Linnæus University
Sweden

Report

# Design Document

*[Subtitle]*

*Author:* Melat Haile
*Supervisor: Mauro Caporuscio*
*Semester:* 22VT
*Course code:* 2DV608

Linnæus University
Sweden

# Table of contents



# Appendix

Provide a Design Document containing the following sections:

# 1. General Overview

System Description Nowadays, there are many stores which offer the same or similar products, but the prices often vary from store to store. In today's times of the recession, people are trying to buy products where they are cheaper and save money. This turns out to be a challenging task because of the sheer number of stores and products that they offer. SmartGrocery is envisioned as a mobile application which should ease up the daily process of buying groceries for the end users. SmartGrocery will enable users to enter groceries they want to buy to a list, and then find out in which store they can buy the cheapest groceries and save money or find the closest store with the requested groceries. Although, there are some existing web sites and applications with purpose of finding cheapest items across multiple stores, they do it on the basis of a single item.

**User shall be able to:**

1. Browse all products in the system
2. Edit the shopping list (add/remove items, modify quantity)
3. Add / update information about a product, including its price and the store where the product is located
4. Get suggestions of stores according to the shopping list and selected criteria (products price or stores distance)
5. Browse nearby stores on a map

**Application shall be able to:**

6. Update the products and stores nearby on user demand,
7. Store user shopping list with quantity of products,
8. Display the list of stores ordered by distance
9. Display the list of products ordered by price
10. Automatically collect products and prices from available store's web site (web scraper),
11. Communicate with Client applications across different platforms,
12. Automatically insert the stores and their locations into the repository (store loader).

# 2. Assumptions and dependencies

**Assumptions**

- The device must be able to connect to the internet via Wi-Fi or cellular.
- Assume the user has a smartphone.
- Assume that the developers will develop the app in 4-6 months.
- Assume that the app works on different platforms such as iOS, android, …
- Assume that the app works with different version of softwares.
- Assume that 98% of users will find the app useful enough to become regular users.

**Dependencies**

- Third party software such as google Maps and the websites of the grocery stores.
- Hardware such as beacons

I prefer to mix top-down and bottom-up approaches. Given the currently available domain knowledge, top-down provides us with a structured and logical architecture. In the specifications, the domain knowledge is encapsulated. The system is documented by the requirements as well as a good collection of architectural views. As a result, each team member will be able to comprehend it. By doing early trials, bottom-up allows us to reduce risks and uncertainties. Early test sets can prove or disprove potential solutions to low-level problems. The "top-down" analysis takes into account the viability of these solutions as well as the domain knowledge gained (via low-level requirements).

The development approach is requirements-driven:

- high-level system requirements are the outcome of the assessment of stakeholder requirements and a top-down analysis of the system, while
- bottom-up experimenting provides low-level needs (for example, hardware specifications).

# 3. Architecturally significant requirements

Refer and link to the requirements that significantly affect the design, and discuss how they must be implemented

**Performance**

**FR 1, 2, 6, 9** -
- Browse all products in the system
- Edit the shopping list (add/remove items, modify quantity)
- Update the products and stores nearby on user demand
- Display the list of products ordered by price

This project's performance will be crucial. In every engagement between the program and the user interacting with the application, the design must support the performance targets that have been agreed upon.This is most likely the project's most processor-intensive component. In addition, the gateways will have to deliver requested pages to users at a suitable speed. All database requests will be handled by the database server.

**Compatibility**

**FR 12** - Automatically collect products and prices from available store's web site (web scraper)
The software should be able to work with other products that are designed to be compatible with other products.

**Portability**

**FR 11** - Communicate with Client applications across different platforms
The app must be applicable to all kinds of mobile devices. As a result, the architecture must be adaptable to many conditions and environments. The application should work on different platforms such as iOS, android, …

**Reliability**

The software can perform a required function for a certain period of time under specified conditions. A redundant database server will be installed, so that if the primary database server fails, the gateways will immediately switch to the secondary database server. The mechanism for synchronizing these 2 databases is still being worked out. Consider:

- Both servers are updated by each gateway.
- The redundant server is constantly checking the tables for fresh data, and an archive field is being implemented.

- A thorough backup of the entire database is performed on a regular basis.
- A trigger on the main database that copies all data to the secondary database automatically.

**Maintainability**
The software should be able to be maintained easily.

**Data Access**

Maintainability and extensibility require the use of a separate data access layer. The data access layer should be in charge of managing connections with the data source and executing commands against it. The data access layer may be dependent on business entities depending on the business entity architecture. However, the data access layer should never be aware of business processes or workflow components.

• Avoid connecting the application model to the database schema directly. Instead, consider implementing a mapping or abstraction layer between the application model and the database schema.

• Rather than enforcing data integrity through data layer code, data integrity should be ensured in the database.

• Move business decision-making code to the business layer.

• Avoid using separate layers of the application to access the database directly. Instead, a data access layer should handle all database interactions.

**Extensibility**
Maintenance and development must be supported by the architecture.

**Reusability**
The code that was developed and the components that were used should be able to be reused without difficulty.

# 4. Decisions, constraints/priorities and justifications

**Client/server architecture**

We can use client-server architectures with layers and tiers now that we've decided to design a progressive mobile application. Clients are often located at workstations, whereas servers are located elsewhere on the network, frequently on considerably more robust machines, in a client-server design. When the clients and server are doing normal processes, this strategy is quite useful.

**A client-server architecture functionality is divided into several tiers**.
**And we will be using 3-tier architecture.**

**3-tier**

In contrast to a two-tier design, which has no intermediary, a middleware sits between client and server in a three-tier client-server architecture. The middleware will initially receive a request from the client to retrieve certain information from the server. It will then be sent to the server for processing. When the server sends a response to the client, the same procedure will be followed. The three main layers of 3-tier architecture are the presentation layer, application layer, and database tier. Each of these three levels is regulated at different points. The presentation layer is managed by the client's device, while the application layer and database tier are managed by the middleware and server, respectively.

The 3-tier architecture is more secure, has an invisible database structure, and offers data integrity given the presence of a third layer that provides data control.

**“Thin client”**

Thin clients don't have their own application code and are completely reliant on the server to function. As a result, they are less dependent on the operating system or mobile device than fat clients.
The advantages of Thin Clients
Thin clients have the following advantages:

- Extended lifecycles
- Reduced power consumption
- Increased central controllability
- More scalability
- Lower susceptibility to malware threats

**Priorities**

1. Performance
2. Portability
3. Compatibility
4. Usability
5. Maintainability

# 5. Architectural Patterns

5.1. Describe the architectural pattern(s) that you will use, why you will use it (them), and how the architecture will be consistent and uniform.

A client-server architecture will be used to build the system. The model-view-controller paradigm is made up of a server-side application and a client-side application (MVC).
The main web server serves the client application. It's a thin client with a lot of business logic in it. It differs slightly from a standard MVC in that it is not solely responsible for displaying data.

The model and controller are made up of the server-side architecture. As RESTful services, the web server interfaces with the database and API.
When a client requests data from the web server, the web server searches the database and the external API for the information and returns it to the client. Depending on the application's requirements, several external APIs can be implemented.

An event-driven architecture, which is popular in current applications created using microservices, uses events to trigger and communicate across disconnected services. The three main components of event-driven architectures are event producers, event consumers, and event routers. The router filters and pushes events to consumers once a producer publishes an event to it. Producer and consumer services are decoupled, allowing for autonomous scaling, updating, and deployment.

To achieve consistency and uniformity in a system it is important to apply the right architectural pattern for the design.

5.2. Describe the overall Architecture and discuss whether the Design Principles have been addressed or not.

**Layering**

Layers allow you to divide functionality into different areas of focus in a design. In other terms, layers indicate the design's logical grouping of components.

- A logical grouping of components should be represented by layers. For instance, use different layers for UI, business logic, and data access components.

- A layer's components should be coherent. To put it in other words, the business layer components should only perform actions that are linked to the application's business logic.
- To specify the interface for each layer, use an Interface type. This will let you to construct many implementations of that interface, which will increase testability.

**User experience**

- Navigation experience which is consistent. For the look and feel, utilize composite patterns, and for UI processing, utilize controller patterns like Model-View-Controller (MVC).
- Each page/section of the interface focusing on a single task
- Large pages with a lot of functionality should be split into smaller pages.

**Navigation**

Designing a navigation approach that allows users to effortlessly navigate through screens
or pages while separating navigation from presentation and UI processing. To prevent user confusion and conceal application complexity, display navigation links and controls in a consistent manner throughout your program.
• When the navigation logic is complicated, use well-known design patterns to separate the UI from it. The "star forms" concept, in which one central screen serves as a "home page," can be used. From that screen, you can move on to other screens to perform more particular tasks, but you'll always come back to the main home page screen.
• Think about using wizards to allow for predictable navigation between forms.
• To handle common actions from various sources, consider utilizing the Command pattern.

**Exception Management**

An effective exception-management technique is critical for the application's reliability and security. Failure to do so may expose sensitive and vital information while also making the application vulnerable to Denial of Service (DoS) attacks. Designing a centralized exception management and logging mechanism, as well as offering access points within your exception-management system to facilitate instrumentation and centralized monitoring that helps system administrators, is a good strategy.

- Always catch exceptions if you really can manage them or if you require extra information.
- Do not expose sensitive information in error messages and log files.

**The following design principles have been met.**
Increase cohesion where possible, reduce coupling, keep the level of abstraction as high as possible, increase reusability where possible, and design for flexibility.

# 6. Architectural views

• Component diagrams

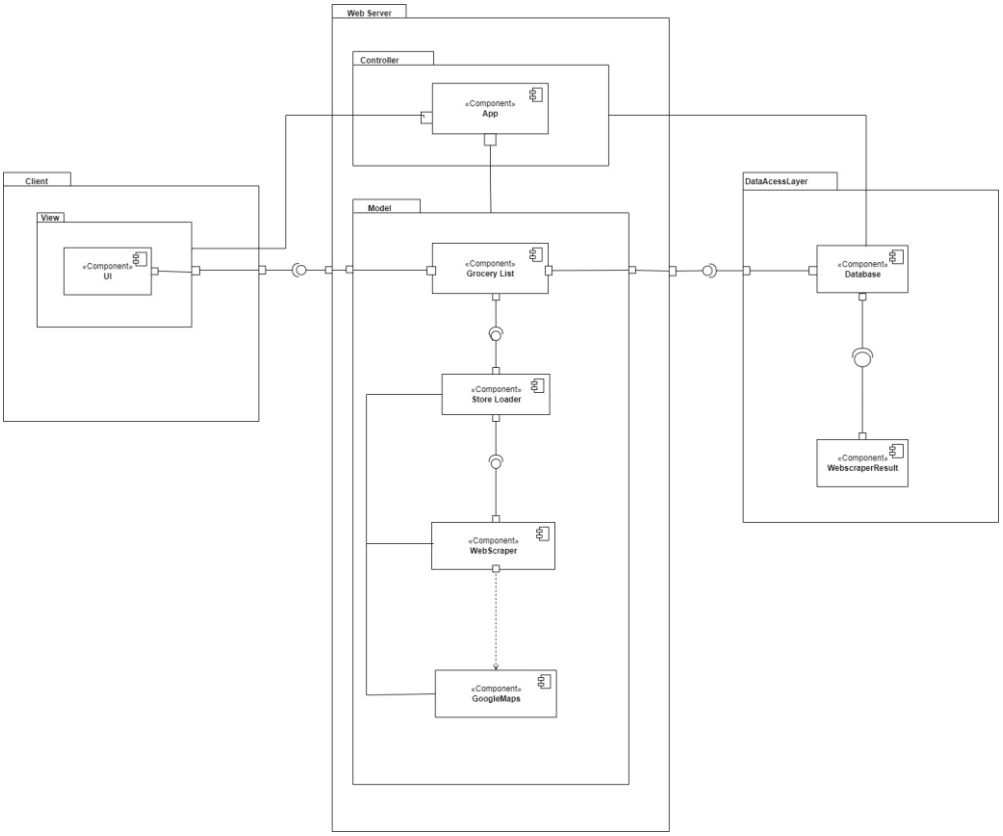


Figure 1

It has three packages: data access layer, webserver, and client. This diagram uses the model-view-controller pattern. It has seven components:
App, Store loader, Web scraper, Web scraper result, UI, Database and GoogleMaps. This architecture is client/server(3-tier) and model-view-controller(mvc) pattern.

• Deployment diagrams

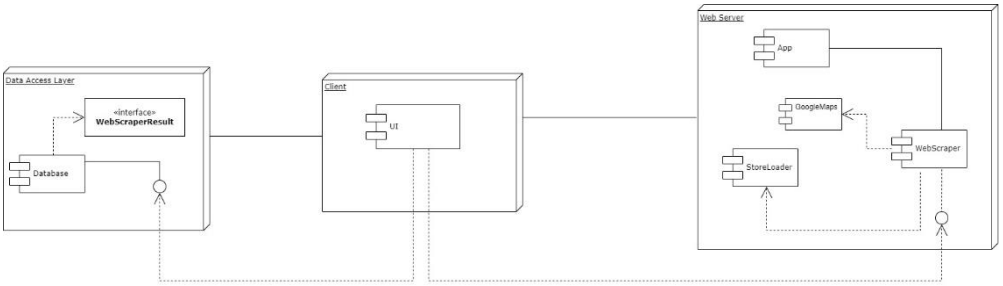


Figure 2

This is a deployment diagram that has 3 nodes: data access layer, client and web server. The UI depends on both the database and web scraper. The client is in the connected to the data access layer and

• **Sequence diagrams –** specify how subsystems interact with each other to implement 1 of the key functionalities (cherry pick the one you refer):

I chose to work on: Display the list of stores ordered by distance

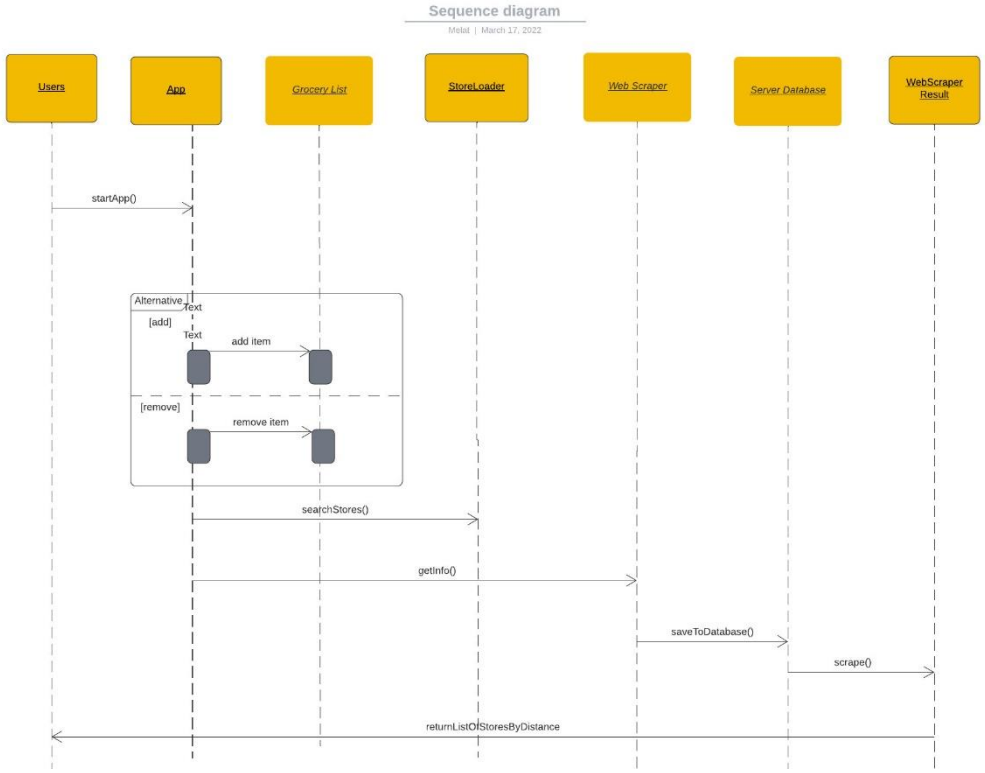


Figure 3

This figure shows a sequence diagram when the application displays list of stores ordered by distance

1. The user starts the application.
2. Adds items to the grocery list. The user can also remove items if he/she doesn’t want it.
3. The application checks stores nearby based on the demand of the user.
4. The application gets information from the web scraper.
5. The application will save data to the database
6. The web scraper result will scrape the data and present it to the user.