



BILBOARD

Team-T4 BugBunny

Design Goals and High-Level Architecture

DILARA MANDIRACI

BURAK DEMIREL

YUSUF TORAMAN

SILA ÖZEL

EREN HAYRETTIN ARIM

Fall 2023

1. Purpose of The System	2
2. Design Goals	2
2.1. Usability	2
2.2. Functionality	2
2.3. Security	3
2.4. Localization	3
2.5. Independency	3
2.6. Maintainability	4
3. Access Control and Security	4
4. Hardware/Software Mapping	6
5. Persistent Data Management	8
6. Subsystem Decomposition	9
6.1. Presentation Layer	10
6.2. Application Layer	10
6.3. Database Wrapper Layer	12
6.4. DatabaseLayer	13
7. Boundary Conditions	13
7.1. Initialization	13
7.2. Termination	14
7.3. Failure	15

1. Purpose of The System

The purpose of this system is to establish a comprehensive and secure web application for the Bilkent University community, designed to support a range of activities, including buying and selling second-hand products, item donations, and borrowing, managing lost and found items, and a comprehensive complaint system. This platform aims to offer an inclusive digital environment for students, staff, and administration personnel, thereby contributing to the overall functionality and community of Bilkent University.

2. Design Goals

2.1. Usability

We will implement an application with a couple of functionalities. So, the usability of the application is an important goal because if our application is easy to use, then the user will spend more time on our application. To achieve this, we will create a simple and clean user interface. Also, we will create a responsive interface that will be compatible with multiple device sizes. The buttons and the input areas will have explanatory text or images (for buttons) on them so the user will know what is wanted from them.

2.2. Functionality

We will create a functional application for Bilkent students to exchange items and share complaints. For exchanging items, our application will work as a mediator between the customer and the seller/lender/donator. But we will not provide a service for money exchange because we thought that this might hinder the security of the application. This is a precaution that we will take so that the users can use this application with peace of mind. The users will only contact the seller/lender/donator

to get the item, and they will agree on a price, if there is any, and the money exchange will be handled between the user and the seller/lender/donator.

2.3. Security

Our application will consist of some sensitive data. For example, we will hold the password of the user. We will store the encrypted data in our database if the data is sensitive. Also, we will use JWT tokens to authenticate the user and keep the user authenticated when they are actively using the application. JWT stores all those data encrypted, so it is hard to decrypt the data and abuse the authentication process. To enhance password security, we will force the users to register with a strong password, and we prompt them to reenter the password for confirmation during registration. Also, we will send a verification email to the newly registered user. Until the user verifies their account, they cannot use our application. This will hinder usability since it might annoy the users, but it is important to prevent non-Bilkent people from using our application with fake email addresses.

2.4. Localization

This application can only be used by Bilkent members. So, the mail address that they will register with must include either “@bilkent.edu.tr” or “@ug.bilkent.edu.tr” extension. This will enhance the security of our application since only Bilkent staff can use this application.

2.5. Independency

The application will enable enhanced modularity, allowing different components such as second-hand sales, donations, borrowing, lost & found, and the complaint system to be independently developed, tested, and maintained. Such a

design ensures easy maintenance and scalability, facilitating updates and adaptation to future enhancements. Also, our backend and frontend code are decoupled so that we can develop them independently of each other.

2.6. Maintainability

Our application may need some changes over time. For instance, some features might be outdated and need to be removed, or we might need to add some new features. In order to do that, we wrote our code in such a way that making changes can be easily done. Our frontend and backend code are independent of each other so changing some parts of the code can be easily done. If we need to change something in our frontend we can do it independently of the backend, and vice versa. Also, we wrote our backend code according to the OOP paradigm, so changes in classes can be easily done without affecting the other parts of the code.

3. Access Control and Security

In our Billboard application, we care about security and authorization issues. We use a logic to authorize only "Bilkent" members to use our system. In addition to our logic, we also get help from some built-in authentication classes of Django. As expected, users can only be able to use the features that are provided and permitted by the front-end user interface. While users have restricted access, the admin can modify, delete, and add all the products, entries, and users.

We use AWS to store our database for our application, a well-known and secure service that enhances the application's security.

Not only for our application, but we also care about our users' privacy and security. When they register to our system, their passwords are hashed and stored in

the database in a hashed way directly from the interface. So, it means there is no place to see a user's password unhashed.

We don't restrict our users according to their status (i.e., faculty member, student, etc.) since we want all users to share their ideas and products with each other. For example, a faculty member may be interested in a product provided by a student. In our access matrix, we wanted to show how users can access different parts when their roles differ. They are not different user types.

	Product	User	Entry	MessageList	Notification
Customer	get_product(...) get_all_products(...) add_to_favorites(...)	get_user(...) create_user(...) update_user(...) delete_user(...)		get_sender(...) get_content(...) send_message(...)	get_product(...) get_user(...) get_category(...)
Borrower	get_product(...) get_all_products(...) add_to_favorites(...)	get_user(...) create_user(...) update_user(...) delete_user(...)		send_message(...) get_sender(...) get_content(...)	get_product(...) get_user(...) get_category(...)
Losing Party		get_user(...) create_user(...) update_user(...) delete_user(...)	get_lost_notice(...) create_lost_notice(...) delete_lost_notice(...)	get_sender(...) get_content(...) send_message(...)	get_product(...) get_user(...) get_category(...)
Donee	get_product(...) get_all_products(...)	get_user(...) create_user(...) update_user(...) delete_user(...)		get_sender(...) get_content(...) send_message(...)	get_product(...) get_user(...) get_category(...)
Vendor	save_product(...) delete_product(...) update_product(...)	get_user(...) create_user(...) update_user(...) delete_user(...)		get_sender(...) get_content(...) send_message(...)	get_product(...) get_user(...) get_category(...)
Lender	save_product(...) delete_product(...) update_product(...)	get_user(...) create_user(...) update_user(...) delete_user(...)		get_sender(...) get_content(...) send_message(...)	get_product(...) get_user(...) get_category(...)
Finder		get_user(...) create_user(...) update_user(...) delete_user(...)	get_lost_notice(...) create_lost_notice(...) delete_lost_notice(...)	get_sender(...) get_content(...) send_message(...)	get_product(...) get_user(...) get_category(...)

Donor	save_product(...) delete_product(...) update_product(...)	get_user(...) create_user(...) update_user(...) delete_user(...)		get_sender(...) get_content(...) send_message(...)	get_product(...) get_user(...) get_category(...)
Reader		get_user(...) create_user(...) update_user(...) delete_user(...)	get_complaints(...) upvote(...) downvote(...) get_target(...) get_target_mail(...)	get_sender(...) get_content(...) send_message(...)	get_product(...) get_user(...) get_category(...)
Sender		get_user(...) create_user(...) update_user(...) delete_user(...)	create_complaint(...) delete_complaint(...)	get_sender(...) get_content(...) send_message(...)	get_product(...) get_user(...) get_category(...)
Admin	delete_product(...) set_product_unavailable(...)	get_user(...) create_user(...) update_user(...) delete_user(...) block_user(...)	delete_complaint(...) delete_lost_notice(...)	get_sender(...) get_content(...) send_message(...)	get_product(...) get_user(...) get_category(...)

Table 1: Access control matrix

4. Hardware/Software Mapping

- Bilboard uses Django 4.2.6 and Django Rest Framework (DRF) 3.14.0 as its backend. Django and DRF offer good maintainability and robustness. Also, DRF highly improves API endpoints. For the frontend, we used React.js 18.2.0. React offers reusable components and good performance.
- Bilboard is a web application that does not require specialized hardware for the client side. Users can access the Bilboard with a web browser. Due to the flexible design of Bilboard's frontend, it can be used effectively with mobile phones via a web browser.
- AWS EC2 provides virtual servers for our database server. These servers allow us to adjust DB size and capacity based on our needs. This flexibility is essential for handling different works without decreasing performance. Configuring EC2 instances is important for our project because it includes selecting the right instance type to match our database's performance and storage requirements, ensuring optimal operation. With AWS, database resources can be easily scaled to meet changing

application demands and maintain consistent performance even during periods of high usage. The PostgreSQL database has high reliability, extensive backup options, and quick recovery mechanisms with AWS, guaranteeing data integrity and continuous operation. Because AWS automates repetitive database operations like backups and patches, your team can concentrate on development rather than database management. Strong data protection is ensured by the setup's advanced security features, which include identity management, network isolation, and encryption.

- Redis is used to cache accessed data as an in-memory data store. Redis lowers the requirement to access the database for requests by keeping this data in its memory, which improves response times. The backend of the Django-rest framework is integrated with Redis. It acts as a middleware between the database and the backend logic. The backend first queries Redis for the necessary information when processing a request. The data is retrieved from the Redis cache, avoiding the database if it is there. If not, the information is removed from the database and saved in Redis for quicker access to the other requests. By lowering the database load, this configuration improves the performance of our application.
- Our application is compatible with Google Chrome Version 119.0.6045.199 (Official Build) (arm64), Safari Version 17.1.2 (19616.2.9.11.12), Mozilla Firefox Version 120.0.1 (64 bit), and Opera 104.0.4944.85.
- We used Postman extensively during the development process to test every functionality locally. We used the well-known API testing tool Postman to verify the functionality of our Django REST framework-developed APIs. We were able to test API endpoints and client-server interactions and confirm responses in a range of scenarios.

5. Persistent Data Management

We used PostgreSQL as our database technology because PostgreSQL works best with the Django framework. Django provides ORM (Object Relational Mapping), which maps the object data directly to database tables, and that way, we do not have to write SQL queries ourselves. Also, most of our data is structured, so we used an SQL database instead of a NoSQL database.

User data, Product data (Secondhand, Donation, Borrow), and Entry data (Complaint, Lost and Found) are stored in the database.

We are also using Redis cache for our project. Redis is used to improve response times for requests. After the cache is created, when the same request is made by the user, the cached data is used instead of making another database query, which would be more time-consuming.

6. Subsystem Decomposition

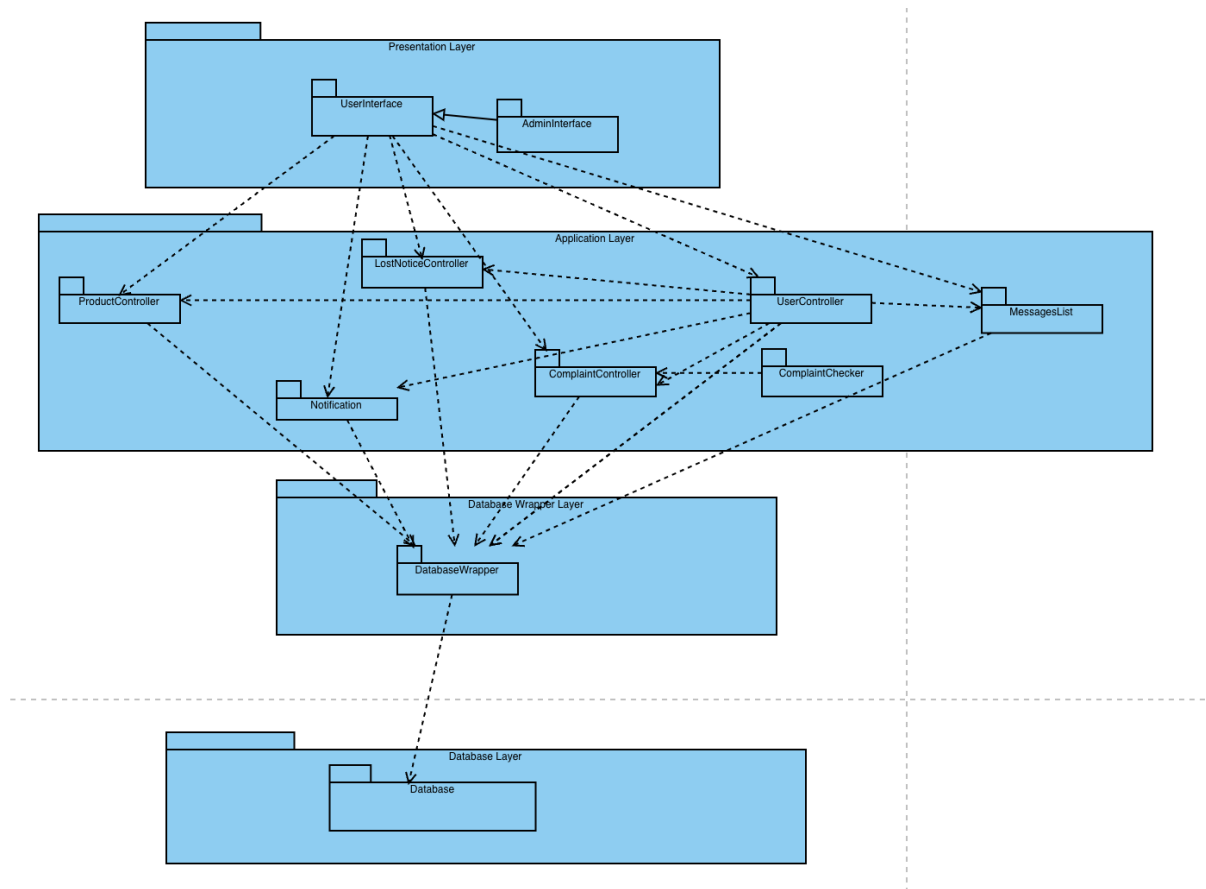


Figure 1: Subsystem Decomposition Diagram

The subsystem decomposition diagram is a detailed architecture of a layered software application. It outlines how different components within the system interact to process user requests and manage data. Our application consists of 4 layers.

6.1. Presentation Layer

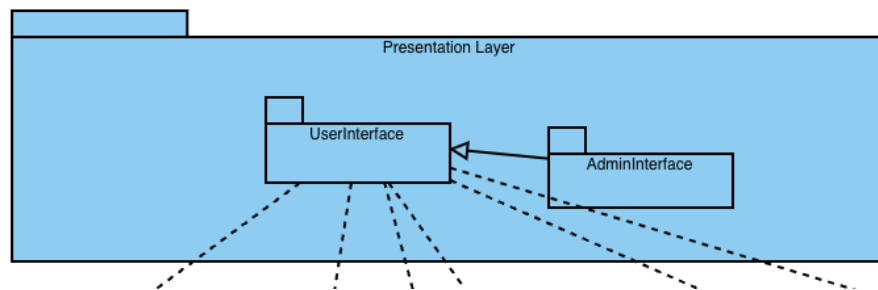


Figure 2: Presentation Layer Diagram

This is the topmost layer, typically consisting of components that handle the user interface and user experience. It includes two interfaces:

- UserInterface:** This component serves as the main interaction point for the application's end-users. It provides the graphical user interface (GUI) that users interact with.
- AdminInterface:** This is a specialized user interface designed for administrators. It provides additional controls and data views that are necessary for system maintenance and management tasks that regular users do not have access to. This interface is generalized to the UserInterface package.

6.2. Application Layer

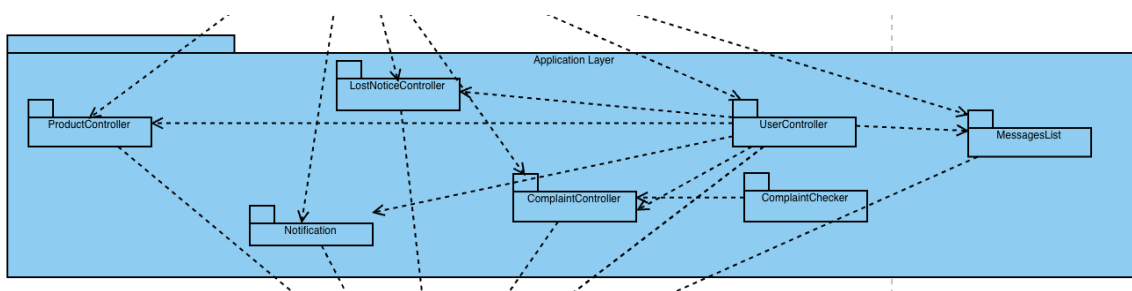


Figure 3: Application Layer Diagram

The application's business logic is contained in this layer. It executes commands from the user, performs operations, and makes logical judgments and assessments. It serves as a bridge between the database wrapper layer and the presentation layer. There are numerous controllers in this layer.

ProductController:

- Manages product-related operations.
- This package is associated with the `UserInterface`, `UserController`, and `DatabaseWrapper` packages.

Notification:

- Manages the sending of notifications to users.
- This package is associated with the `UserInterface`, `UserController`, and `DatabaseWrapper` packages.

LostNoticeController:

- This manages the lost notices in our application.
- This package is associated with the `UserInterface`, `UserController`, and `DatabaseWrapper` packages.

ComplaintController:

- This package manages user complaints.
- This package is associated with the `UserInterface`, `UserController`, and `DatabaseWrapper` packages.

UserController:

- Manages user-related operations like creating, updating, or deleting user accounts.
- This package is associated with the `UserInterface` and `DatabaseWrapper` packages.

MessagesList:

- This package manages controllers related with the Messages.
- This package is associated with the `UserInterface`, `UserController`, and `DatabaseWrapper` packages.

ComplaintChecker:

- This package consists of third-party software that checks for inappropriate complaints.
- This package is associated with the `ComplaintController` package.

6.3. Database Wrapper Layer

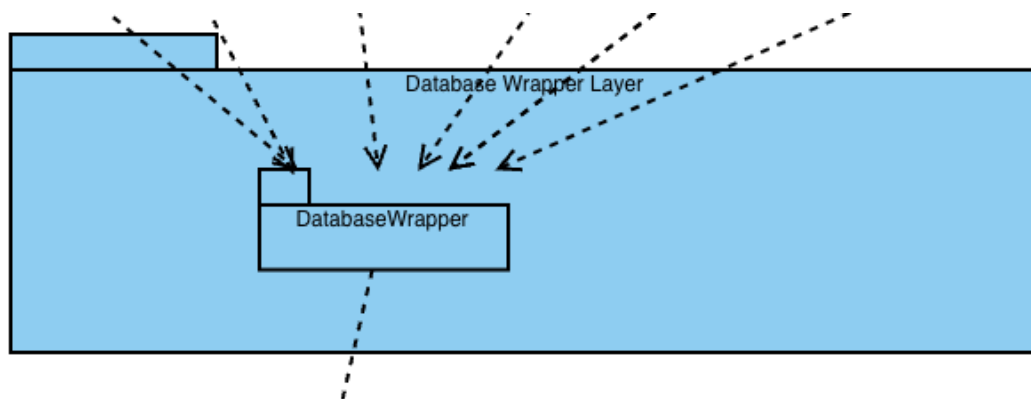


Figure 4: Database Wrapper Layer Diagram

This layer is an intermediary between the Application Layer and the Database Layer. It ensures that the Application Layer doesn't need to know the specifics of the database schema or querying language.

- **DatabaseWrapper:** It abstracts the complexity of database queries and operations from the application logic, providing a simplified and consistent interface for data access and manipulation.

6.4. DatabaseLayer

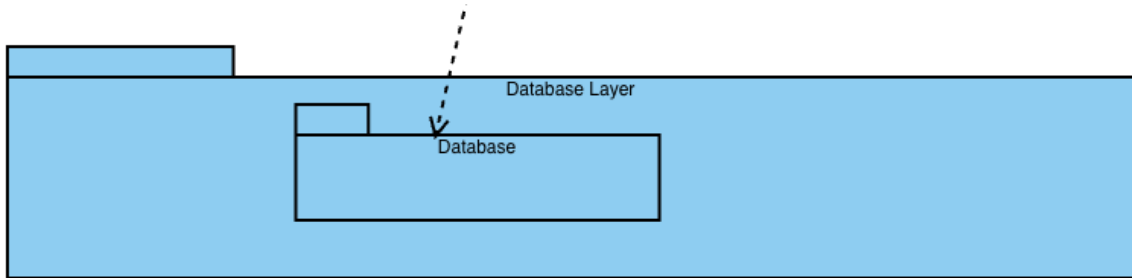


Figure 5: Database Layer Diagram

This is the data persistence layer, where the actual database resides. It is responsible for data persistence and provides mechanisms for storing, retrieving, updating, and deleting data as needed

- **Database:** The actual storage mechanism. In this project, PostgreSQL.

7. Boundary Conditions

7.1. Initialization

The main page of the Bilboard website, which offers details about the program and its capabilities, is what a user sees when they first visits. This website, which provides information on the application's goal of enabling used secondhand sales, donations, borrowing, complaints, and lost-and-found services for Bilkent University members, can be accessed without logging in or registering.

If a user prefers to log in or register, they are navigated to the authentication page. Here, JWT tokens work for secure login, password recovery, and email verification processes. For new users, the registration process includes Bilkent email verification to ensure that only university members can access the full features of the

application. Upon successful login, the user interface transitions to display the main modules in the navigation bar. These modules include options for secondhand sales, donations, borrowing, complaints, and lost & found. Each module, when selected, fetches relevant data from the PostgreSQL database hosted on AWS RDS, ensuring up-to-date information is displayed to the user.

The DjangoREST framework interacts with AWS services on the backend, specifically RDS, for managing databases. This guarantees secure, scalable, and dependable data processing. The AWS-hosted storage is set up to manage varying loads and maintain performance effectiveness. The server is continuously monitored and its status is checked regularly. Different features and access levels are offered based on the user's role (admin or regular user). This initialization phase establishes expanded control and access for administrators within the program.

7.2. Termination

Bilboard application has several different termination scenarios. The first and most common scenario is user logouts using the provided interface. When the user clicks on the log-out button, their access and refresh tokens will expire by log-out logic provided by the backend, and their session will end.

The second scenario is if users do not log out but do not show an activity refresh token will expire, and they need to log in again to the application because their session will end after the predefined time interval. Refresh token creates access token in a predefined and relatively shorter period, so if the user loses their refresh token, they will not be able to have an access token and lose their access to their session.

Also, we want to make sure that data is saved to our database and system ensures that all changes are applied if there are any. For instance, we will ensure that

if a user in a change information page changes personal information or updates product details and they try to log out without saving or canceling the process, we will ask them to ensure their action.

The third scenario may be if there is a security problem or maintenance and renovation issue, the admin may terminate the application for a while, notifying users beforehand.

The last and most unwanted scenario is unexpected error cases if there is a failure system that terminates the application.

7.3. Failure

Our database uses the AWS PostgreSQL. Therefore, if a failure occurs due to AWS or the system encounters any unexpected error, immediately ceases the CRUD operations of the database to prevent any possible data loss. The system regularly takes backups of databases. In this way, possible data losses are prevented.

Since it is a web-based application, there might be some connectivity issues in our database, PostgreSQL, Billboard will try to reconnect and retrieve the necessary data. Users may be shown with a notification like a pop-up about the temporary failure of the database and recommended to retry after a short period.