Software Architecture Document

Version 7.0 for

Biblio

Prepared by

Name	**Student ID**	**Email**
Abdulla Alhaj Zin	40013496	abdullah.hzen@gmail.com
Dave Bhardwaj	40000679	davebhardwaj1@gmail.com
Giovanni Gebran	40018637	ggebran95@gmail.com
Kayne Herrmann	40007153	kh_group@outlook.com
Kenza Boulisfane	40043521	kenza.boulisfane@gmail.com
Kevin Camellini	26771009	kevincamellini@gmail.com
Kevin Lin	40002383	Lin_Kevin_1995@hotmail.com
Kevin Yau	27058276	kevin.yau@outlook.com
Nizar Belhassan	27519443	M.nizar.belhassan@gmail.com
Nour El Natour	40013102	nour_elnatour@hotmail.com

Instructor:	Dr. C. Constantinides
Course:	Software Architecture and Design I
Date:	11/23/2018

Document history

Date	Version	Description	Author
09/26/2018	<1.0>	SAD 1.0	Team 5
09/28/2018	<1.5>	SAD 1.5	Team 5
10/11/2018	<2.0>	SAD 2.0	Team 5
10/12/2018	<2.5>	SAD 2.5	Team 5
10/17/2018	<3.0>	SAD 3.0	Team 5
10/22/2018	<3.5>	SAD 3.5	Team 5
10/26/2018	<4.0>	SAD 4.0	Team 5
05/11/2018	<4.5>	SAD 4.5	Team 5
10/11/2018	<5.0>	SAD 5.0	Team 5
11/11/2018	<5.5>	SAD 5.5	Team 5
11/20/2018	<6.0>	SAD 6.0	Team 5
11/21/2018	<6.5>	SAD 6.5	Team 5
11/23/2018	<7.0>	SAD 7.0: Release Version	Team 5

Table of contents

List of figures

1. Introduction

The following Software Architecture Document provides an architectural overview of the **Biblio** system. It can be regarded as a a map, as it allows the reader to understand how the system is structured, by using different architectural views. This introduction includes the purpose, scope, definitions, acronyms, abbreviations, references, and overview of the system.

1.1. Purpose

The purpose of this documented software is to build an online system to manage a library catalog that includes printed materials such as books and magazines, as well as unprinted materials such as musics and movies. The system is implemented within the Concordia University information system ecosystem and serves as a means of facilitating the management of a library catalog. This document mentions all the architectural diagrams in details of this web application so that its behavior would be represented in forms of design diagrams. This project has been implemented under the guidance of Dr. C. Constantinides, and is intended for use by librarians (administrators) as well as users subscribed to the library.

1.2. Scope

Biblio is a web application built to facilitate the management of a library catalog and create a handy and simple application that is useful to both library workers and the members of the library. The following document focuses on all the possible architectural views used to build this library system, as well as all the nonfunctional and the designs of the functional requirements that the web application is based upon. It also provides a list of the major quality attributes of the system as described in the ISO-9126 (I) standard.

1.3. Definitions, Acronyms and Abbreviations

- SAD: Software Architecture Document.
- Administrator: user that can alter the database, and manage users and transactions.
- Registered user: a user who can borrow items from the library catalog.
- CRUD: Create, read, update or delete.
- Heroku: Cloud platform used to build and monitor web applications.
- PostgreSQL: Open source object-relational database system.
- TDG: Table Data Gateway.
- Uow: Unit of Work.
- IMAP: Identity Mapper.
- Operational errors: errors stemming from erroneous user input.
- OCL: Object Constraint Language.
- Node.JS: runtime based on JavaScript.
- HTML: Hypertext Markup Language.
- CSS: Cascading Style Sheets.

- ISO-9126: International Standard for product quality.

2. Architectural Representation

In order to be able to document, implement and deploy a system successfully, it is necessary to view it from a number of different perspectives. In this document, the **Biblio** system will be represented in a 4+1 view approach, shown in Figure 1, to ensure a deep understanding of its architecture, and clear up any ambiguity and misconceptions. The following views are used:

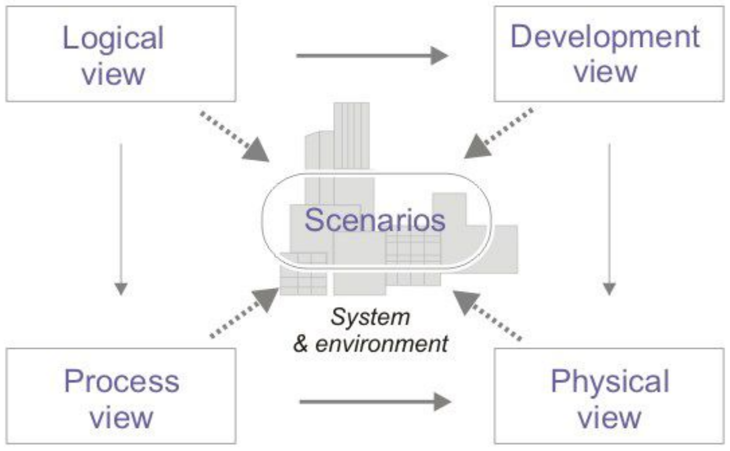


Figure 1: 4 + 1 view model

1. Logical View

In this view, the intended audience is the designers. Interaction and class diagrams are used to demonstrate the functionalities provided to end-users by the web application.

2. Development View

In this view, the intended audience is the programmers. The UML diagram represents the implementation of the software management from the programmers' point of view by providing a package diagram. This view is also called the Implementation view which depicts the physical composition of the implementation. The implementation elements include the source code data, the directories and the files.

3. Process View

In this view, the intended audience is the integrators. The activity diagram is, which is a UML diagram, demonstrates the communication between different system processes as well as the system's behaviour at run-time.

4. Physical View

In this view, the intended audience is the deployment managers. The deployment diagram, which is a UML diagram, describes the physical interactions that happen amongst the system's physical components. In **Biblio**'s case, it is the interaction between the web-application, the server, and the database.

5. Use Case View

In this view, the intended audience is all the stakeholders of the system. Three use case scenarios are provided to show the sequence of interactions between all processes and objects in the system in order to explain the behavior of the system.

6. Data View

In this view, the intended audience is the data specialists and the database administrators. The data model represents the persistent elements used in the database.

7. Architectural Requirements

a. Goals

The **Biblio** architecture has been designed to fulfill the following objectives:

- Facilitate the management of a library catalog for library administrators.
- Help users acquire books and media for a specific amount of time, provided in a user friendly environment with ease of navigability.
- Meet the ISO-9126 quality standards.
- Improve functionality offered by the system.
- Expose the structure of the system while hiding any implementation details.

b. Constraints

There are many design and implementation constraints for the system being built, the main ones being:

- **User Interaction Simplicity:** The user is non-technical, however is familiar with popular internet technologies and platforms such as Amazon, Netflix and Kijiji. Hence, the end user should be able to navigate and utilize all of the software's features with ease.
- **Security:** The system should encrypt all of the users data and activity. In addition, the system should be protected against SQL and URL injections. Thus, the privacy of users and the integrity of the system should be a main priority.
- **Scalability:** The architecture of the system should be implemented in such a way that it allows the system to scale without major architectural modifications. In other words, the architecture, design patterns, database and code should be chosen carefully in order to allow component reuse and traffic growth.
- **Availability:** The architecture should allow the system to be continuously accessible to the users. In fact, there should be very little down time, even during peak system usage. Hence, down time should only be present in the case of a planned system enhancement or modification.

Functional Requirements (Use Case View)

The following table only references the functional requirements that are deemed to be in direct relation to the architecture of the system, as they are considered to have central functionality in the system.

Source	Name	Architectural Relevance	Addressed in
UC1	CRUD	The first main feature of the system which allows the admin to create, read, delete, and update a catalog item using the TDG, IMAP, and UoW.	SRS section 3.3.1
UC2	Search and Filter	The second main feature of the system which allows the end-user of the system to search and filter the items of the	SRS Section 3.3.2

		catalog using the TDG, IMAP, and UoW.	
UC3	Loan	The third main feature of the system which allows the end-user of the system to loan items from the library using the TDG, IMAP, and UoW.	SRS Section 3.3.3

Non Functional Requirements

The following section touches upon the non functional requirements that have a direct connection to the architecture of the system.

Source	Name	Architectural Relevance	Addressed in
SRS	Performance efficiency	The system shall respond to user input quickly.	SRS section 3.4.1
SRS	Compatibility	The system shall work with any operating system the user chooses.	SRS section 3.4.2
SRS	Usability	The system shall have a user-friendly interface with clear descriptive links.	SRS section 3.4.3
SRS	Reliability	- The system shall be usable while handling errors. - The system shall not crash if errors arise.	SRS section 3.4.4
SRS	Security	- The system shall block any unauthorized attempts at usage while still providing services to legitimate users. - Administrators shall not have access to users' passwords.	SRS section 3.4.5

8. Use Case View (Scenarios)

Use Case 1 Summary

An administrator logs into the system. He creates a new item. Then, he views the item in the Catalog. The administrator then modifies the title by pressing the update button next to the corresponding item. After modifying, he deletes the newly modified item.

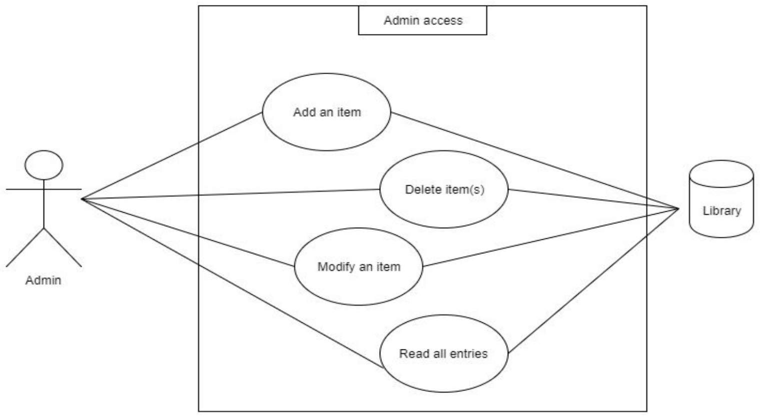


Figure 2: CRUD Use Case Model

Use Case 2 Summary

A user (includes administrator) may filter the catalog page by selecting one or multiple sorting criteria. The user is redirected to the Catalog page with the filtered items. The user may enter a search query in the header of the view. The user is then redirected to the Catalog page with items matching the user's search query. The user may filter the search output of the items found in the view.

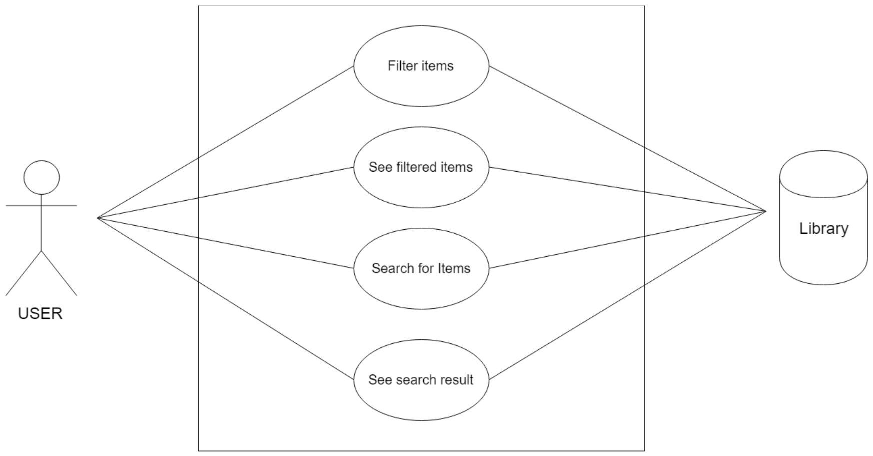


Figure 3 - Search and Filter Use Case Model

Use Case 3 Summary

A user logs into the system. The user visits the Catalog, views the details of the first item and loans the item. The user then goes on his transactions page, and returns the item he borrowed.

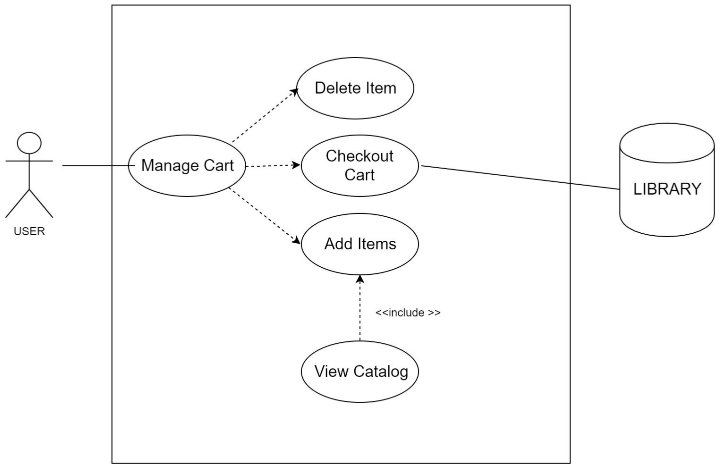


Figure 4: Loan Use Case Model

9. Logical View

Layers

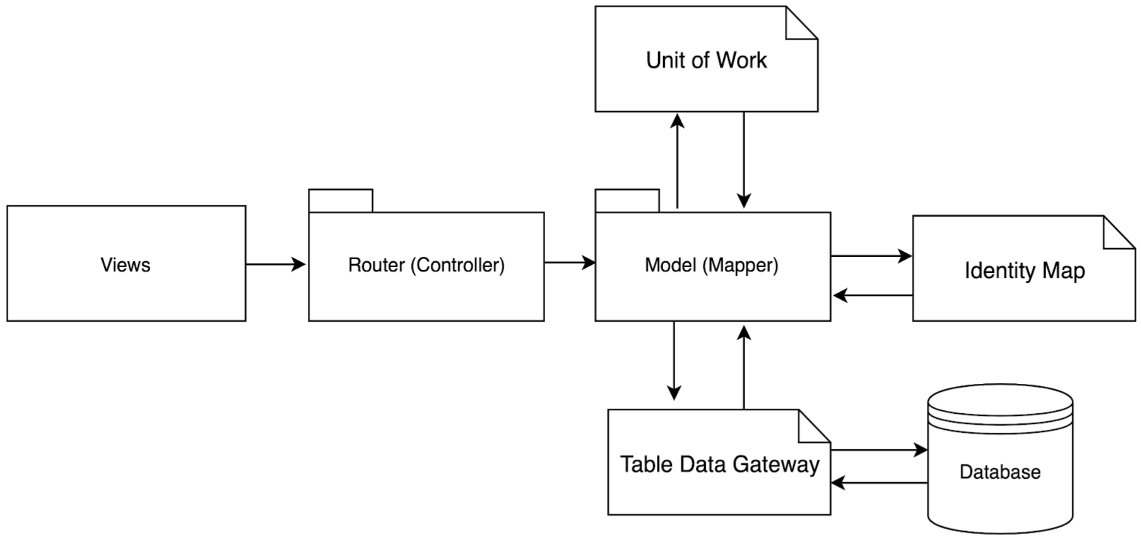


Figure 5: Class Diagram Layers

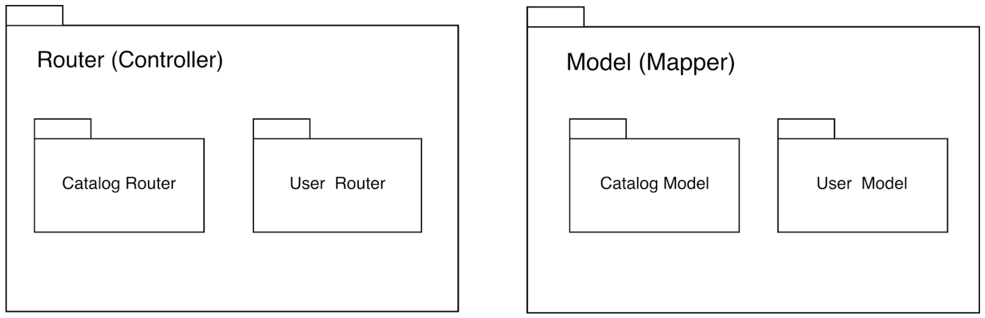


Figure 6: Router and Model Overview of Catalog and User

Z-Specification

ITEM, NUMBER, REPORT

InitLoanItems

$$\begin{array}{|l}\hline LoanedItems \\ \hline LoanedItems = \varnothing \\ \hline\end{array}$$

LoanedItems

$$\begin{array}{|l}\hline Catalog : \mathbb{P}\, ITEM \\ LoanedItems : \mathbb{P}\, ITEM \\ \hline LoanedItems \subseteq Catalog \\ \hline\end{array}$$

LoanItem

$$\begin{array}{|l}\hline \Delta LoanedItems \\ Item? : ITEM \\ \hline Item? \in Catalog \\ Item? \notin LoanedItems \\ LoanedItems' = LoanedItems \cup \{Item?\} \\ \hline\end{array}$$

REPORT ::= ok | reachedMaxLoan

Success

$$\begin{array}{|l}\hline result! : Report \\ \hline result! = ok \\ \hline\end{array}$$

MaxLoanedItems

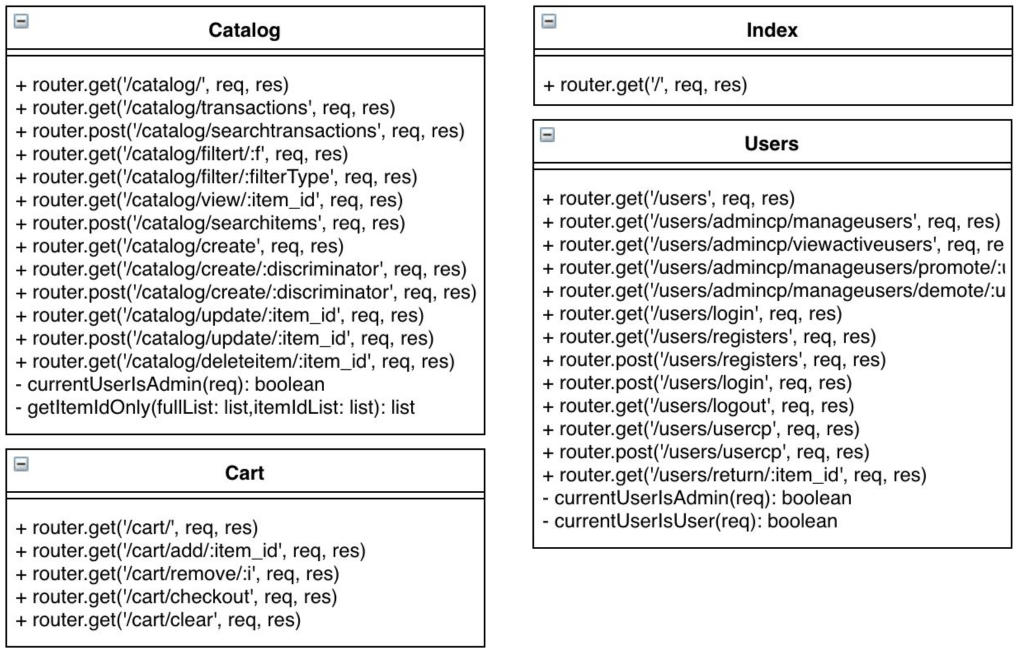
$$\begin{array}{|l}\hline \Xi LoanedItems \\ maxNumber? : NUMBER \\ LoanedItems : ITEM \\ result! : REPORT \\ \hline LoanedItems \leq 5 \\ result! = reachedMaxLoan \\ \hline\end{array}$$

(LoanItem $\wedge$ Success) $\vee$ MaxLoanedItems

Figure 7: Z Specification Diagrams to Loan an Item

Class Diagrams

Controller Class Diagram



Catalog

```
+ router.get('/catalog/', req, res)
+ router.get('/catalog/transactions', req, res)
+ router.post('/catalog/searchtransactions', req, res)
+ router.get('/catalog/filtert/:f', req, res)
+ router.get('/catalog/filter/:filterType', req, res)
+ router.get('/catalog/view/:item_id', req, res)
+ router.post('/catalog/searchitems', req, res)
+ router.get('/catalog/create', req, res)
+ router.get('/catalog/create/:discriminator', req, res)
+ router.post('/catalog/create/:discriminator', req, res)
+ router.get('/catalog/update/:item_id', req, res)
+ router.post('/catalog/update/:item_id', req, res)
+ router.get('/catalog/deleteitem/:item_id', req, res)
- currentUserIsAdmin(req): boolean
- getItemIdOnly(fullList: list,itemIdList: list): list
```

Cart

```
+ router.get('/cart/', req, res)
+ router.get('/cart/add/:item_id', req, res)
+ router.get('/cart/remove/:i', req, res)
+ router.get('/cart/checkout', req, res)
+ router.get('/cart/clear', req, res)
```

Index

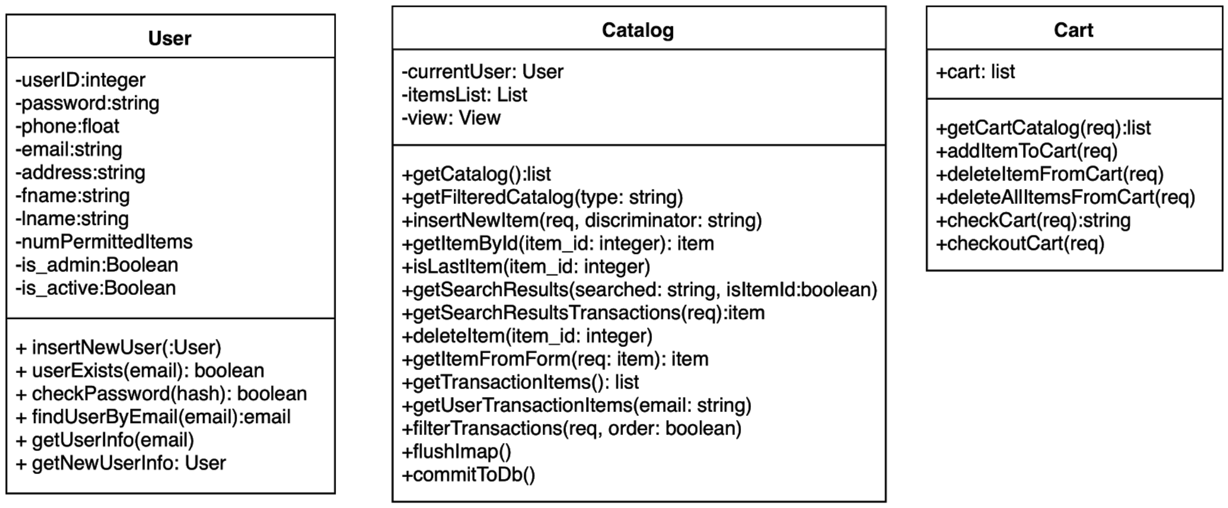
```
+ router.get('/', req, res)
```

Users

```
+ router.get('/users', req, res)
+ router.get('/users/admincp/manageusers', req, res)
+ router.get('/users/admincp/viewactiveusers', req, re
+ router.get('/users/admincp/manageusers/promote/:u
+ router.get('/users/admincp/manageusers/demote/:u
+ router.get('/users/login', req, res)
+ router.get('/users/registers', req, res)
+ router.post('/users/registers', req, res)
+ router.post('/users/login', req, res)
+ router.get('/users/logout', req, res)
+ router.get('/users/usercp', req, res)
+ router.post('/users/usercp', req, res)
+ router.get('/users/return/:item_id', req, res)
- currentUserIsAdmin(req): boolean
- currentUserIsUser(req): boolean
```

Figure 8: Controller Class Diagram

Models Class Diagram



User

```
-userID:integer
-password:string
-phone:float
-email:string
-address:string
-fname:string
-lname:string
-numPermittedItems
-is_admin:Boolean
-is_active:Boolean
```

```
+ insertNewUser(:User)
+ userExists(email): boolean
+ checkPassword(hash): boolean
+ findUserByEmail(email):email
+ getUserInfo(email)
+ getNewUserInfo: User
```

Catalog

```
-currentUser: User
-itemsList: List
-view: View
```

```
+getCatalog():list
+getFilteredCatalog(type: string)
+insertNewItem(req, discriminator: string)
+getItemById(item_id: integer): item
+isLastItem(item_id: integer)
+getSearchResults(searched: string, isItemId:boolean)
+getSearchResultsTransactions(req):item
+deleteItem(item_id: integer)
+getItemFromForm(req: item): item
+getTransactionItems(): list
+getUserTransactionItems(email: string)
+filterTransactions(req, order: boolean)
+flushImap()
+commitToDb()
```

Cart

```
+cart: list
```

```
+getCartCatalog(req):list
+addItemToCart(req)
+deleteItemFromCart(req)
+deleteAllItemsFromCart(req)
+checkCart(req):string
+checkoutCart(req)
```

Figure 9: Models Class Diagram

Table Data Gateway Class Diagram

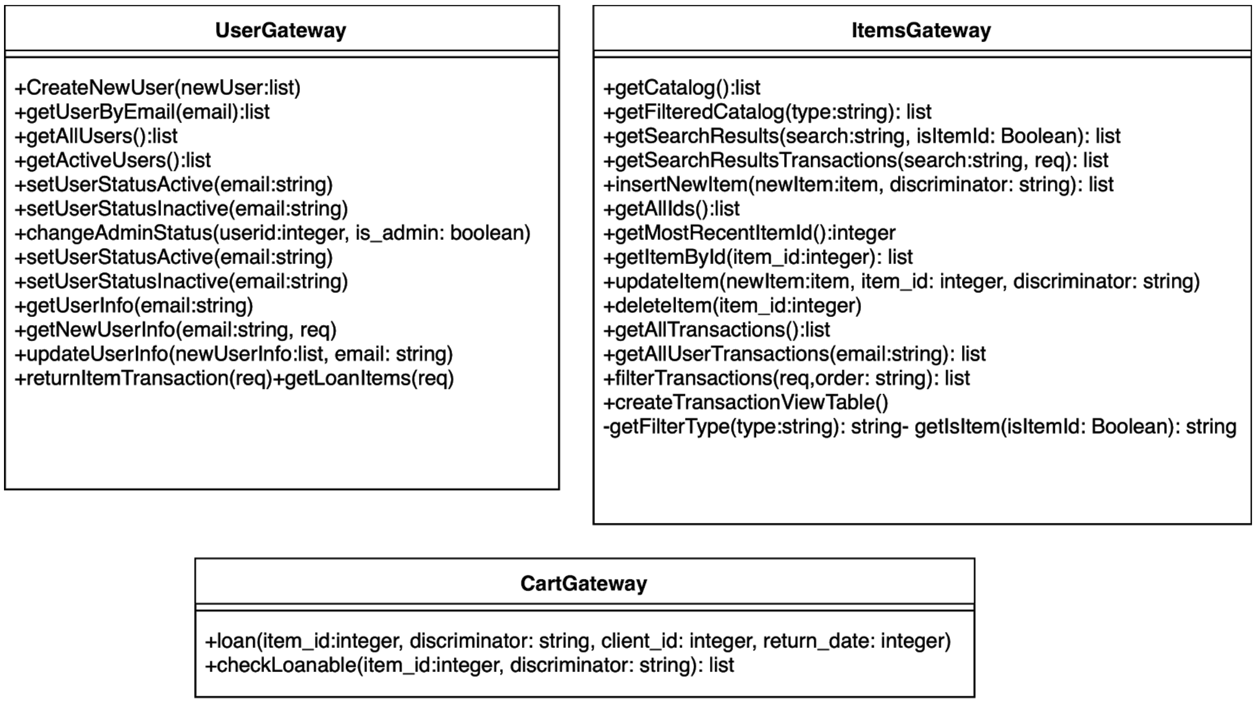

UserGateway
+CreateNewUser(newUser:list)
+getUserByEmail(email):list
+getAllUsers():list
+getActiveUsers():list
+setUserStatusActive(email:string)
+setUserStatusInactive(email:string)
+changeAdminStatus(userid:integer, is_admin: boolean)
+setUserStatusActive(email:string)
+setUserStatusInactive(email:string)
+getUserInfo(email:string)
+getNewUserInfo(email:string, req)
+updateUserInfo(newUserInfo:list, email: string)
+returnItemTransaction(req)+getLoanItems(req)
ItemsGateway
+getCatalog():list
+getFilteredCatalog(type:string): list
+getSearchResults(search:string, isItemId: Boolean): list
+getSearchResultsTransactions(search:string, req): list
+insertNewItem(newItem:item, discriminator: string): list
+getAllIds():list
+getMostRecentItemId():integer
+getItemById(item_id:integer): list
+updateItem(newItem:item, item_id: integer, discriminator: string)
+deleteItem(item_id:integer)
+getAllTransactions():list
+getAllUserTransactions(email:string): list
+filterTransactions(req,order: string): list
+createTransactionViewTable()
-getFilterType(type:string): string- getIsItem(isItemId: Boolean): string
CartGateway
+loan(item_id:integer, discriminator: string, client_id: integer, return_date: integer)
+checkLoanable(item_id:integer, discriminator: string): list

Figure 10: Table Data Gateway Class Diagram

Identity Mapper Class Diagram

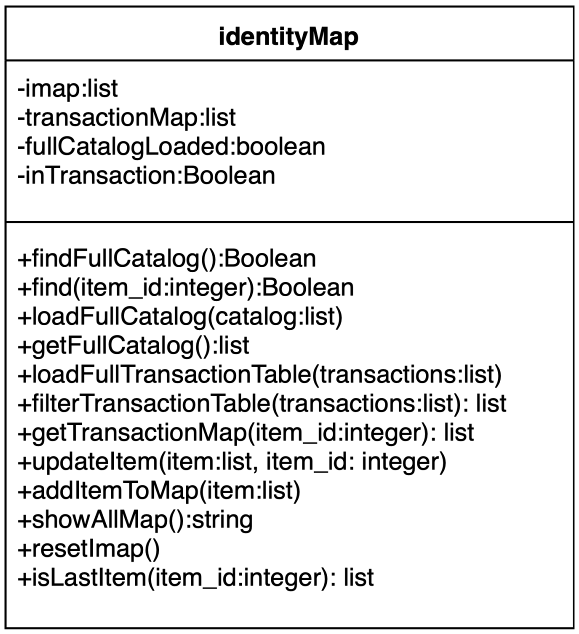

identityMap
-imap:list
-transactionMap:list
-fullCatalogLoaded:boolean
-inTransaction:Boolean
+findFullCatalog():Boolean
+find(item_id:integer):Boolean
+loadFullCatalog(catalog:list)
+getFullCatalog():list
+loadFullTransactionTable(transactions:list)
+filterTransactionTable(transactions:list): list
+getTransactionMap(item_id:integer): list
+updateItem(item:list, item_id: integer)
+addItemToMap(item:list)
+showAllMap():string
+resetImap()
+isLastItem(item_id:integer): list

Figure 11: Identity Mapper Class Diagram

Unit Of Work Class Diagram

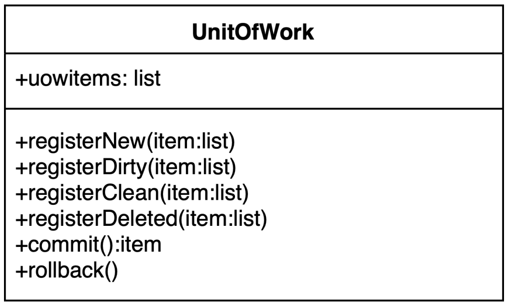

UnitOfWork
+uowitems: list
+registerNew(item:list)
+registerDirty(item:list)
+registerClean(item:list)
+registerDeleted(item:list)
+commit():item
+rollback()

Figure 12: Unit of Work Class Diagram

Object Constraint Language (OCL)

“A user can have only one account, hence a user has a unique id and email.”

```
context User
     inv: allInstances() -> isUnique(email)
     inv: allInstances() -> isUnique(userID)
```

“A user can only view his/her own transactions.”

```
context User
     inv:  self.transactions  ->  forAll(t1,t2:  Transactions  |
     t1<>t2 implies t1.userID <> t2.userID)
```

“A cart is unique to a client”

```
context Cart
     inv: allInstances() -> isUnique(user)
```

“No client or administrator may be logged in more than once at any given time.”

```
context User
     inv: self.session_id -> size() = 1
```

“A user cannot loan a magazine.”

```
context User
     inv: self.transactions -> collect(magazine)
                             -> isEmpty()
```

“An administrator cannot perform a loan or a return.”

```
context User
```

```
        inv: self.is_admin = true
        implies
                self.transactions.loan -> size() = 0
                and
                self.transactions.return -> size() = 0
```

“Admins can demote/promote regular users.”

```
context User :: toggleAdminStatus(user: User)
        pre: user.isAdmin = false
        post: @pre = true

context User :: toggleAdminStatus(user: User)
        pre: user.isAdmin = true
        post: @pre = false
```

“The system maintains a registry of its active users through a collection of pairs of id and timestamp. Administrators have access to this registry but cannot modify it.”

```
context Transaction
        inv: userId <> null
        inv: loanDate <> null
```

“There must always exist at least one registered admin in the system.”

```
context User
        inv: self.isAdmin -> size () >= 1
```

“While viewing the catalog or performing a search, a client may select an item to be placed on a cart.”

```
context User :: addItem()
        pre: self.cart -> size() = 0
        post: self.cart -> size() = @pre + 1
```

“A printed item can be loaned for 1 week, and a media item can be loaned for only for 2 days.”

```
context Book
        inv: self.loanPeriod = 7
context Movie
        inv: self.loanPeriod = 2
context Music
        inv: self.loanPeriod = 2
```

“There is a limit to the maximum number of items a user may borrow.”

```
context Cart
     inv: self.loanedItems -> size() >=0 and <=5
```

“An administrator can enter new records into the database, modify existing records, or delete existing records.”

```
context User :: insertNewItem()
     inv: self.isAdmin = true
     pre: self.catalog -> size() >= 0
     post: self.catalog -> size() = @pre+1

context User :: updateItem()
     inv: self.isAdmin = true
     pre: self.catalog -> size()
     post: self.catalog -> size() = @pre

context User :: deleteItem()
     inv: self.isAdmin = true
     pre: self.catalog -> size() > 0
     post: self.catalog -> size() = @pre -1
```

“Administrators can access, view the history of transactions.”

```
context User :: getTransactionItems()
     inv: self.isAdmin = true
     inv: self.transactions -> collect (item)
```

Subsystems

Table Data Gateway

The Table Data Gateway (TDG) contains all SQL statements, which are utilized throughout the project. Thus, open query modifications, only one file file need to be changed instead of modifying many different files. In fact, the TDG objects receives messages from the mapper and then communicates with the database, which will execute the required SQL queries and then return a result. Hence, the TDG acts as an middle man in order to reduce coupling and increase cohesion.

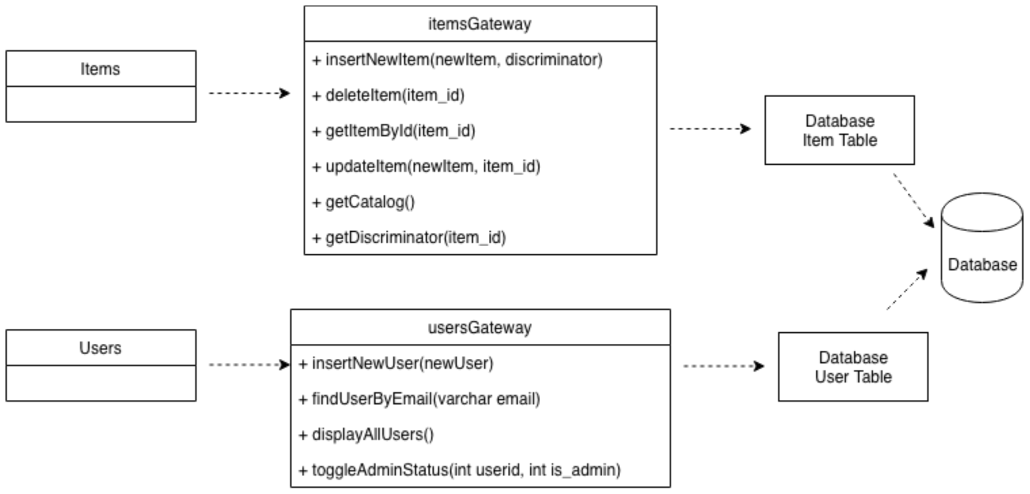
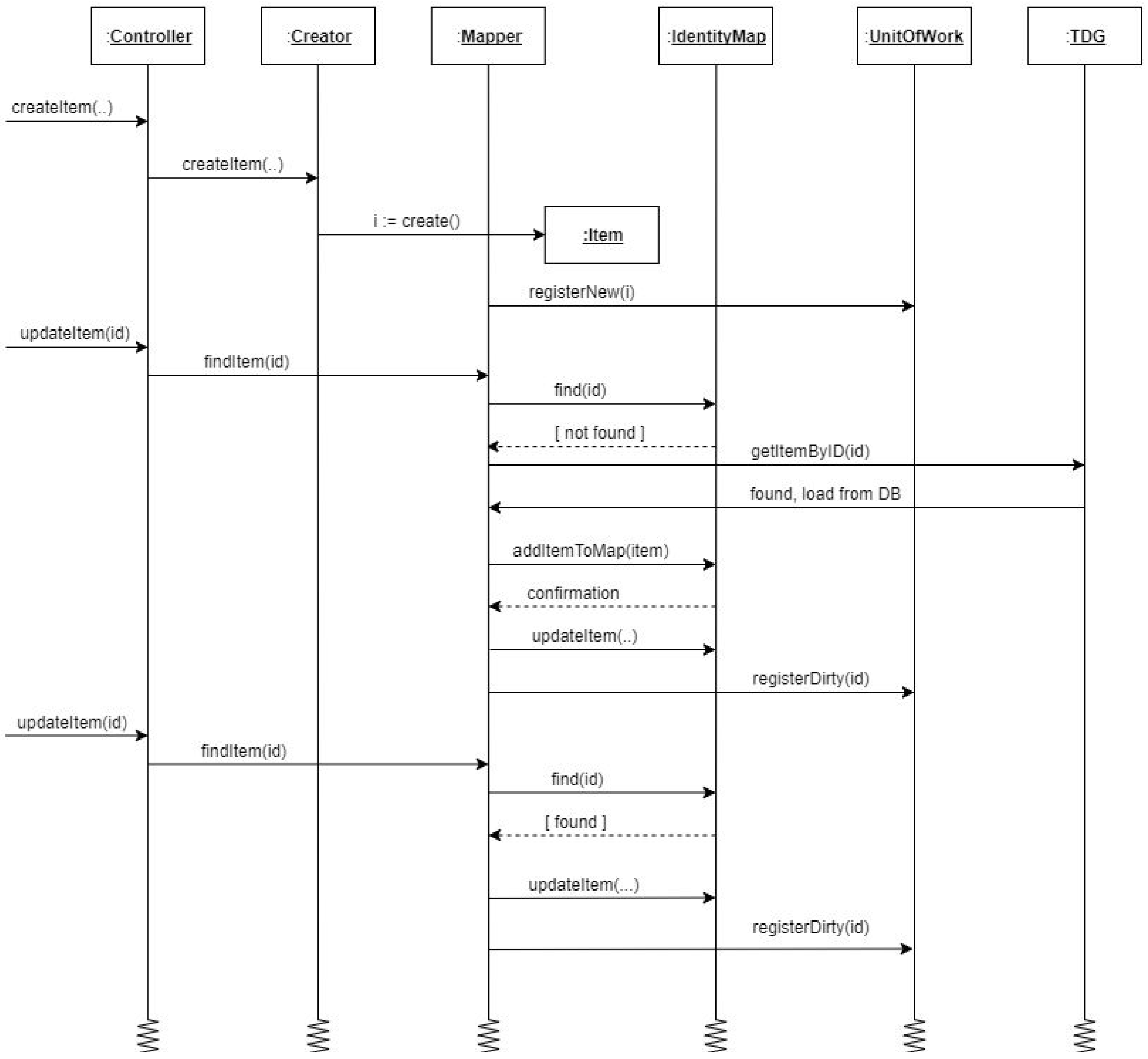


Figure 13: Table Data Gateway Decomposition

Identity Mapper

Figure 13: Identity Map for CRUD items System Sequence Diagram (1/2)



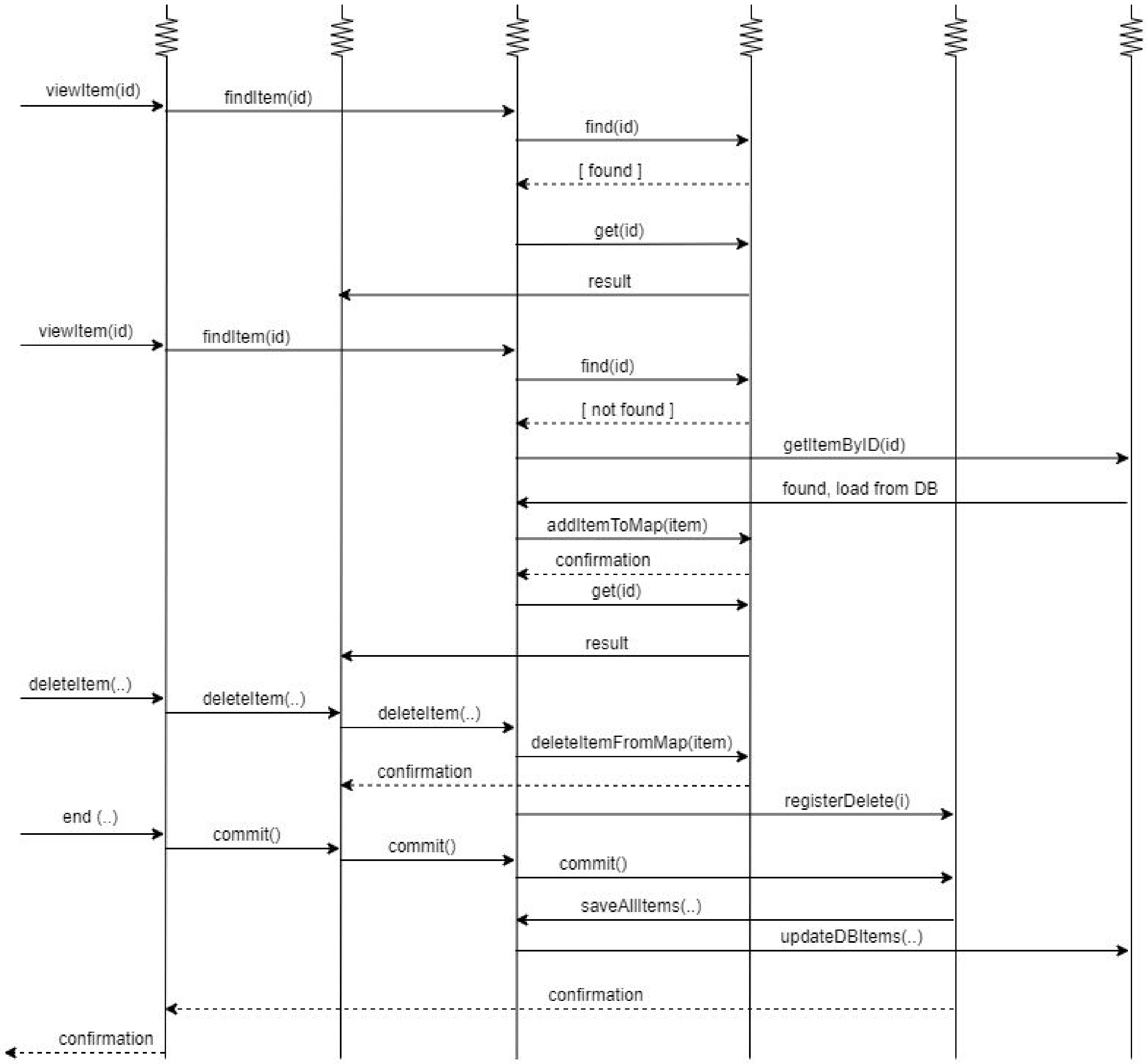


Figure 14: Identity Map for CRUD items System Sequence Diagram (2/2)

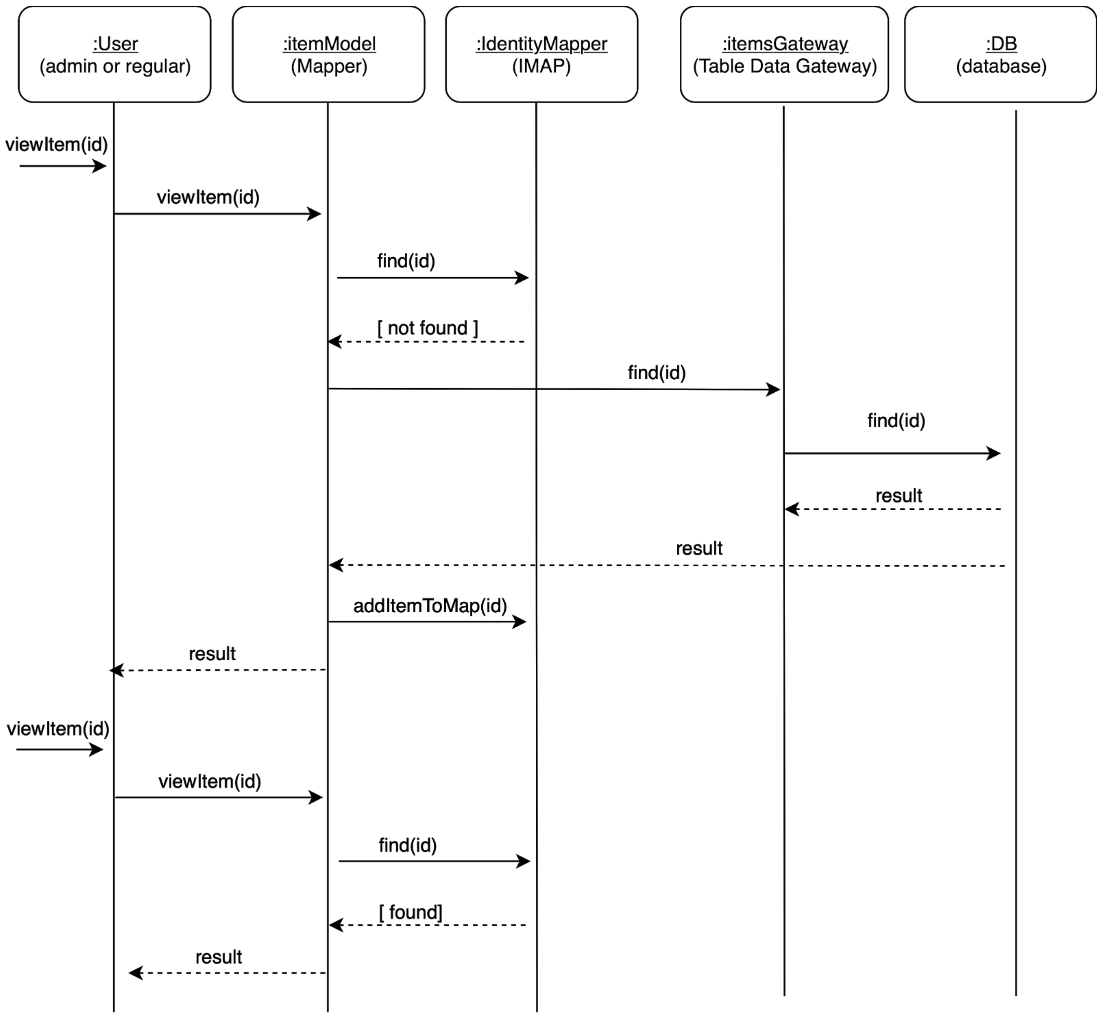


Figure 15: Identity Map Viewing Single Items System Sequence Diagram

Unit of Work

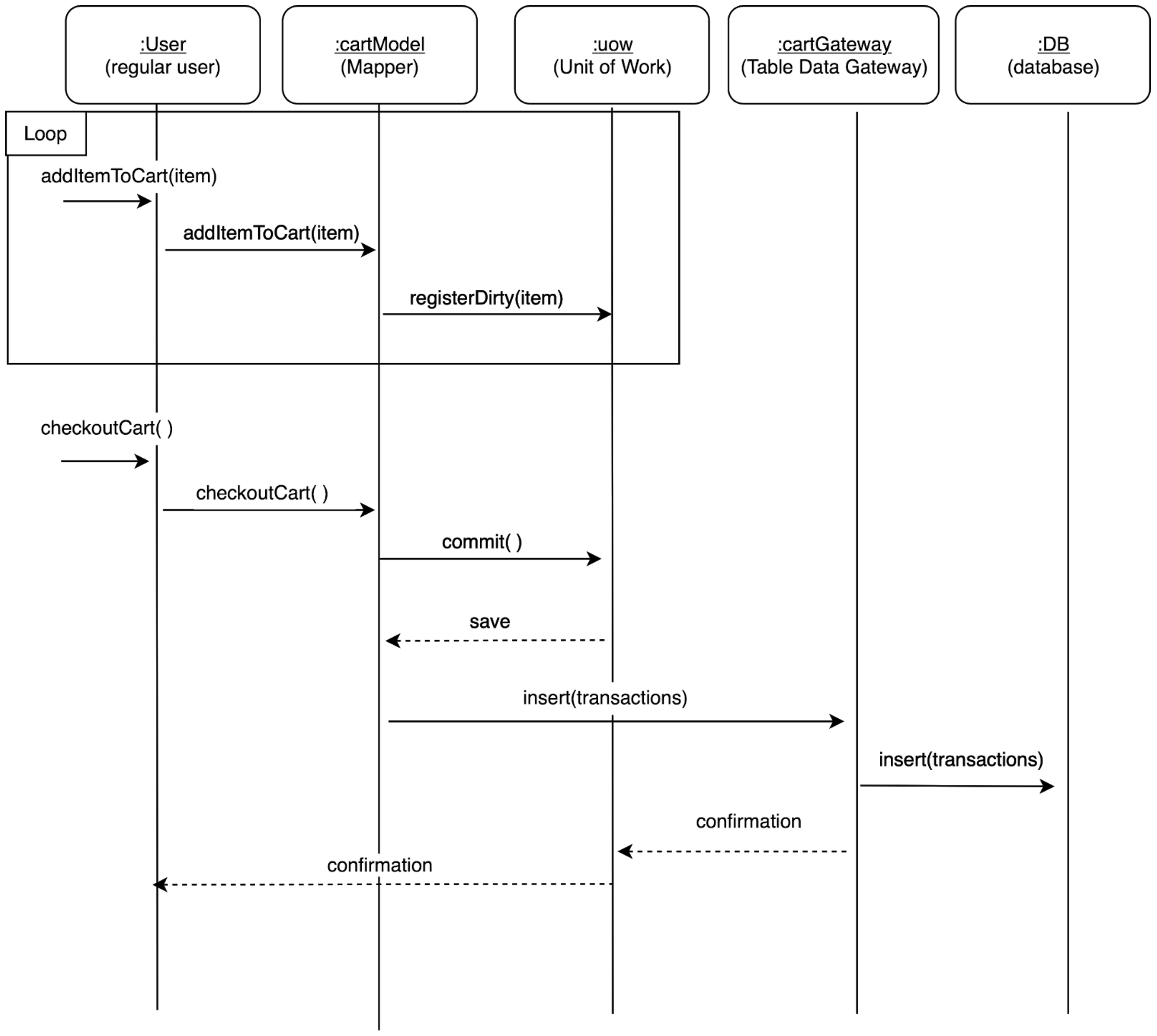


Figure 16: Unit of Work Loaning Items System Sequence Diagram

Use Case Realizations

Interaction Diagrams

1. createItem Operation:

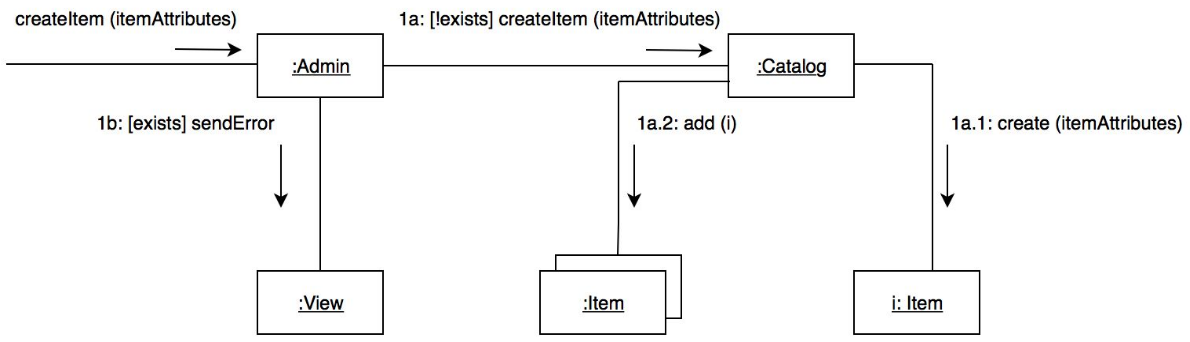

createItem (itemAttributes)
1a: [!exists] createItem (itemAttributes)
:Admin
:Catalog
1b: [exists] sendError
1a.2: add (i)
1a.1: create (itemAttributes)
:View
:Item
i: Item

Figure 17: Create Item interaction Diagram

2. readItemList Operation:

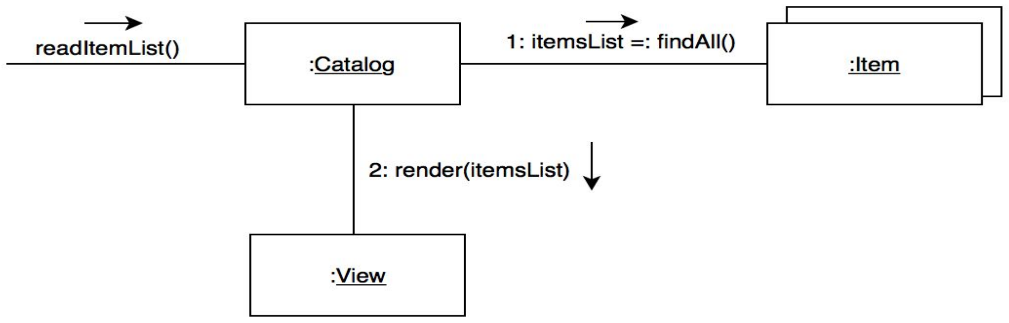


Figure 18: View item Interaction Diagram

3. updateItem Operation:

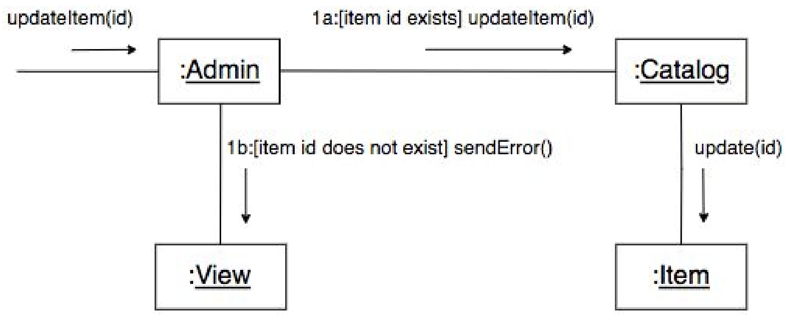

updateItem(id)
1a:[item id exists] updateItem(id)
:Admin
:Catalog
1b:[item id does not exist] sendError()
update(id)
:View
:Item

Figure 19: Update Item Interaction Diagram

4. deleteItem Operation:

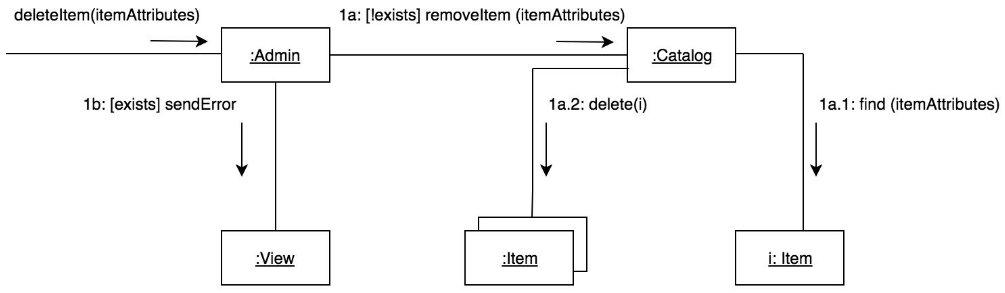

deleteItem(itemAttributes)
1a: [!exists] removeItem (itemAttributes)
:Admin
:Catalog
1b: [exists] sendError
1a.2: delete(i)
1a.1: find (itemAttributes)
:View
:Item
i: Item

Figure 20: Delete Item Interaction Diagram

5. searchItem Operation:

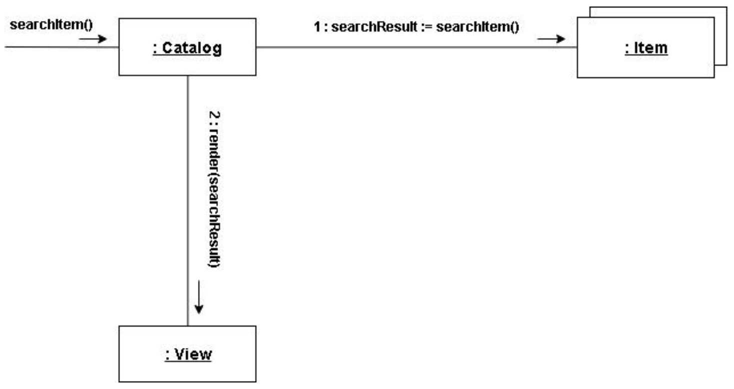


Figure 21: Search Item Interaction Diagram

6. filterItems Operation:

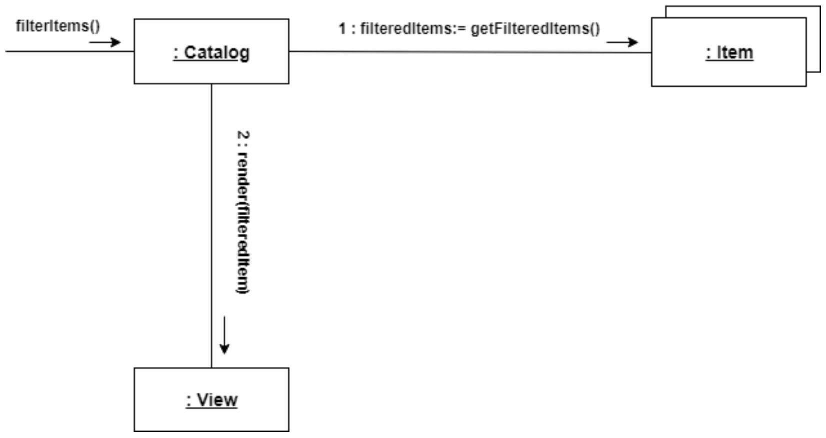


Figure 22: Filter Items Interaction Diagram

7. addItemToCart Operation:

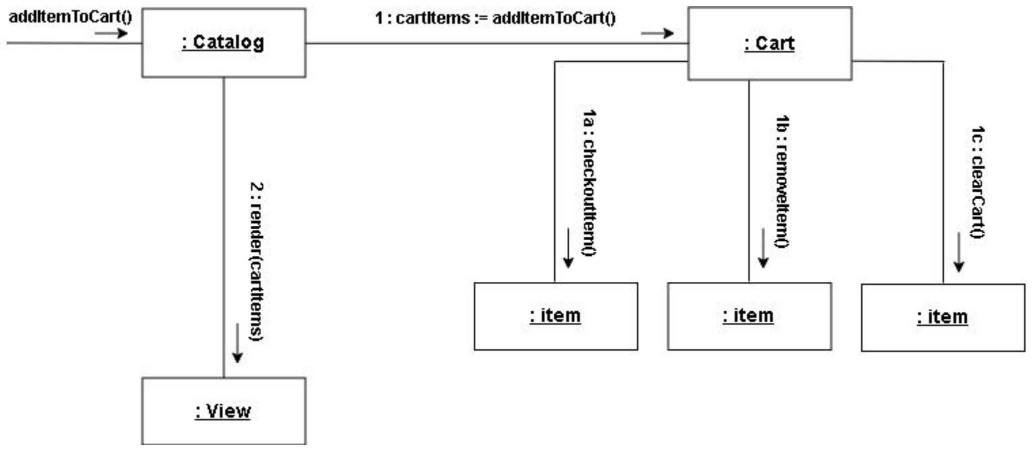


Figure 23: Add to Cart Interaction Diagram

8. loanItem Operation:

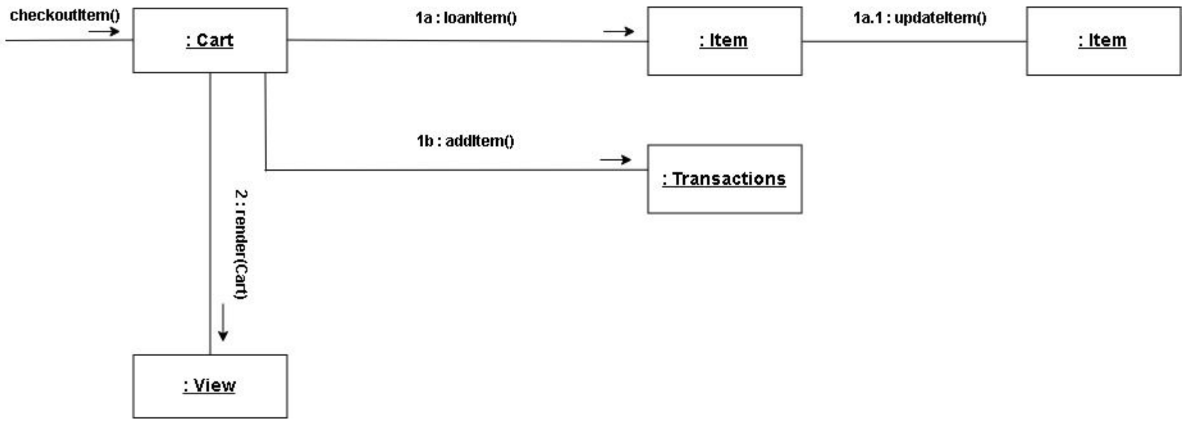


Figure 24: Loan item Interaction Diagram

Architecture Diagram

The architecture diagram shown below in Figure 24 illustrates the high level interaction of the different components of the software system.

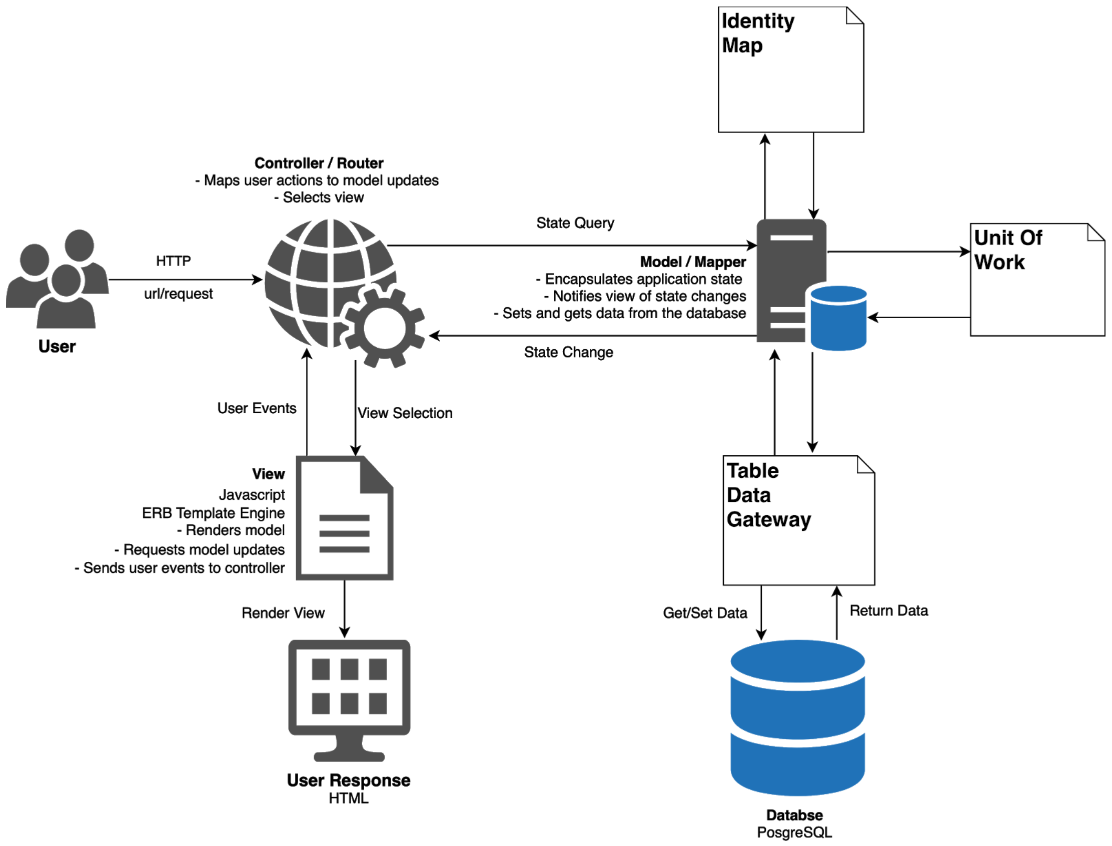


Figure 25: MVC Architecture Diagram

10.Process View

The web-application is using Node.js which is based on a single thread that serves the whole website. Therefore, everything runs in parallel on the same thread with synchronization that is built-in in the framework. The following activity diagram shows an example of a parallel flow for an activity in the web-application.

Activity Diagram

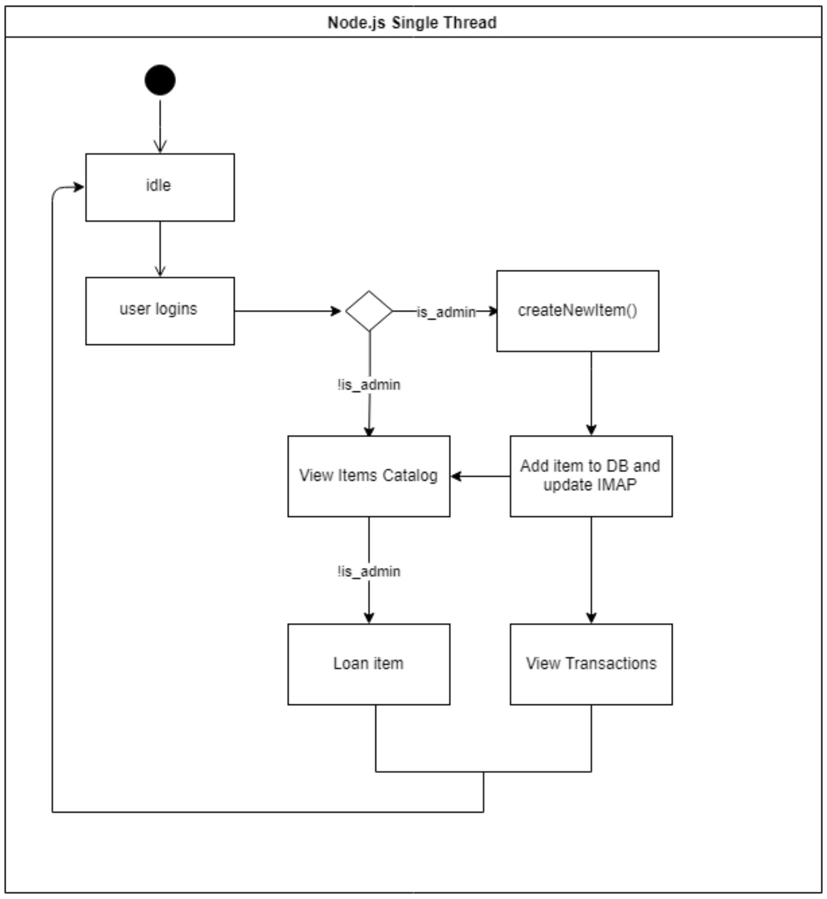


Figure 26: Process View: Activity Diagram

11. Deployment (Physical) View

The deployment view of Biblio, shown in figure 22, illustrates the physical components of the architecture that shapes the physical system. It is made up of three components: the client workstation, server and database. The client workstation uses a web browser to access the software system through an HTTP request. The Heroku Cloud Server service hosts the application container on a Hobby Dyno which contains all the application artifacts and logic. The application communicates with a similar Heroku Cloud Server hosting the systems PostgreSQL Database which accesses through SQL queries.

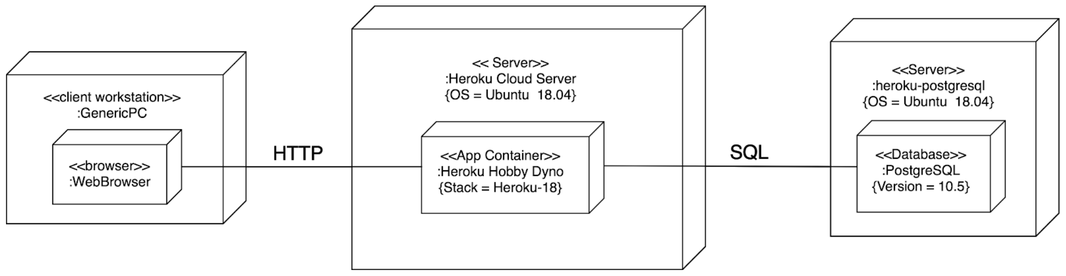


Figure 27: Deployment (Physical) View

12.Data View

The following section describes the data view which illustrates the data entities and their relationships.

12.1. Database Data Model

The data model diagram shown below in figure 23 defines how the logical structure of the database is modeled.

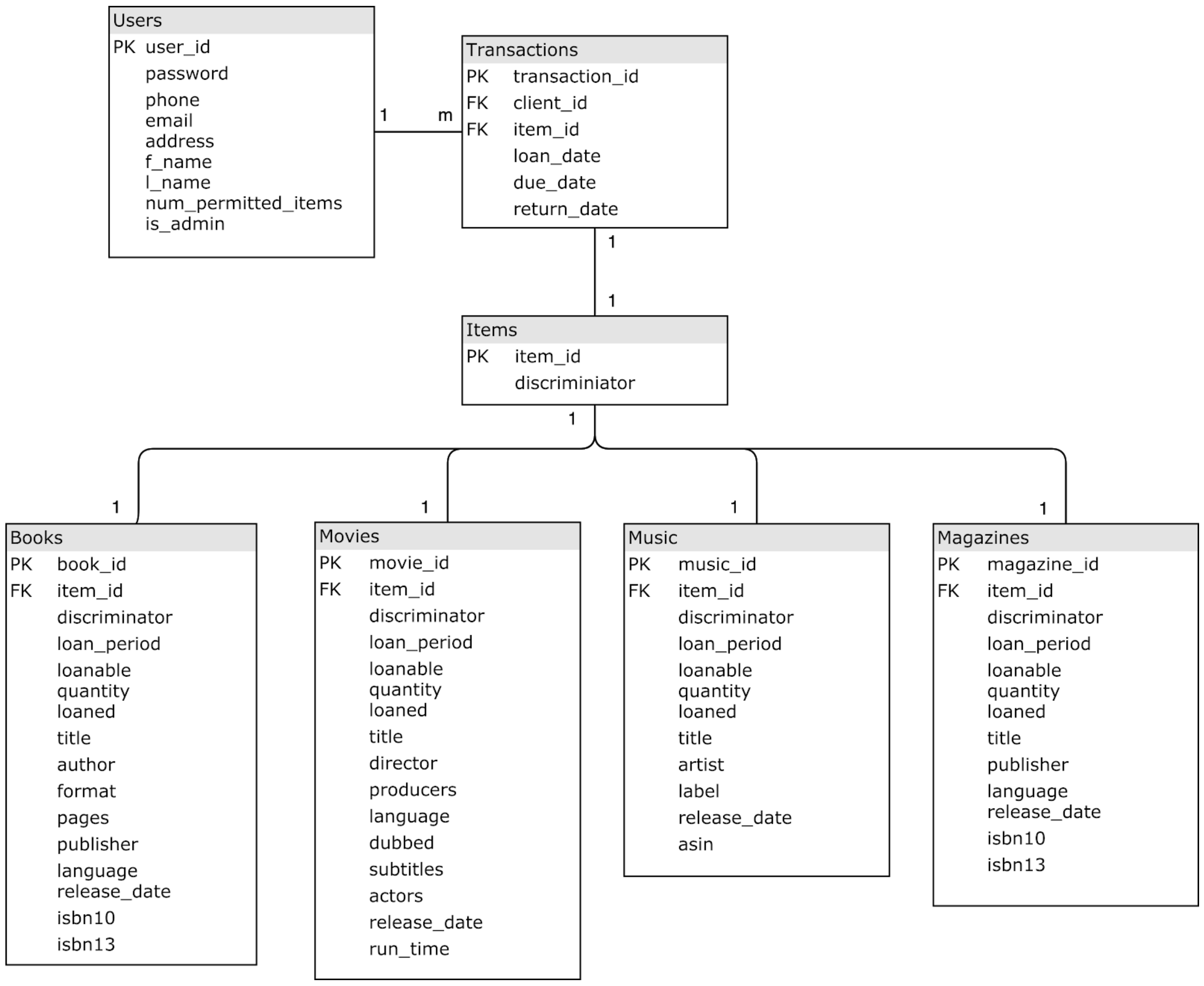


Figure 28: Database Data Model Diagram

13.Quality

13.1. Security

Description: The system shall block any unauthorized attempts at usage while still providing services to legitimate users.
Solution: By requiring a valid email and password to access some parts of the application.
Description: Administrators shall not have access to users' passwords.
Solution: By hashing passwords before storing them in the database.

13.2. Portability

Description: The system shall be compatible with all modern browsers such as: Google Chrome, Safari, Mozilla Firefox and Internet Explorer.
Solution: The system was created using Node.js, bootstrap CSS and HTML templates, which are all accessible on modern browsers.
Description: The system shall not require any hardware interfaces.
Solution: It is accessible through a URL.

13.3. Maintainability

Description : The system shall be able to maintain its functionality after its deployment.
Solution : Without the help of the developers, administrators shall be able to maintain all the core features of Biblio such as users and catalog items.

13.4. Performance efficiency

Description : The system shall utilize the least amount of memory and resource as possible during regular operation.
Solution : The system utilizes an Identity Mapper and a Unit of Work in the architecture design in order to limit the amount database queries and connections needed.