. Each route represents a resource or set of resources that can be provided by the server (via GET requests), or created and updated by the client (via PUT, POST, and DELETE requests). A resource is identified by an id, which matches the Primary Key of that resource in our database.

<table>
<thead>
<tr>
<th>Route</th>
<th>Supported HTTP methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>/residences</td>
<td>GET</td>
</tr>
<tr>
<td>/residences/{id}</td>
<td>GET</td>
</tr>
<tr>
<td>/residences/{id}/locations</td>
<td>GET</td>
</tr>
<tr>
<td>/residences/{id}/locations/{id}</td>
<td>GET</td>
</tr>
<tr>
<td>/residences/{id}/locations/{id}/machines</td>
<td>GET</td>
</tr>
<tr>
<td>/residences/{id}/locations/{id}/machines/{id}</td>
<td>GET, PUT</td>
</tr>
<tr>
<td>/residences/{id}/locations/{id}/contacts</td>
<td>GET, POST</td>
</tr>
<tr>
<td>/residences/{id}/locations/{id}/contacts/{id}</td>
<td>GET, PUT, DELETE</td>
</tr>
<tr>
<td>/residences/{id}/locations/{id}/lostitems</td>
<td>GET, POST</td>
</tr>
<tr>
<td>/residences/{id}/locations/{id}/lostitems/{id}</td>
<td>GET, PUT, DELETE</td>
</tr>
<tr>
<td>/residences/{id}/locations{id}/stats</td>
<td>GET</td>
</tr>
</tbody>
</table>