Team LunchBruhs / LunchMenu

1. Overview of Problem

The restaurants in UCD change their menu every day making it a difficult task to decide where to eat. We decided to make a system that allows restaurants to host their menu online for students to access via an iOS app. We decided to implement this idea as a distributed system consisting of a menu service for each restaurant and a central hub service that the iOS app can interact with.

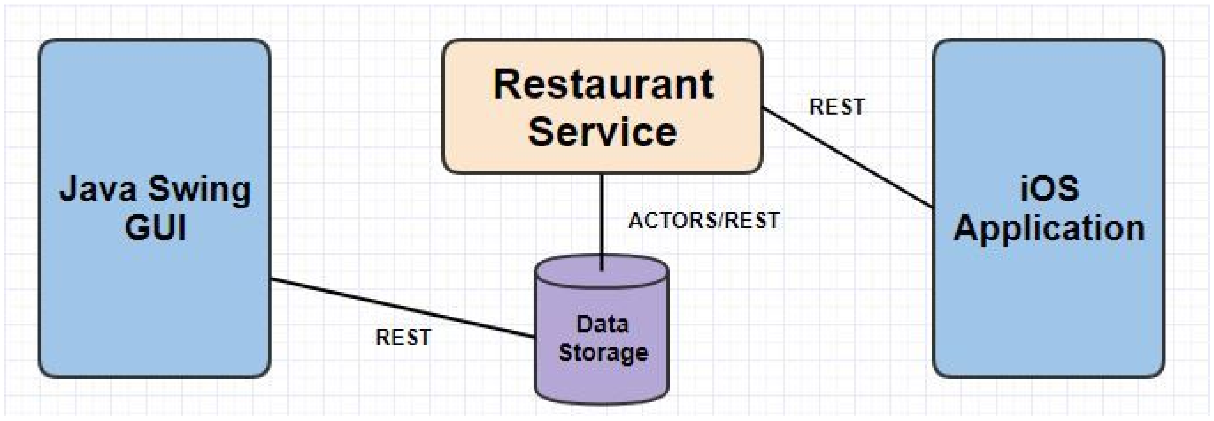
The menu services expose the restaurant's menu through a REST API. By separating the restaurants out into separate services it allows other applications to interact with them independently of our system. Having the services separate also protects us from having the app go offline if there is an issue with the menu service.

The hub service acts as the interface for the iOS app. When it's API is polled by the iOS app, it gathers the menus from all of the restaurant services and sends them back to the app.

We decided to implement an application to allow the restaurant owners to easily update their menus. This app bypassses the hub and communicates directly with that restaurant's menu service to update the menu.

We think a distributed solution to this problem is appropriate as it allows each restaurant to host its own menu what can be accessed by services other than our system. This could be used to query each restaurant for a central message board etc. By setting this system up as a distributed system it is very easy for a new restaurant to add themselves to the service. If we joined all of the menu services into a central database service then we could be at risk of the whole system going down if that service was to crash. By designing the system to use separate services for each menu this is no longer an issue, if a service goes down that menu is simply not displayed.

2. System Architecture



As shown in the diagram above, there are two largely independent applications of our project, our Java GUI and our iOS application, along with our backend restaurant service and data storage.

Our GUI allows users to select a restaurant from storage, and displays a list of fields for the user to fill in. What they fill in is a menu item and when they have finished filling in the details they click add and the menu item will be sent back to be updated in storage. The interaction between these two components is simply done using REST; a GET method to access the information to begin with, and a POST method to update the menus.

Our iOS application displays the current restaurants and their menus from storage in a clean, user-friendly interface. The application interacts with the restaurant service via a GET request, which returns the information needed.

Within the restaurant service, the task of accessing the menus is distributed via actors. Each worker actor is in charge of accessing a particular restaurants menu, and returning that to the master actor to make up the full json string to return to the iOS application. Each worker actor interacts with the data storage via a REST GET request.

Finally, the data storage component. This component has both GET and POST request methods, and stores the json string for each restaurant in a separate file on the system. The GET request simply accesses one of these files and returns the json string, and the POST request is used for updating or creating new json files.

3. Technology Choices

Technology	Motivation for Use
iOS	Phone apps are a common way of allowing users to interact with a system. By using an app it allows us to extend in the future to do more than just display menus. This could be giving notifications when a certain food is on sale etc.
REST	REST is a very clean API with a simple interface for doing HTTP operations. As we would only need GET and POST operations this was an easy choice.
Actors	Actors were selected to distribute the accessing of different restaurant menus within this project. This choice was made mainly for two reasons. Firstly, because of the fact that actors are able to work asynchronously, therefor speeding up the process by being able to access all of the restaurants simultaneously. Secondly, since each task is handled by an individual worker actor, should one of these actors fail, the system will still continue to work and will simply not display that portion of information within the application.
Java Swing GUI	We initially wanted to make this a web page but jQuery was more trouble than we expected. Instead we decided to make a GUI application using Java's Swing API. This is the most popular GUI API for java and is well documented.

4. The Team

Student Number	Name	Assigned Task(s)
12254676	Mohsen Qaysi	- Implement a full iOS App - Read JSON Data from the system using REST.
14308031	Carl McCaffrey	- Implement the MenuService to store a restaurant's menu JSON and provide access to the JSON through a REST API
14701935	Samuel Bryan	- Actors implementation

		- Link together Actor & REST code
14343161	Jack O'Beirne	- Java User Interface - Adding menu Items to Json from interface

4. Task List

4.1 Task 1: iOS App:

The idea started when we were brainstorming. I had an issue which bothered me a lot ever since I came to UCD. I always go to a restaurant and find out that nothing for me to eat as the menus changes everyday. I wished there was a website or UCD Mobile Application to show you what is available in each restaurant in campus. We decided to make a mobile application to solve this issue:

- I started to image the idea by create a sketch for it. I usually do it this way on all the projects I work on. The reason for doing it this way is to do help guide me visally.
- The design was every simple for a reason; to make it as easy as possible for the user to get the information with a little clicking involved.
- I research everything needed to create the app such as:
    - URLSession: for downloading content from the internet. In our case to download JSON Objects.
    - Create a dummy JSON Data to allow the app to work until the backend fished. The JSON data structure was flatten to allow easy access to data when parsing them using the URLSession API. Also, a local server was used to host the data.
    - After having the data and server setups done, the app development started in stages:
        1. Creating the first UIView: This view has the restaurant lists in a table view format which has an image of the restaurant and name. To easily identify them.
        2. Menu UIView: the user than choose a restaurant menu to look at. When clicking on the menu, the app will go to the second UIView. The view display the menu for that day.
        3. Menu layout and design:
            a. The design included the most important information the user need such as:
                i. Name of the restaurant.
                ii. Type of food they serve.
                iii. Opening hours and days.
                iv. The food menu itself.

v. Allergies in a very colorful layout to make it easy to understand - In icons format.

Issues faced:

- The allergies section which has the icons in it was the hardest part in the app development phase. I developed all the features until I reached that section.
- I was using the interface builder to develop the app, but creating that inner section was impossible. This issue forced me to recreate the app from scratch.
- This way helped me to have more flexibility to make the app the way I wanted it to be. But making it using only code with no interface builder ment I have to write tones of code to achieve the same thing.
- The issue was solved using this method of develop the app. However, the layout constraints were really hard to create. I used something called “NSLayoutConstraint with Visual Format” to achieve it the end.

Here’s a video of the app in action: https://youtu.be/uRzgrFqfZlY

4.2 Task 2: Implement the MenuService:
Initially I was going to implement a full SQLite database to sure the restaurant's menu. We had decided that we were going to be using REST to handle communication between the services and that we were going to send the menus using JSON.

The REST API consists of A GET and POST request on the ‘/menu’ URL. A POST request will save the given menu data to the disk and a GET request will grab the menu data from the disk and send it back to the caller.

The initial implementation of this service had the following flow:

- POST: JSON -> Java Object -> SQLite tables
- GET: SQLite tables -> Java Object -> JSON

The MenuService did not look at or modify the data in any way and this was all unnecessary. I re-implemented the backend of the service to simply dump the received JSON directly to a file on the disk for a POST request and to grab the JSON from the disk and send it back for a GET request.

The final version of the MenuService is a much simpler and contains less dependencies and potential failure points than the full SQLite database implementation of it.

4.3 Task 3: Actors implementation:
As we had not directly completed an assignment on actors within the duration of this course, the first step was to get used to how actors work. I did this by implementing the examples given in the module slides.

It seemed fairly straightforward to get the actors to do the tasks we had established, however there were a few issues along the way:

- How to shutdown the actor system when the work has finished.
- How to put a wait in the code until the actor system is terminated.
- How to access variables from the master actor.

The first two issues I solved independently from the group, by simply investigating more into how actor systems work. The first problem I solved by terminating the actor system from within the master actor after it's work was complete, using one of the actor system methods. For the second problem I simply put a while loop that loops until the actor system is terminated, which I was able to check by a boolean function in the actor system.

The final issue I discussed with the group. We decided to have a static string within our REST GET request, that the master actor would then update with the complete json string once the task was complete, allowing us to then set this as the response entity for access within the iOS application.

With all of these issue solved, the actor implementation was complete. The iOS application would make a GET request to the restaurant service, this service then starts the actor process, the master actor makes a number of worker actors equal to the number of restaurants, and each of them get the menu json via a GET request and return that to the master actor, once all of the worker actors are finished the master actor builds up the complete json and returns that to the iOS application to display the information.

4.4 Task 4: User Interface for adding menu items:
For this part of the project, the original idea was to use a web page, using jquery/javascript and html/css to try to read the json on the localhost, and then allow the different restaurants update their menus with new items, convert that back to json and then push that to he rest service. However after several different methods were attempted and encountering the same issue every time with accessing the json, we were unable to find any workable solution and so had to abandon using a webpage. It was then that we decided to use a gui builder to build a user interface, to allow input of menu items that then pushes that to the json.

To accomplish this in MenuManager.java, after accessing the specified restaurant, based of the URL given in the user interface, we used gson to:

- Convert the Json into a Menu, in the loadMenu method using gson.fromJson(), which is specified by Menu.java, which reads the whole Json for the restaurant specified and converts it into the specified fields in the Menu.
- Then when a user wants to add an item to the menu, in the addItem method it converts the MenuItem which is taken in, MenuItem.java is a class set up with specified variables and then is used as the ArrayList type for the menu ArrayList in MenuManager.java, it adds it to the menu, and then it converts the menu back to json using gson.toJson and then posts that to the restaurant.

In the user interface, it's setup to take the name, served with, cost, calories and allergy info, for a dish and then when they hit add, in the action listener for the add button, it converts cost and calories to double and int respectively, and then adds those to the menu item, and for the allergies, being radio buttons, we had it check if it was clicked and then add that to the array list of allergies in the menu item, and then add the item to the menu using the addItem discussed above.

5. Reflections
The project went very well. We chose to use REST and Akka as the core of the system and those technologies work very well in the context of our system. REST is a clean way for different subsystems to communicate using HTTP. Being based on HTTP allows us to use it not only as our primary communication system between the parts of the system itself, but also as the way the iOS app and Java GUI communicate with the system. It has the limitation of being limited to HTTP requests, but we only required GET and POST requests in our usage.

Akka actors were used within our system to access the JSON menu information from our data storage. The reason we chose actors for this task was mainly for the heavy distribution of work, and the easy error handling that they provide. Because users will view our menu information from an iOS application, it is likely that a large number of requests will be needed at any given time. Actors allowed us to distribute this work, so that the actors can asynchronously access each restaurant at once (with one worker actor per restaurant), speeding up the system considerably. One other major benefit to using actors for this task is the way that the actor system handles errors (for example, one worker actor failing). Since each worker actor is independant, should one of them fail, the overall system will still finish, and only not display the menu information regarding that particular restaurant that the worker actor was dealing with, rather than the full system crashing and no information being displayed at all. Our experience with using actors within this project has shown that they are easy to integrate within other APIs/applications, in our case the REST api, making the code fairly intuitive.

The mobile app allows users to easily access the system from their mobile phones. By making an app instead of a website it allows us to have greater control over the look of the interface and makes it easier for the user to use on a daily basis. The only issue with the app are that it is currently only written for iOS and will not work on Android phones, but this is not a major issue as an Android port of the app would not be difficult to make.

In the original specification of the project we had planned to use a Web Client to allow the restaurants to manipulate their menus on the system so that they could add menu items, remove them or update items. We later found out however through many different attempts that the web client was going to be complicated to get working because of a recurring problem connecting to the JSON in the localhost, because we could not access it due to Access Control errors. After several different attempts to get this web client working we had to abandon it because of this recurring issue and not being able to find a working solution, so instead we decided to use a Java Swing GUI to allow the users to add items to the menu by posting it to REST.

During this project we learned a lot about Akka Actors Java Swing GUIs and iOS development. We had some experience with REST before from the practical earlier in the module but everything we know about actors was learned while working on this project. One of our team members had some experience with iOS development and learned how to interact with REST in an iOS application. The Swing GUI was fairly straightforward and the Java documentation taught us everything we needed to know to use Swing to create an interface for adding items to a menu in the system.

References

[1] Sandoval, J. Restful java web services: Master core rest concepts and create restful web services in java. 2009.

[2] Louvel, J., Templier, T. and Boileau, T. Restlet in action: developing RESTful web APIs in Java. 2012.

[3] Gupta, M. Akka essentials. 2012.

[4] Thurau, M. Akka framework. 2012.

[5] Piroumian, V. Java GUI Development. Sams. 1999.