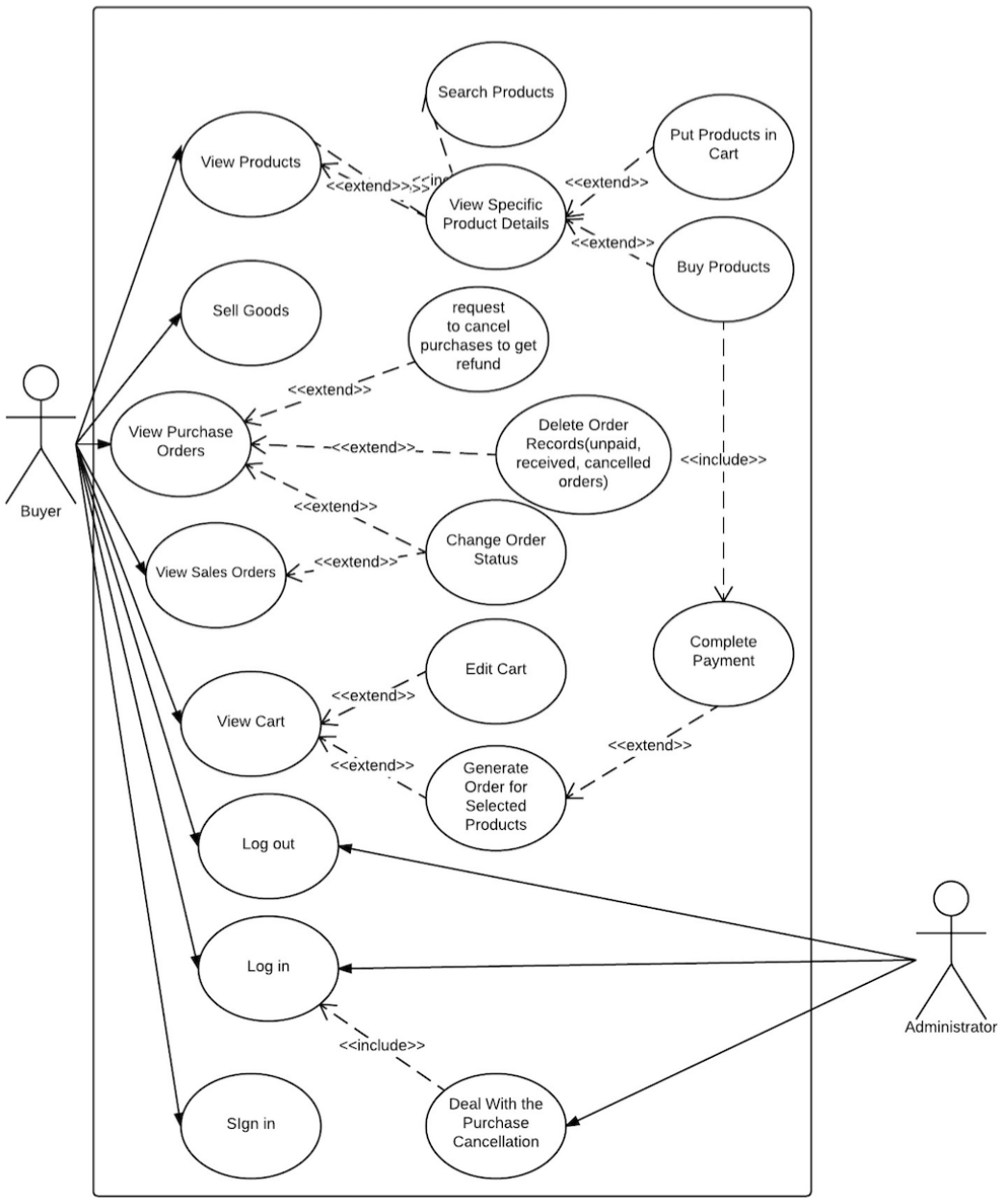
Ebuy — Feature B

Group 26
(Lindong Li, lindongl, 655251)
(Xingyu Ji, xingyuj, 622656)

Use Case


Search Products
View Products
Put Products in Cart
<<extend>>
View Specific Product Details
<<extend>>
Buy Products
Sell Goods
request to cancel purchases to get refund
<<extend>>
Buyer
View Purchase Orders
<<extend>>
Delete Order Records(unpaid, received, cancelled orders)
<<include>>
<<extend>>
Change Order Status
View Sales Orders
<<extend>>
Complete Payment
Edit Cart
<<extend>>
View Cart
<<extend>>
<<extend>>
Generate Order for Selected Products
Log out
Log in
Administrator
<<include>>
SIgn in
Deal With the Purchase Cancellation

View Products

Use Case Name	View Products
Primary Actor	Participant
Preditions	Participant is in the home page
Trigger	Participant wants to view products
Basic Path	1. Participant view products in the home page
Alternative Paths	none
Postconditions	The products page is displayed to the participant
Exception Paths	1. Error message page may show up as server is inaccessible

Search Products

Use Case Name	Search Products
Primary Actor	Participant
Preditions	Participant is in the home page
Trigger	Participant wants to search a particular type of product
Basic Path	1. Participant types key words of products in the search box and then triggers the search function 2. The search result is displayed to the participant 3. Participant triggers the sort function to sort search result according to the prices, distances and other attributes
Alternative Paths	1a. Participant chooses one category of products to search products, for example, clothes, electronic products etc.
Postconditions	The result adhere to search and sort criteria is displayed to participant
Exception Paths	1. Error message page may show up as server is inaccessible

View Specific Product Details

Use Case Name	View Specific Product Details
Primary Actor	Participant
Preditions	Participant is in the products list page
Trigger	Participant wants to get more information about one product
Basic Path	1. Participant triggers directing to the specific product details page function
Alternative Paths	none
Postconditions	New page about the product detailed information is displayed to the participant
Exception Paths	1. Error message page may show up as server is inaccessible

Put Products in Cart

Use Case Name	Put Products in Cart
Primary Actor	Participant
Preditions	Participant is in the specific product details page
Trigger	Participant manages to put products in cart
Basic Path	1. Participant specifies the quantity of products that he or she wants 2. Participant triggers putting products in cart function 3. If participant has logged in, the operation success message would be returned to participant
Alternative Paths	3a. If participant has not logged in, the login page would be displayed to participant 3a1. participant fills in the login information 3a2. if the login information is correct, the previous page would be returned. if the login information is incorrect, return to the step 3a1
Postconditions	Product has been added to participant's cart
Exception Paths	1. Error message page may show up as server is inaccessible 2. Error message page may show up when the product is no longer available

Buy Products

Use Case Name	Buy Products
Primary Actor	Participant
Preditions	Participant is in the specific product details page
Trigger	Participant manages to buy a type of product immediately
Basic Path	1. Participant specifies the quantity of products that he or she wants 2. Participant triggers buy immediately function 3. If participant has logged in, a page requesting receiver information would be displayed to the participant 4. Participant fills in the receiver information and select the link to pay 5. If the receiver information is proper, the new order would be generated and the pay page would be displayed to the participant 6. Participant fills in the payment information 7. If the payment is successful, the successful information would be returned to participant
Alternative Paths	3a. If participant has not logged in, the login page would be displayed to participant. 3a1. participant fills in the login information 3a2. if the login information is correct, the previous page would be returned. if the login information is incorrect, return to the step 3a1 5a. If the receiver information is improper, go back to step 4 7a. If the payment is failed, the failed information would be returned to participant
Postconditions	New order has been generated
Exception Paths	1. Error message page may show up as server is inaccessible 2. Error message page may show up when the product is no longer available

View Purchase History

Use Case Name	View Purchase Orders
Primary Actor	Participant
Preditions	Participant has accessed to the system
Trigger	Participant wants to view his/her purchase history
Basic Path	1. Participant access to “Purchase History” 2. If participant has logged in, the purchase history would be displayed to the participant
Alternative Paths	2a. If participant has not logged in, the login page would be displayed to participant 2a1. participant fills in the login information 2a2. if the login information is correct, the previous page would be returned. if the login information is incorrect, return to the step 2a1
Postconditions	Purchase History is displayed to the participant
Exception Paths	1. Error message page may show up as server is inaccessible

Delete Order Records

Use Case Name	Delete Order Records
Primary Actor	Participant
Preditions	Participant is in the purchase history page
Trigger	Participant manages to delete one or more order records
Basic Path	1. Participant selects one or more records 2. Participant triggers deleting orders function
Alternative Paths	none
Postconditions	The deleted records will not show up in purchase history any more
Exception Paths	1. Error message page may show up as server is inaccessible

Change Order Status

Use Case Name	Change Order Status
Primary Actor	Participant
Preditions	Participant is in the purchases orders or sales orders page
Trigger	The participant manages to change the order status
Basic Path	1. If the participant is buyer and if the payment has been completed, buyer could change the order status to received after receiving products 2. The correct status of the order would be displayed to the participant
Alternative Paths	1a. If the participant is buyer and if the payment has not been completed, the buyer could change the order status to payed 1a1.Participant triggers the completing payment function 1a2.Participant fills in the payment information 1a3.If the payment is successful, the successful information would be returned to participant If the payment is failed, the failed information would be returned to participant 1c. If the participant is seller and if the payment status is paid, the seller could change the status to shipped after shipping products
Postconditions	The order status has been changed correctly.
Exception Paths	1. Error message page may show up as server is inaccessible

View Cart

Use Case Name	View Cart
Primary Actor	Participant
Preditions	Participant has accessed to the system
Trigger	Participant wants to view the content of the cart
Basic Path	1. Participant select the link to view the cart 2. if the participant has logged in, the correct content of the cart would be displayed
Alternative Paths	2a. If participant has not logged in, the login page would be displayed to participant 2a1. participant fills in the login information 2a2. if the login information is correct, the previous page would be returned. if the login information is incorrect, return to the step 2a1
Postconditions	the correct content of the cart would be displayed to the participant
Exception Paths	1. Error message page may show up as server is inaccessible

Edit Cart

Use Case Name	Edit cart
Primary Actor	Participant
Preditions	Participant is in the cart page
Trigger	Participant wants to change the items of the cart
Basic Path	1. Participant change the quantity of the product 2. The correct content of the cart would be displayed to the participant
Alternative Paths	1a. Participant trigger the function to delete one or more products from the cart completely 1a1. Participant selects one or more products that he or she wants to delete 1a2. Participant triggers the delete function 1a3. A dialogue would be displayed to the participant to ask if the participant would confirm this operation 1a4. If participant select Yes, the product would be deleted from the cart completely. If participant select No, the operation would not be operated
Postconditions	The correct content of the cart would be displayed to the participant
Exception Paths	1. Error message page may show up as server is inaccessible

Generate Order For Selected Products

Use Case Name	Generate Order for Selected Products
Primary Actor	Participant
Preditions	Participant is in the cart page
Trigger	Participant wants to select and buy some products in the cart
Basic Path	1. Participant selects what they want to buy in the cart 2. Participant triggers the checkout function 3. Participant fills in the receiver information and select the link to pay 4. If the receiver information is proper, the new order would be generated and the pay page would be displayed to the participant
Alternative Paths	4a. If the receiver information is improper, go back to step 3
Postconditions	the new order would be generated
Exception Paths	1. Error message page may show up as server is inaccessible 2. Error message page may show up when the product is no longer available

Complete Payment

Use Case Name	Complete Payment
Primary Actor	Participant
Preditions	The order has been generated
Trigger	The participant manages to complete the payment
Basic Path	1. Participant triggers the completing payment function 2. Participant fills in the payment information 3. If the payment is successful, the successful information would be returned to participant
Alternative Paths	3a. If the payment is failed, the failed information would be returned to participant
Postconditions	The payment has been completed
Exception Paths	1. Error message page may show up as server is inaccessible

Log in

Use Case Name	Log in
Primary Actor	User
Preditions	User has signed up
Trigger	User wants to log in the system
Basic Path	1. User selects a link to log in 2. User selects the user type 3. User supplies identification details 4. If the user has validated identification details, then website is redirected to the personal center
Alternate Paths	4a. If the identification details are not validated then go back to step 2
Postconditions	The user has logged in
Exception Paths	1. The Website may be unable to connect to the server at any time

Log out

Use Case Name	Log out
Primary Actor	User
Preditions	User has logged out
Trigger	User wants to log out
Basic Path	1. User selects a link to log out 2. User will be redirected to the home page
Alternate Paths	none
Postconditions	The user has logged out
Exception Paths	1. The Website may be unable to connect to the server at any time

Sign up

Use Case Name	Sign up
Primary Actor	Participant
Preditions	none
Trigger	Participant wants to sign in
Basic Path	1. Participant selects a link to sign in 2. Participant fills the form 3. Participant submit the form 4. If the details are proper like the form of the email address is correct etc, the successful information would be returned and the participant would be redirected to the home page
Alternate Paths	4a. If the details are improper, then return to step 2
Postconditions	The user has signed up
Exception Paths	1. The Website may be unable to connect to the server at any time

Request to Cancel Purchases to Get Refund

Use Case Name	Request to Cancel Purchases to Get Refund
Primary Actor	Participant
Preditions	the status of purchases order is paid or received
Trigger	Participant wants to cancel purchases to get refund
Basic Path	1. User selects one order(the status of the order must be paid or received) to request to cancel purchases 2. The status of the order turns into pending
Alternate Paths	none
Postconditions	The request to cancel requests has been sent to participants
Exception Paths	1. The Website may be unable to connect to the server at any time

Deal With the Purchase Cancellation

Use Case Name	Deal With the Purchase Cancellation
Primary Actor	Administrator
Preditions	The administrators has logged in
Trigger	Administrator needs to deal with the requests of canceling purchases
Basic Path	1. Administrator selects a link to view one request details 2. Administrator contacts the seller and participant to deal with this request 3. According to the results of the negotiation, if the request could be approved, Administrator approves this request
Alternate Paths	3a. According to the results of the negotiation, if the request could not be approved, Administrator rejects this request
Postconditions	The request would be approved or rejected and the status of the order would turns in canceled or the original status(paid or received) accordingly
Exception Paths	1. The Website may be unable to connect to the server at any time

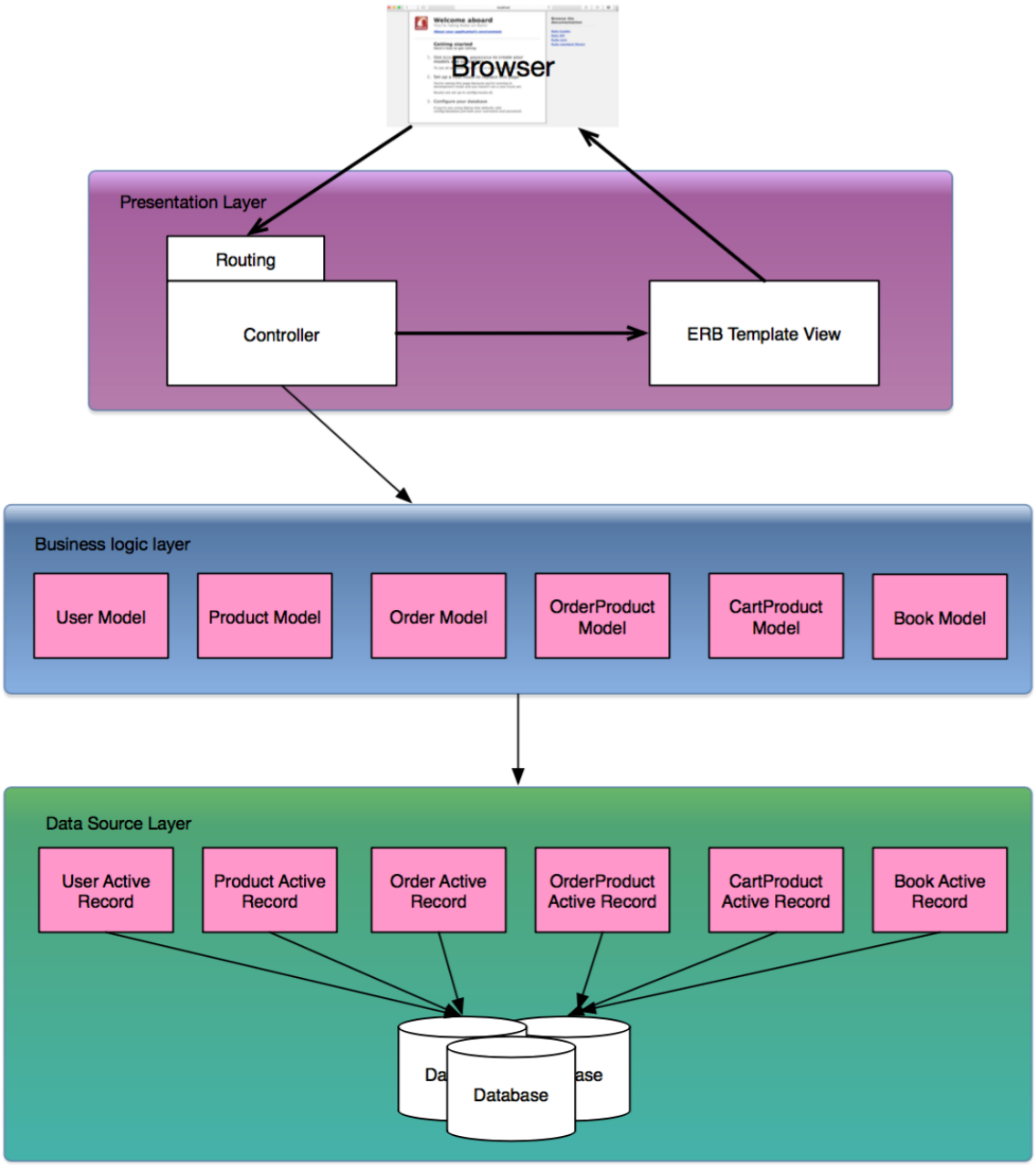
Sell Goods

Use Case Name	Sell Goods
Primary Actor	Participant
Preditions	The Participant has logged in
Trigger	Participant wants to sell goods
Basic Path	1. Participant selects link to sell goods 2. Participant selects what kind of products they want to sell 3. Participant fills the form of details of selling product 4. Participant submit the form 5. If the details are proper like quantity is positive etc, the successful information would be returned to the participant
Alternate Paths	5a. If the details are improper then go back to step 3
Postconditions	The new selling products would be added
Exception Paths	1. The Website may be unable to connect to the server at any time

View Sales Orders

Use Case Name	View Sales Orders
Primary Actor	Participant
Preditions	Participant has accessed to the system
Trigger	Participant wants to view his/her sales orders
Basic Path	1. Participant access to “check sales” 2. If participant has logged in, the sales orders would be displayed to the participant
Alternative Paths	2a. If participant has not logged in, the login page would be displayed to participant 2a1. participant fills in the login information 2a2. if the login information is correct, the previous page would be returned. if the login information is incorrect, return to the step 2a1
Postconditions	Sales orders are displayed to the participant
Exception Paths	1. Error message page may show up as server is inaccessible

High level architecture



Note：Some models and active records that are different types of product are not listed in this diagram. Book model and Book active record is an example of them.

Presentation layer

Routing

The Routing is responsible for working out where in the application the request should be sent and how the request should be passed. Routing is a part of the front controller.

Controller

A Controller is responsible for handling the requests from the browser, interrogating the models and passing data back to the views.

ERB Template View

A View represents the user interface of the application. They are usually HTML files with embedded ruby code.

Business Logic Layer

Domain model

Domain model is an oriented-object model and couples the behavior to the objects in the domain using methods.

Data Source Layer

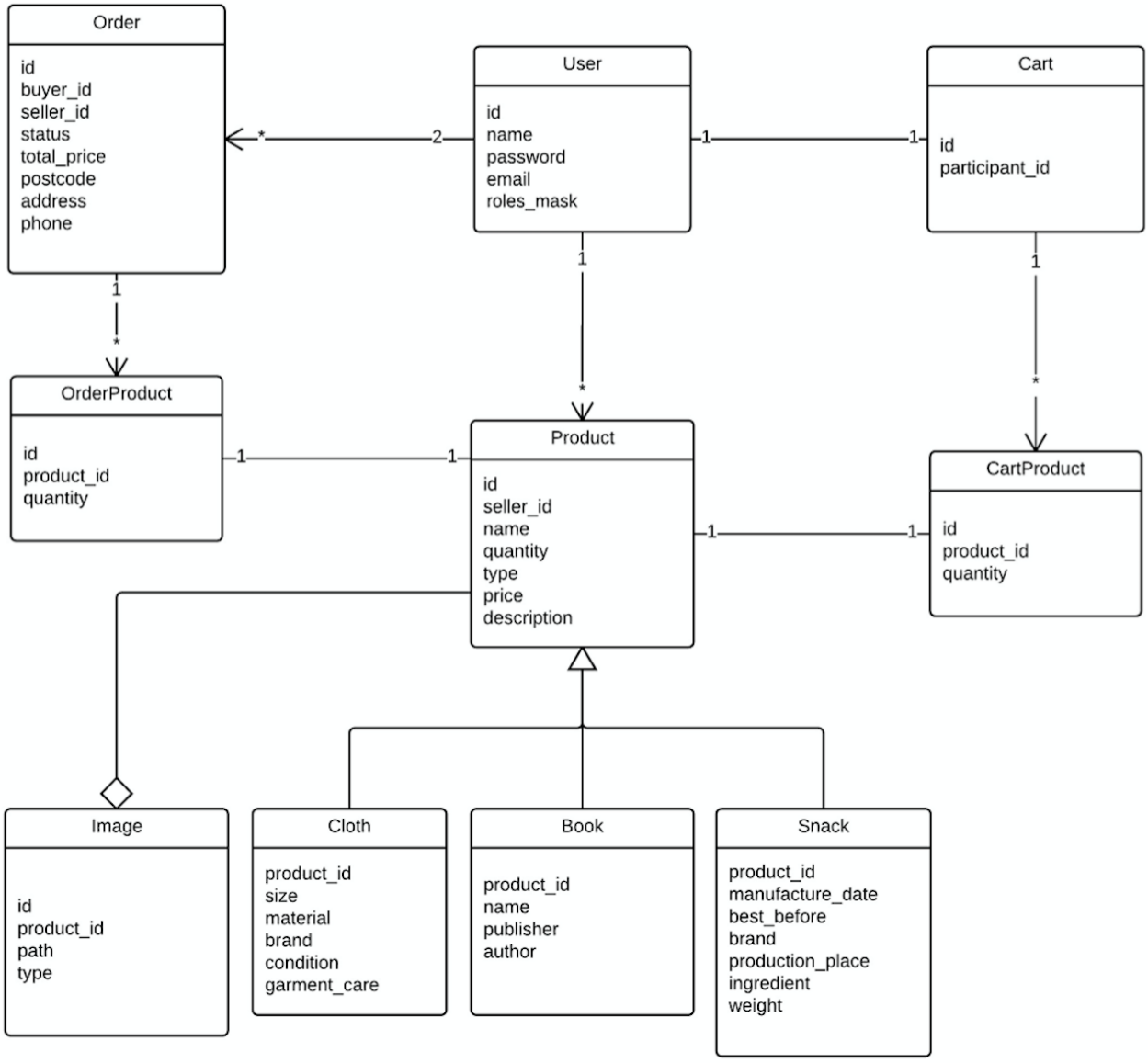
Active Record

Active record is an object that wraps a row in a database table and encapsulate the database access.

Database

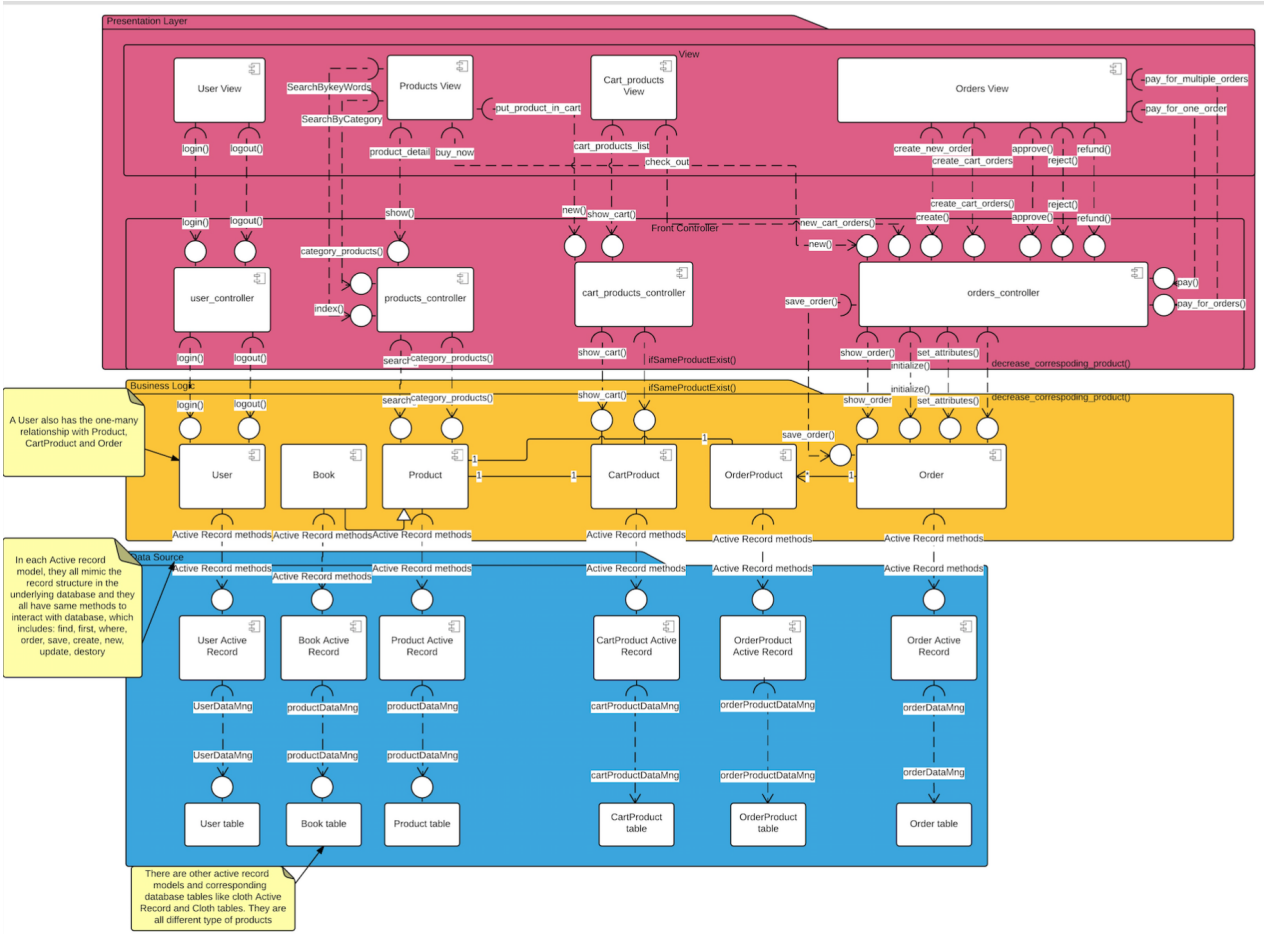
Database is used to store the data of the application. In our project, we use the SQLite for development and the PostgreSQL for production.

Domain Diagram



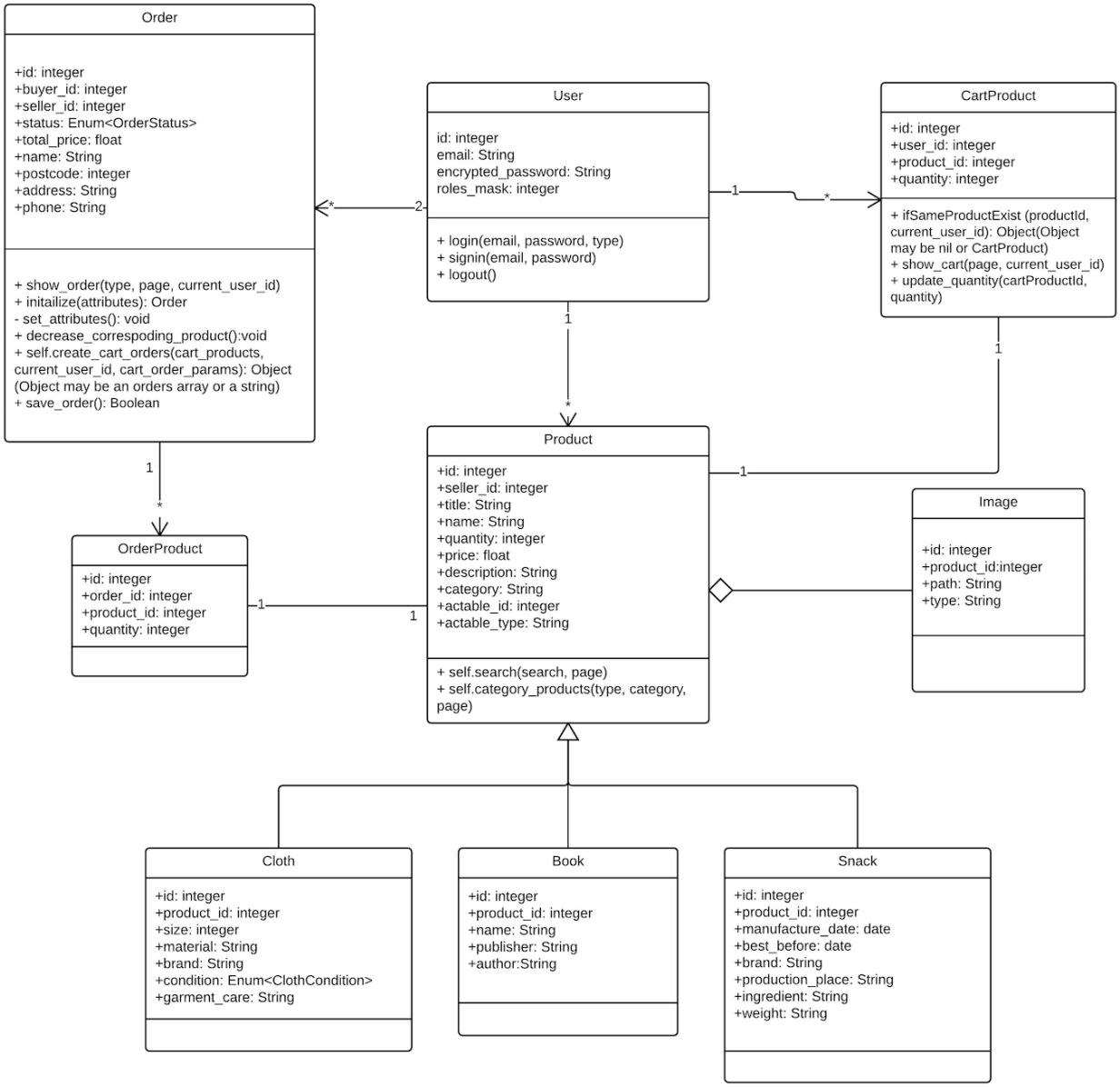
Structure, interaction, and behaviour

Component diagram



Note: The details of the domain model are in the class digram. The details of the active record methods are in the interface of the Data source layer.

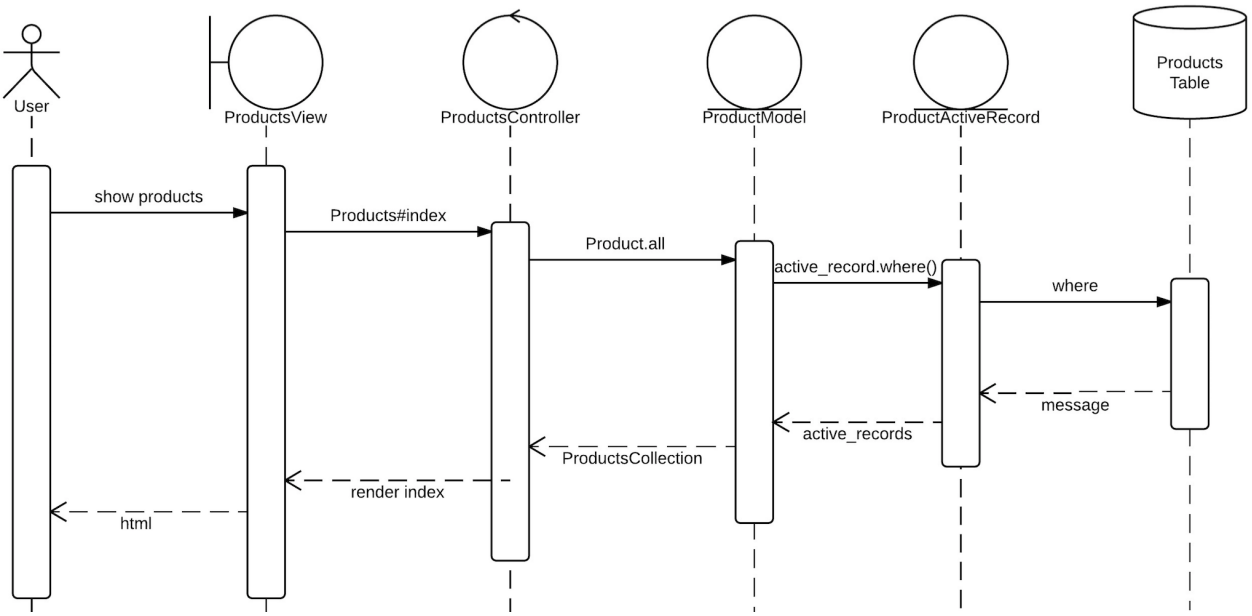
Class Diagram



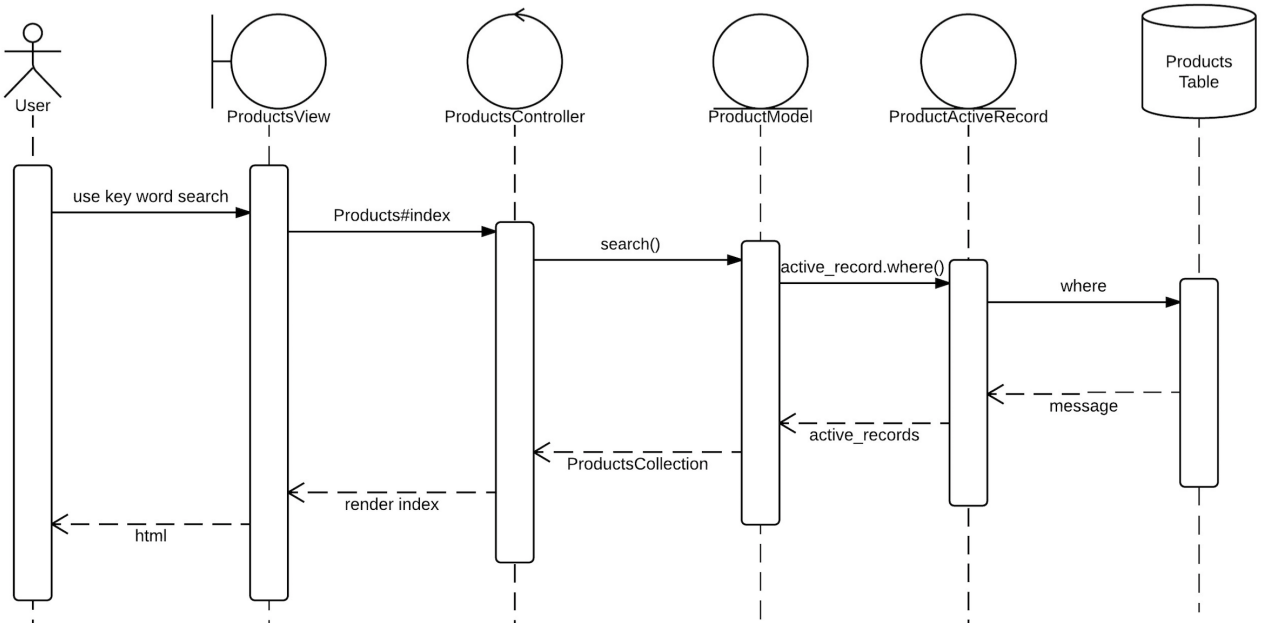
Note: this class digram is for the domain model in the business logic

Sequence Diagram

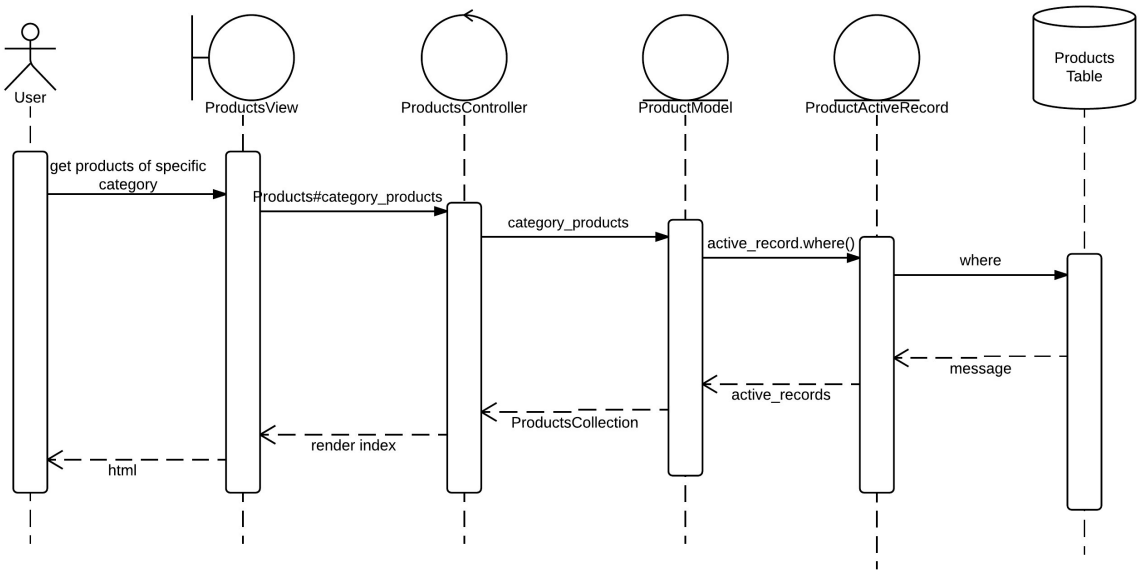
View Products



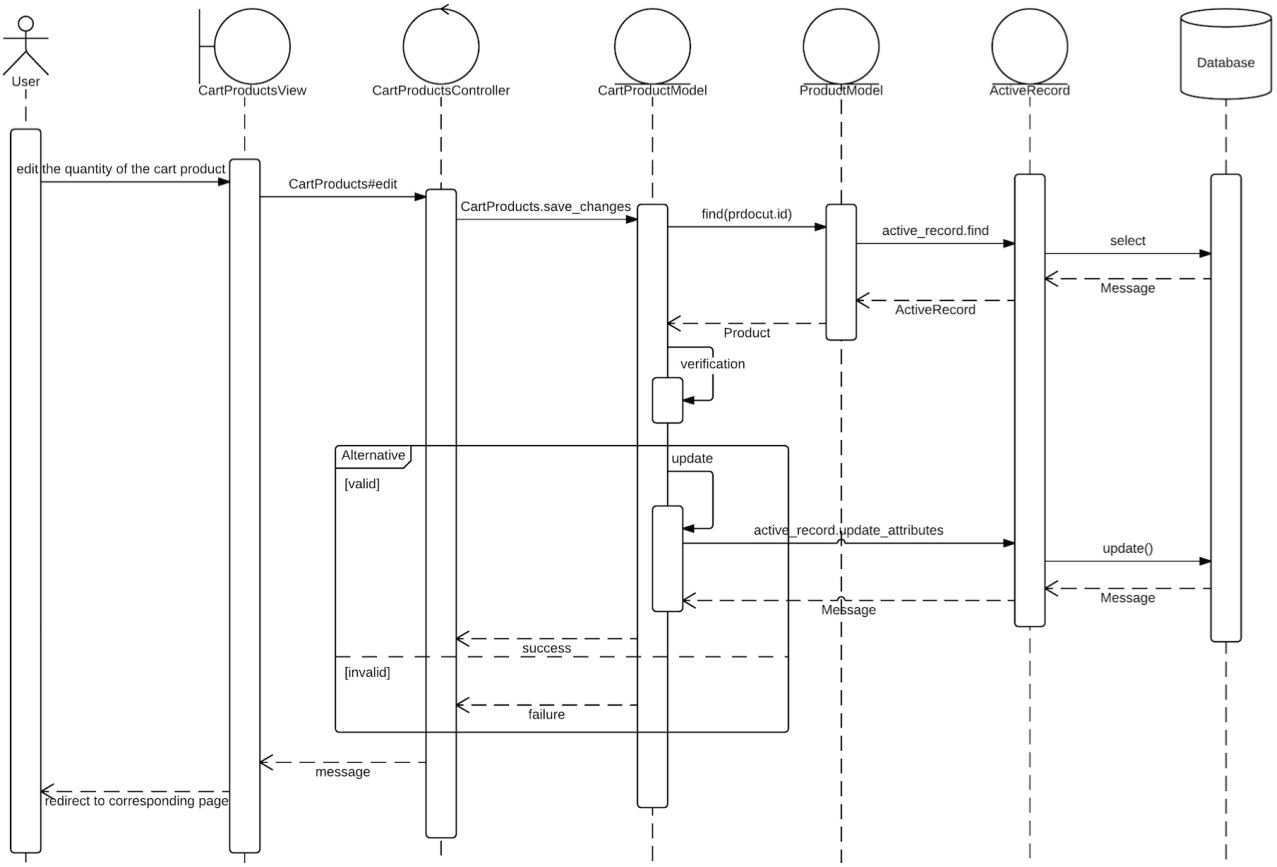
Search Products by Keyword



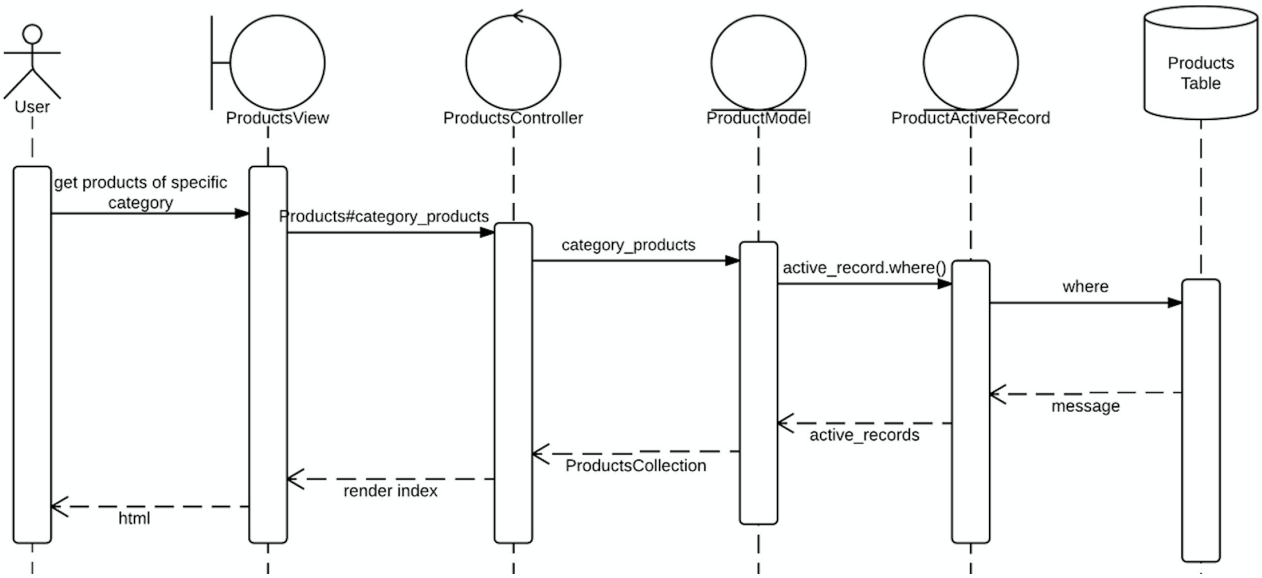
Search Products by Category



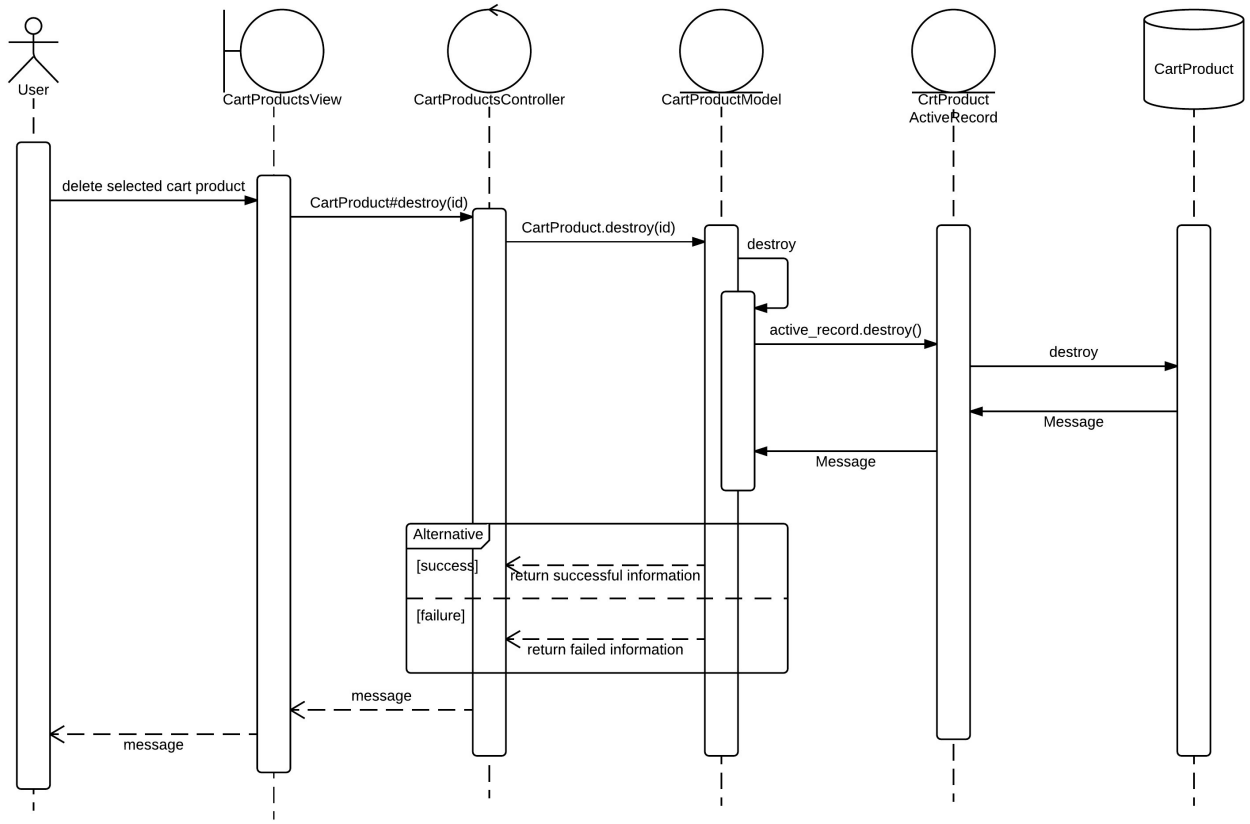
Edit Cart



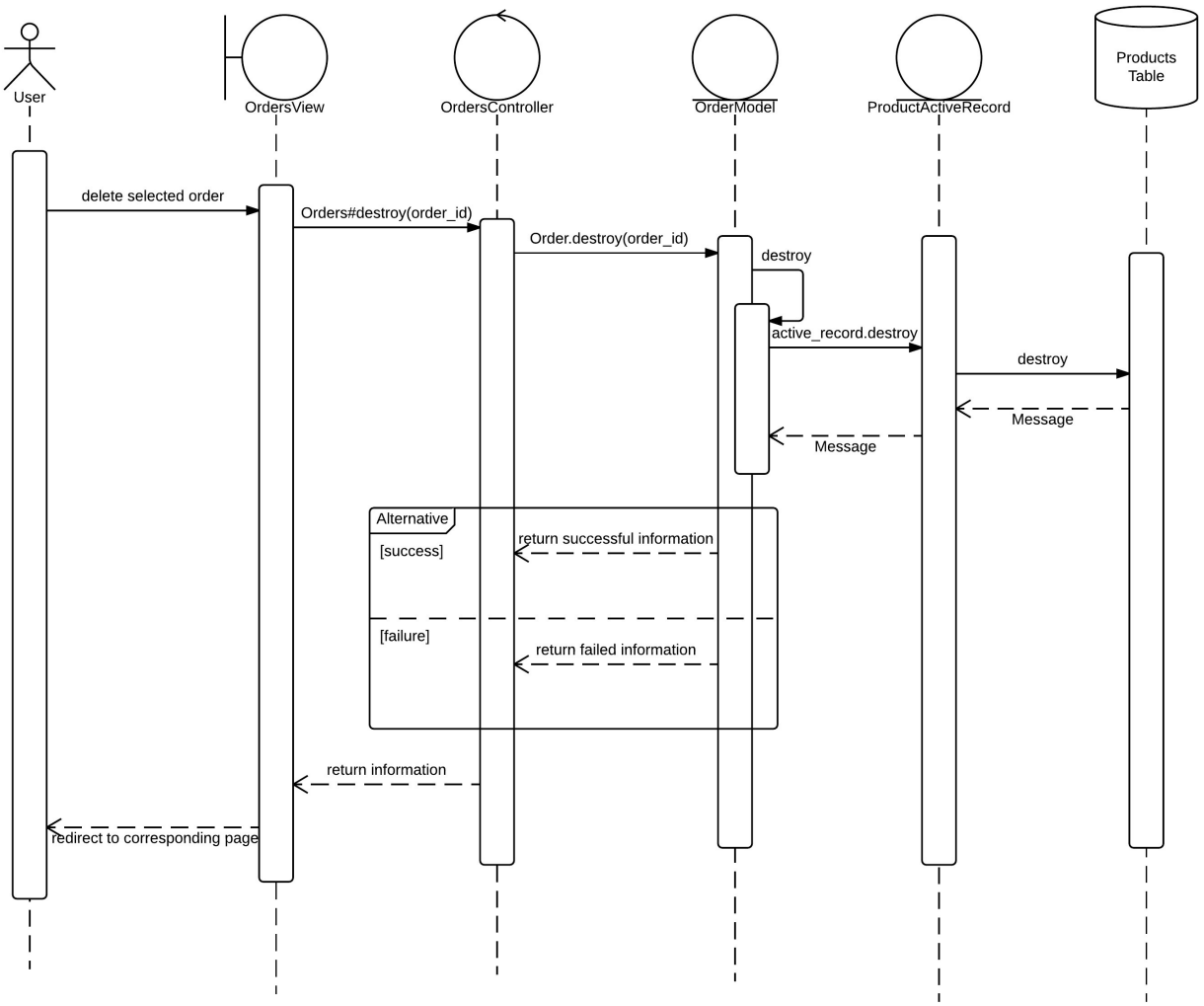
Show Cart



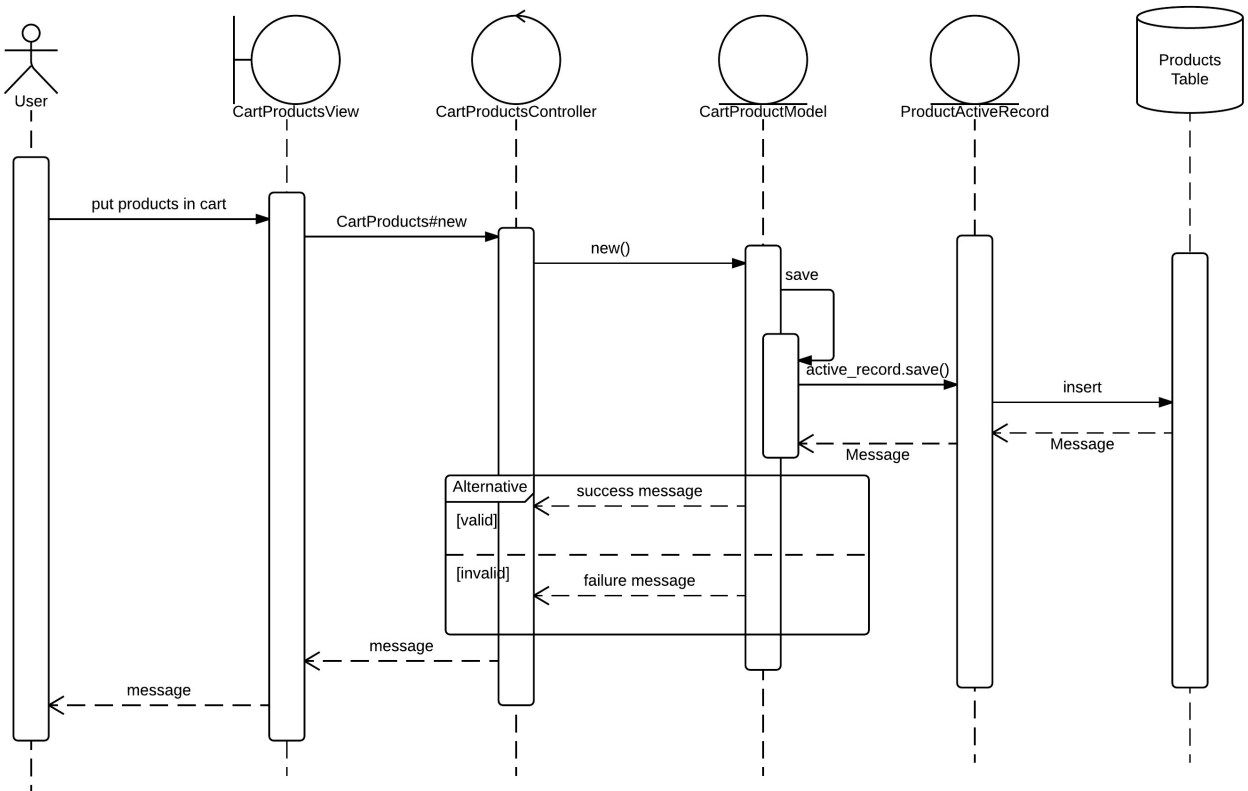
Delete Cart Product



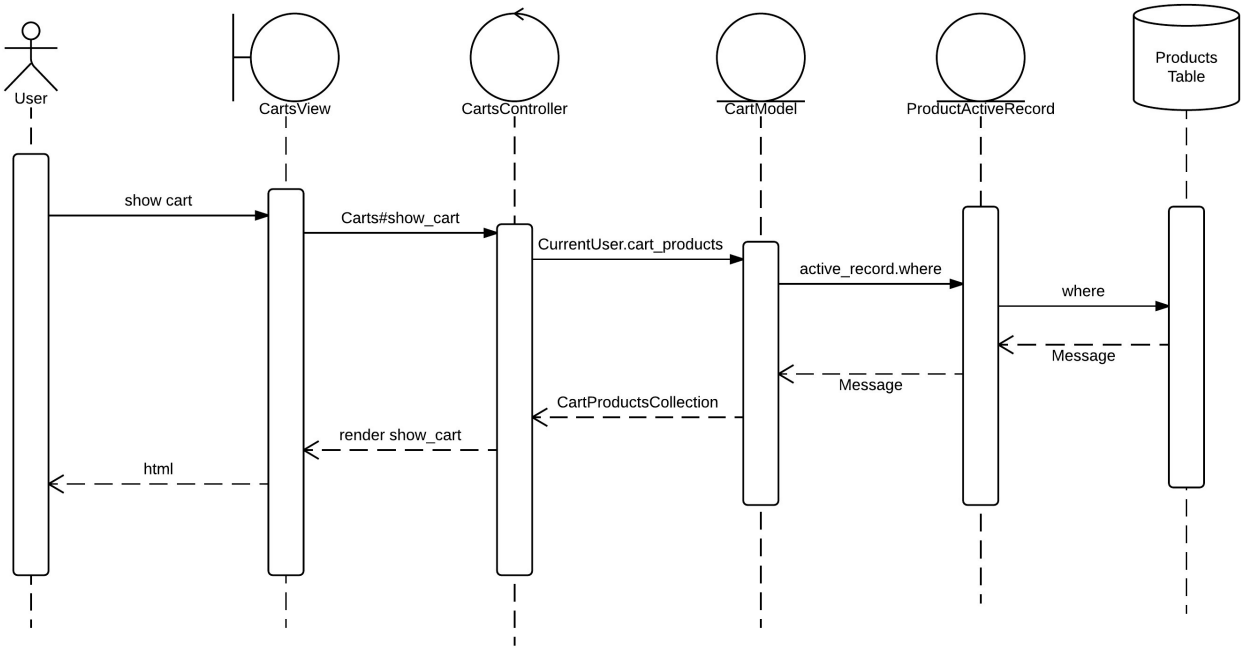
Delete Order Records



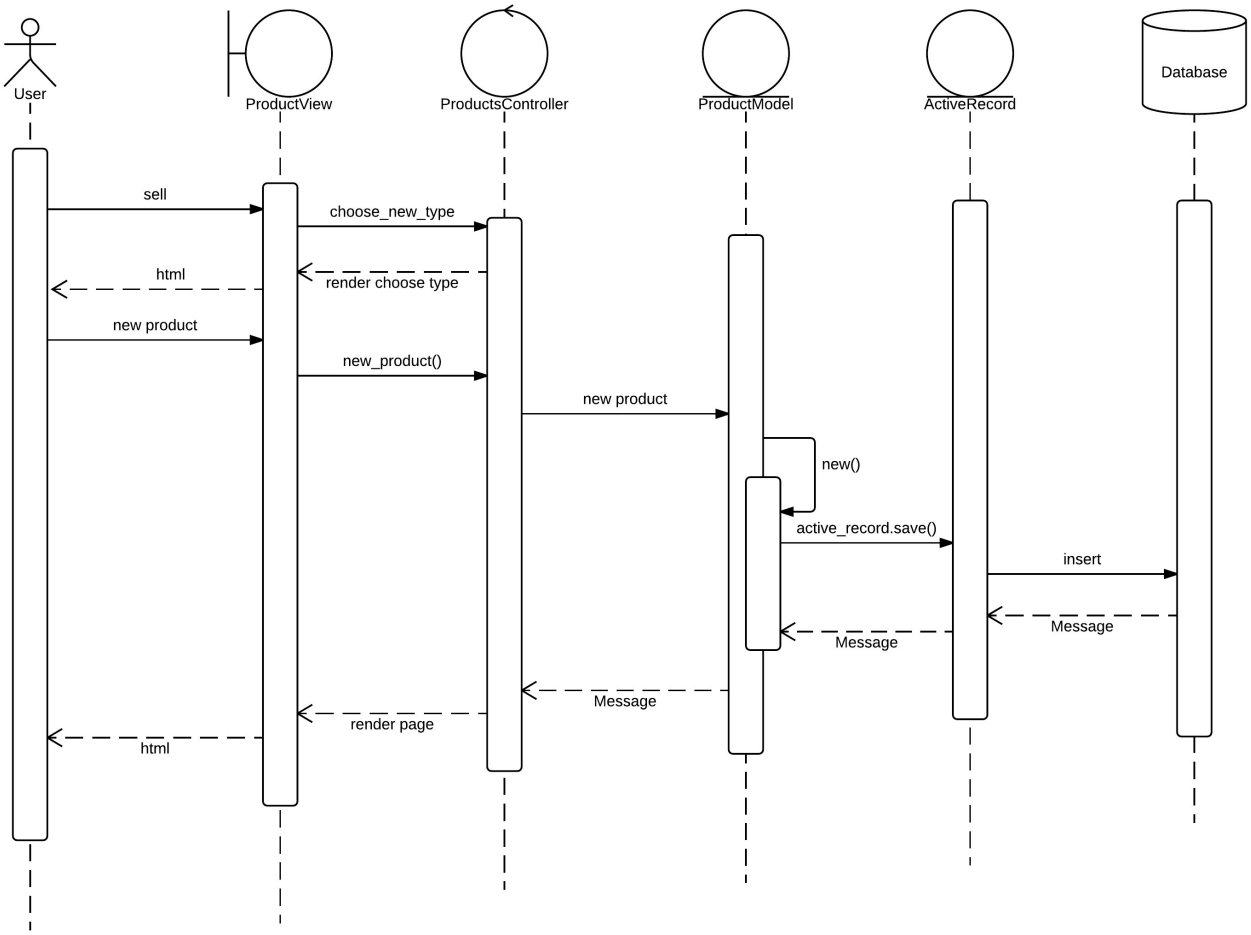
Put products in cart



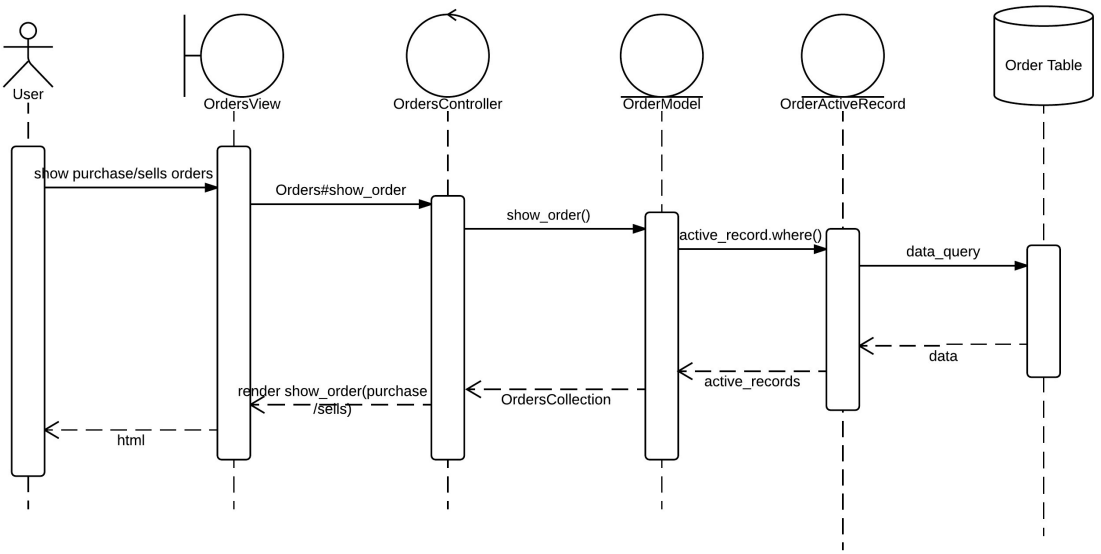
View Cart



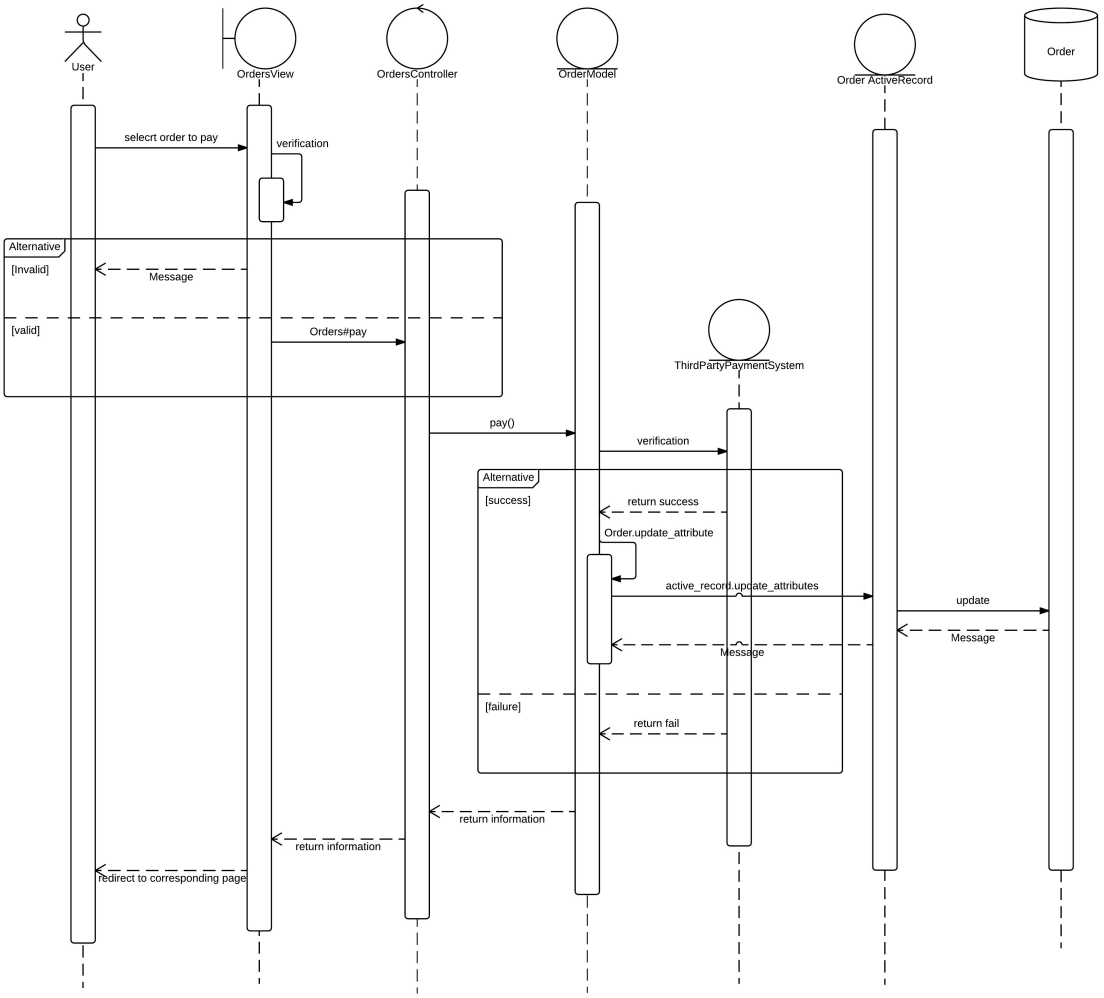
Sell Goods



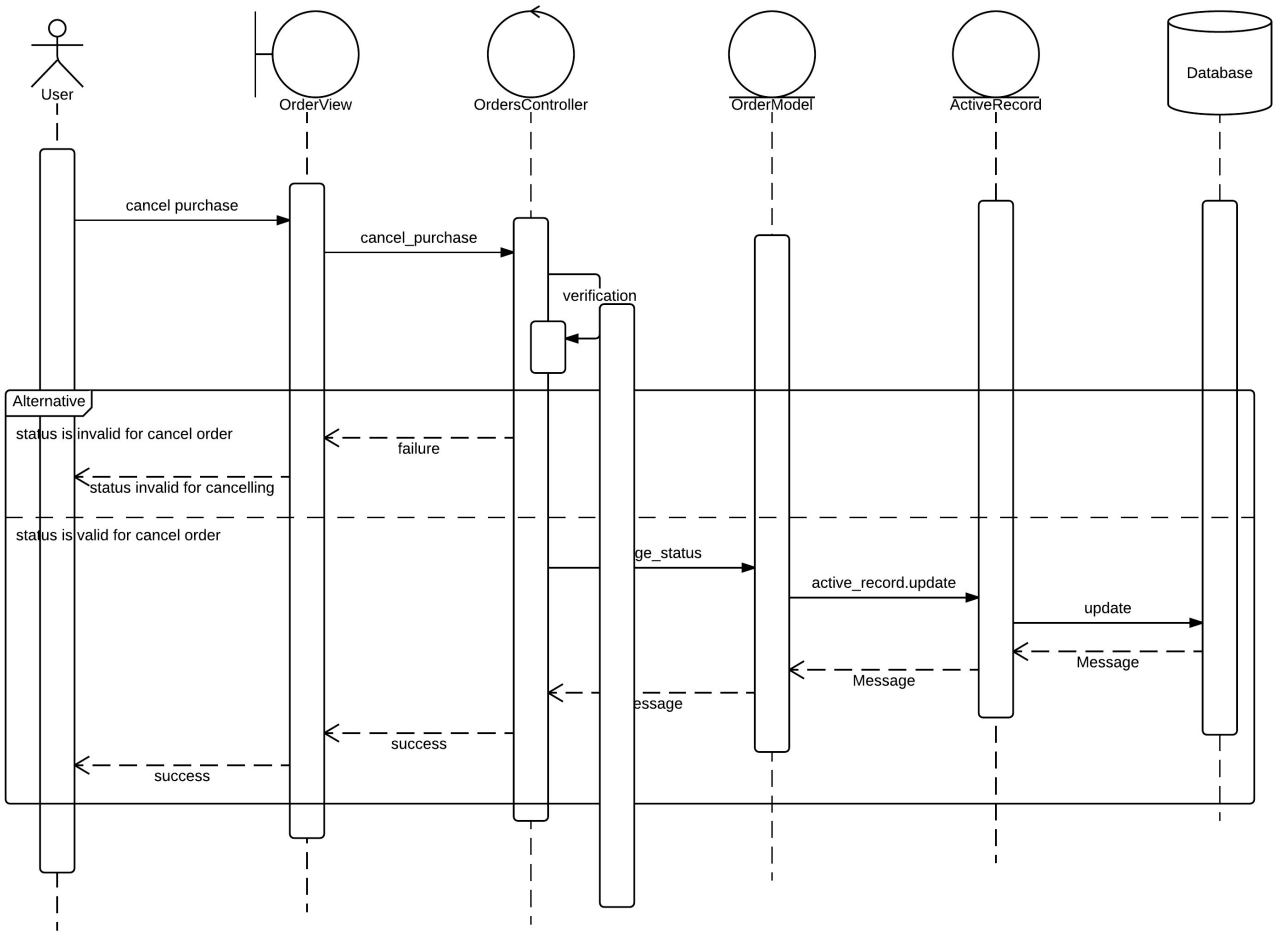
View purchases/sells orders



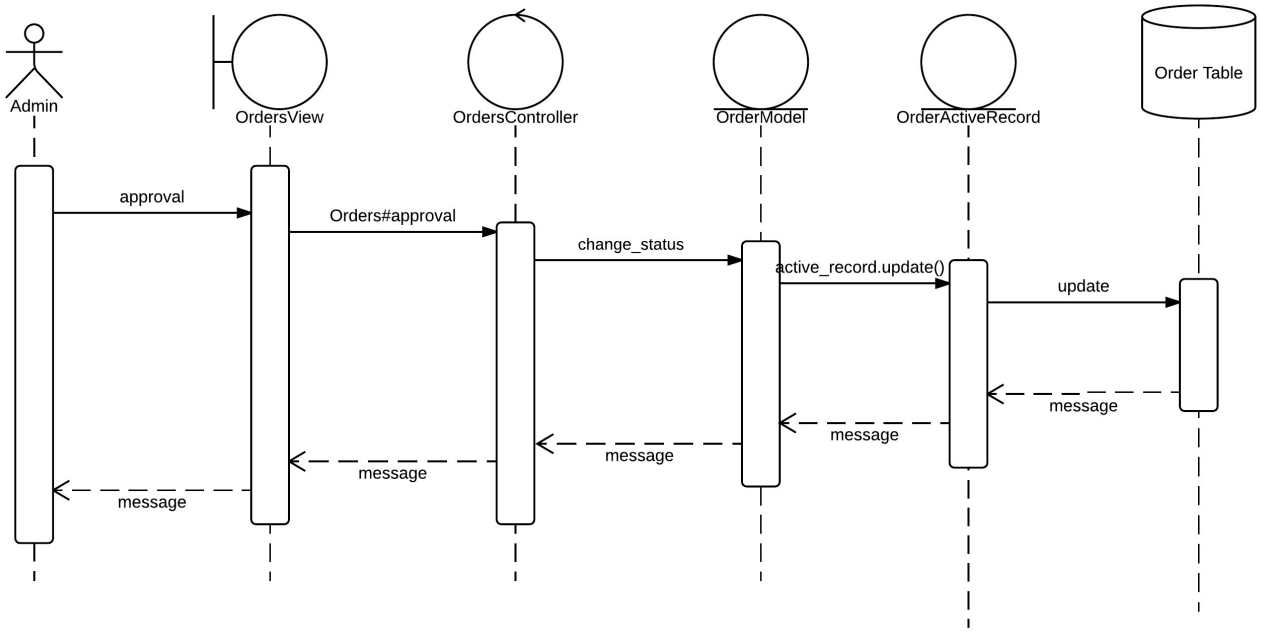
Complete payment



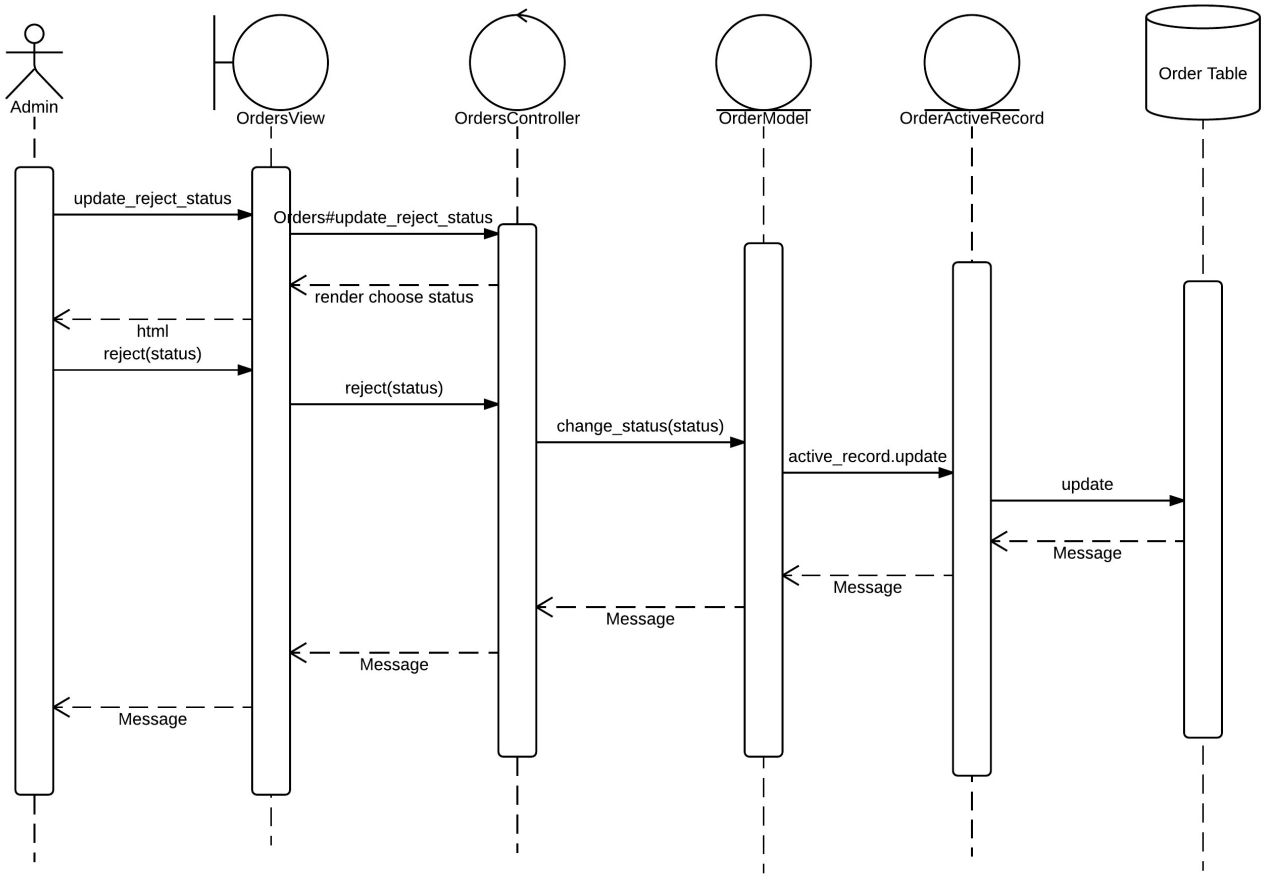
Request to cancel purchases



Approve purchases cancellation



Reject purchases cancellation



Interface

Presentation Layer

View module

The detail of the interfaces in the view module has been listed in the Component diagram, all the interfaces in the view module could be classified into three types: post, put, delete.

Interface	Specification
get(requests[])	Renders the servlet to handle a GET request. This is mainly used to retrieve the data like, getting products list, getting product detail etc.
post(requests[])	Renders the servlet to handle a POST request. This is mainly used to create or modify the data like, creating a new order, putting purchased products in the cart etc.
delete(requests[])	Renders the servlet to handle a DELETE request. This is mainly used to delete data like, deleting one order, deleting one order etc.

Controller module

Interface	Specification
sign_up(String email, String password)	calls the methods in the user model to create a new user. If creation is successful, return to the previous page, if not, return the message that the registration fails.
login(String email, String password)	calls the methods in the user model to verify if the given email matches to the password. If authentication is successful, return to the previous page. If not, return the message that the authentication fails.
logout()	Terminates the session and redirect the user to home page.
products_controller: index(String search, integer page)	calls the methods in the Product model to retrieve the products list of the specific page from all products when the search is null. If the search is not null, retrieve the products list from the relevant products of the specific page. Then return the products_index page with the correct information.
products_controller: show(integer productId)	calls the methods in the Product model to retrieve the product information with the specific productId and then return the products_show page with the correct information.
cart_products_controller: show_cart()	calls the methods in the domain layer to retrieve the data of the current user's cart and return the correct view with the retrieved data.

Interface	Specification
cart_products_controller: new(integer amount,integer productId)	calls the methods in the domain layer to retrieve the current quantity of the product with the productId, If the amount is not more than the current quantity, calls the methods in the domain layer to create a new cart product record and return the cart product detail view. If the amount is more than the current quantity, returns the message that the stock is not enough.
cart_products_controller: update(integer amount,integer productId)	calls the methods in the Product model to retrieve the current quantity of the product with the productId, If the amount is not more than the current quantity, calls the methods in the domain layer to update the cart product record and change the view correctly. If the amount is more than the current quantity, returns the message that the stock is not enough and set the amount of the cart product as the quantity of the stock.
cart_products_controller: destroy(integer cartProductId)	calls the methods in the CartProduct model to delete the selected cart products. After deletion, change the view correctly.
orders_controller: show_order(String type)	calls the methods in the Order model to retrieve the all orders of the specific type of the current user, and then returns the orders_index page with correct inforamtion.
orders_controller: new(integer productId, integer amount)	calls the methods in the Product model to retrieve the current quantity of the product with the productId, If the amount is not more than the current quantity, return to the orders_new page. If the amount is more than the current quantity, return the message that the stock is not enough.
orders_controller: newCartOrders(integer[] cartProductId)	for each cartProductId, retrieve the amount of the cart product by calling the methods in the CartProduct model and retrieve the stock of the same product. If the amount of the each cart product is not more than the stock of the corresponding product, return to the orders_new page. If not, return the message that which product that does not have the enough stock.
orders_controller: create(String name, integer postcode, String address, String phone, integer productId, integer amount)	calls the methods in the Product model to retrieve the current quantity of the product with the productId, If the amount is not more than the current quantity, call methods in the Order model to create the new order and then return the orders_show page. If the amount is more than the current quantity, return the message that the stock is not enough.

Interface	Specification
orders_controller: createCartOrders(String name, integer postcode, String address, String phone, integer[] cartProductId)	for each cartProductId, retrieve the amount of the cart product by calling the methods in the CartProduct model and retrieve the stock of the same product. If the amount of the each cart product is not more than the stock of the corresponding product, call the methods in the Order model and for different seller, create different orders and then returns the orders_index page. If not, return the message that which product that does not have the enough stock.
orders_controller: pay()	If the payment is successful, retrieve the orderIds in the session and then call the methods in the Order model to change the status of the order and finally, return the message that the payment is successful. If not, return the message that the payment fails.
orders_controller: approval()	When a administrator user approval a order to be cancelled, this action would be called. This action will change the status of corresponding order to “cancelled”
orders_controller:update_reject_status()	This action is a “get” action, it will be called when the administrator reject the order requested for cancellation. This action will render a page, allowing administrator to choose a new status for the requested order.
orders_controller:reject()	After choosing a new status of a requested order, the reject action will be called to change the corresponding order to the selected status
orders_controller:refund()	Buys are able to cancel a paid order through action refund. This action will change the status of the refund order to “Pending”, thus, administrator could be able to check this order.

Business Logic Layer

Interface	Specification
sign_up(String email, String password)	Create a new User record in the database. This function is implemented by devise
login(String email, String password)	Retrieve the email and password to authenticate. This function is implemented by devise.
logout()	Terminates the session. This function is implemented by devise.
product: search(String search, integer page)	if the search is not null, retrieve the data relevant to the search content from the database and return them to the controller. If the search is null, retrieve all products in the database and return them to the database.
product: category_products(String type, String category, integer page)	retrieve all products of the category of the type and return them to the controller. (Type is used to describe a kind of product like, Cloth. Category is used to classify products in one type like, Top)
cart_product: show_cart(integer page, integer current_user_id)	retrieve all cart products of the current user from the database and return them to the controller.
cart_product:ifSameProductExist(*productId*, *current_user_id*)	If there is same product exists in the cart, return this cart_product, else return nil.
order: show_order(String type, integer page, integer current_user_id)	retrieve all orders of the specific type(purchase, selling) of the current user from the database and return them to the controller.
order: initialize()	Overwrite constructor of order calling private function set_attributes, enable generating OrderProduct for order.
order: set_attributes()	Private function set_attributes, called by initialise. create OrderProduct for generated order and set other attributes
order: decrease_correspoding_product	Used by “buy it now” feature, to decrease quantity of product after corresponding order is created.
order: self.create_cart_orders	Static method create_cart_orders, used for checkout from cart. This method using transaction to ensure orders of every selected product successfully created. Otherwise, rollback all operations and return error information.

Data Source Layer

Interface	Specification
find(*args)	Find by id - This can either be a specific id (1), a list of ids (1, 5, 6), or an array of ids ([5, 6, 10]). If no record can be found for all of the listed ids, then RecordNotFound will be raised. If the primary key is an integer, find by id coerces its arguments using to_i.
first(limit = nil)	Find the first record (or first N records if a parameter is supplied). If no order is defined it will order by primary key.
where()	The where method allows you to specify conditions to limit the records returned, representing the WHERE-part of the SQL statement. Conditions can either be specified as a string, array, or hash.
order()	To retrieve records from the database in a specific order, you can use the order method.
save()	This method is used to save the model. If the model is new, a record gets created in the database, otherwise the existing record gets updated.
create(attributes = nil, &block)	Creates an object (or multiple objects) and saves it to the database, if validations pass. The resulting object is returned whether the object was saved successfully to the database or not.
new	Using the new method, an object can be instantiated without being saved
update()	Once an Active Record object has been retrieved, its attributes can be modified and it can be saved to the database by using this method.
destroy	An Active Record object can be destroyed which removes it from the database.

Note: these methods exist in every active record module.

Patterns and Patterns Rationale

Patterns of Architecture

MVC Pattern	
Why we choose this pattern	• MVC separates the presentation from the model. There are several benefits of this. Firstly, when we develop a view, we just need to think about the UI and how to make a good user interface and when we develop the model, we just need to think about the business policies, perhaps database interactions. Secondly, different users could see the same basic model information in different ways. In our project, sellers and buyers all could view the order information, but in a different ways. Finally, allowing us to test domain logic easily. • MVC separates controller from the view. Controller would be responsible for deciding how to deal with request and which view would be returned. The view would be responsible for displaying.
How to embody in our code	• We have models, controllers and views in our code and they all conform to the principle of this pattern.

Patterns of Presentation Layer

Template View	
Why we choose this pattern	• The domain of our project is small and most pages have the same layout. It is a natural way to embed the code in the template view to generate web pages containing dynamic information. • Template view is easy to implement.
Why we do not choose other patterns	• The main reason that we do not choose transform view and the two step view is because they are all involved with design complexity and implementation complexity.
How to embody in our code	• We use the erb files which are files embed the ruby code into HTML.

Front Controller	
Why we choose this pattern	• By using front controller, we just have one single entry point(Routing in our project) for all requests, which makes the configuration much easier. • Even though mostly there is a constraint of using the front controller that is design complexity, since we use the ruby on rails in our project, we do not need to worry about too much about this problem. Because it is quiet simple to implement this pattern in ruby on rails and we just need to set the right configuration in the routes.rb, then it could work very well.
Why we do not choose other patterns	• Although the page controller is easy to understand and quiet simple to implement, the main reason that we do not choose the pattern is that there would be some duplication code for generic actions and it is inevitable to put complex logic in single controller since one controller would handle all logic related to one page.
How to embody in our code	• In our code, there is a file called routes.rb which will be used by routing to dispatch the request to the correct action of the correct controller. And there are some controllers which contain multiple actions. Each action would deal with the request, call methods in the model and return the right view. • There is also a application_controller. Mostly all action controllers would inherit from this controller. The method in this controller could be accessed by all action controllers and we could put some method which will be executed before all other actions here, like authentication.

Patterns of Business Logic Layer

Domain Model	
Why we choose this pattern	• Domain model provides high extensibility for the system, since if we want to add more behavior in the future, we only need to add classes/ objects to the domain model. • Domain model aids in managing the complexity of the domain, since the domain logic reflects the business domain.
Why we do not choose other patterns	• For the transaction script, the main reason that we do not choose it is that it tends to result in duplicate code. Even we could use sub-routines to factor out code duplication, but there is another problem that common behavior is not easy to spot. • For the table module, the main reason that we do not choose it is that its scalability and extensibility is not very well.
How to embody in our code	• There is a single instance handling the business logic for all rows in a database table or view. In our code, the Product model would handle all business logic of all products.

Patterns of Data Source Layer

Active Record	
Why we choose this pattern	• Since our domain logic is not too complex, active record is a suitable pattern to use. • Since this pattern places logic related to the table rows in one class, it promotes high cohesion. • Since one active record objects class mimics one table structure in the database, the mapping relationship between database and the active record objects is simple and precise. • Since we have the design-time type checking, this pattern could realize the type safety. • Since by using this pattern, each model object inherits from a base active record object, we could access all the methods relating to the persist data. This feature makes it very easy to start with because it is very intuitive.
Why we do not choose other patterns	• For the row data gateway, the main reason that we do not choose it is that it requires to create a lot of boiler plate code, including new classes to act as the gateways, which would increase the effort of the maintenance. • For the data mapper, the main reason that we do not choose it requires an extra layer to act as the mapper. Since our project is not so complex and even our domain layer may need change in the future, the data source layer is also required to change according to the change of the domain, it may be not worth adopting this pattern.
How to embody in our code	• We use the rails framework, which use active record as ORM.

Identity filed: For the key representation, we use the meaningless, simple, table-unique keys and for the key generation, we use auto - generate	
Why we choose this pattern and ways of key representation and generation	• The general idea is extremely simple • The reason that we use the meaningless key is that it is sufficient to identify one object and does not have duplication problem. • The reason that we use the simple keys is that it is easy and mostly, in one table, we just have one primary key. • The reason that we use the table unique key is that it is more reasonable and can work in most cases. • The reason that we use the auto-generate is that it is easy and sufficient for creating keys.
Why we do not choose other patterns and ways of key representation and generation	• The main reason that we do not choose GUID is that generated keys are long and make little sense. • The main reason that we do not choose table scan is that the entire database must be read-locked during the function called, which may influence the performance. • The main reason that we do not choose key table is that we need to create a key table, which is not necessary.
How to embody in our code	• Each object has a database ID filed. • The keys are just integers from 1. • Each table only has one primary key. • The key is automatically generated by the database management system.

Foreign key mapping	
Why we choose this pattern	• It is simple, since the mapping between the domain relationship and database tables can be done mechanically.
How to embody in our code	• We use the foreign key to embody the domain relationship like one order may have multiple order items and in this situation, each order item in this order has the id of this order.

Embedded value	
Why we choose this pattern	• The reason that we use this pattern is that it results in a much neater relational database
How to embody in our code	• In our design, each user should have a unique cart, however, since the cart does not have other attributes except the the foreign key of the User Id and one user only has one unique cart, we decide to use the embedded value to represent the cart.

Class table inheritance	
Why we choose this pattern	• All columns in all tables are relevant, so there would be no confusion about the wasted space. • The relationship between domain model and tables in the database is simple: one table per class in the hierarchy. • Since our project would provide function to let participants to search product key words among all products, it would be efficient to have a table containing all products with basic attributes like, name. • For the disadvantage of this pattern that if we use joins or multiple queries to retrieve a single row or update rows, it would be inefficient, since we use the ruby on rails to implement, there are some brilliant solutions to solve this problem. Through using gem “active_record-acts_as” our project achieve multiple-table-inheritance (MTI) for ActiveRecord models, which allows our system to manipulate corresponding records in both super class and subclass at one time. For example, in our system Product is a super class with common attributes, and class book, cloth and snack are subclasses that inherited from product which have their own attributes. Through using “active_record-acts_as” we can simply mark class product to “actable” and other subclass to “act_as_product”, Now Book, Cloth and Snack act as Product, they inherit Products attributes, methods and validations. Manipulates against product or its subclasses will effect to their corresponding super class or subclass at the same time. Therefore, we can simply evade drawbacks of “class table inheritance” and take advantages of this pattern.
Why we do not choose other patterns	• For the single table inheritance, even there are some advantages of utilizing this pattern like, simplicity, stability, no joins, we still decide not to adopt this pattern and there are some reasons: firstly, in our project, there would be a lot of different types of products inherit from the base class Product, if we put all attributes in one table, so many files would be left null. Secondly, there is a potential risk that the additional fields may also take up space in memory. • For the concrete table inheritance, the main reason that we do not choose it is that it is not friendly for global finding. Since for the key word searching, the type of the instance is unknown, all tables must be checked, which will be inefficient. Besides, since attributes of one type of products may change in the future, if it is a super class, all subclasses of this class are required to change.
How to embody in our code	• In our project, we use inheritance for various kinds of products. The base class is the Product, and there are some subclasses, like Book, Cloth. The Product would have fundamental attributes that all products have and subclasses would have specific attributes according to their types. We divide the field information over several tables and each class would map to one table in the database. The attributes of one model class map to the attributes of the table in the database.

Sessions

Client Session State - Cookie	
Why we choose this design	• Although there are some constraints by using cookie to store session state like the length limit(about 4kb), the cookie sent every request and others, the main reason that we choose it is that in our project, we just need to store small data like current user id and others and using cookie is simple and could meet our demands properly.
Why we do not choose other patterns	• The main reason that we do not choose the server session state and the database session state is that they are more complex than cookie and may need additional hardware support.
How to embody in our code	• We use a gem called devise to implement the user module, which using the cookie to store the current user id.

Concurrency

Optimistic offline lock	
Why we choose this design	• The pattern is easy to implement, since we just need to add an extra column in the table. • It allows concurrent access from other processes, which is important in our project, since it could improve the user experience. • By using ruby on rails, we could avoid the danger about losing work/ data, since if there is an inconsistency happens, an exception would be thrown out and we could catch that exception and do some rescue. • Since our project is a C2C website, there would be various buyers, various sellers and many kinds of interesting products. The chance that one resource is accessed concurrently is low.
Why we do not choose other patterns	• The main reason that we do not choose the pessimistic lock pattern is that it would result in low liveness, which would impact the user experience that is vital to a commercial website. • As we have mentioned in the previous section, the change of conflict is low. • The main benefit of using this pattern is that no work/data lost which could be achieved to some extent by using optimistic lock in the ruby on rails.
How to embody in our code	• In this project, the table which requires ability of concurrency is modified by adding a column “lock_version”. This column is used to uniquely identify the table’s version. The version of a table will automatically increased by one after one time’s modification against the table. The idea of the optimistic offline lock in this project is that when a user modifying a table the user will get a ‘lock_version’, if this version is changed during storing of the user’s modification it means a concurrency issue is occurred. Therefore, the modification will be aborted.

Security

Authentication enforcer	
Why we choose this pattern	• Authentication enforcer improves the security by centralizing the authentication, since this pattern reduce the number of places in the presentation layer that the mechanism is accessed. • Authentication enforcer improves the maintainability, since the security policy is encapsulated in a centralized module which supports migration to different types of authentication. • Authentication enforcer reduces duplicate code in the system via reuse and supports reuse in other systems.
How to embody in our code	• In our project, we use the devise to authenticate. The devise could encrypt and stores a password in the database to validate the authenticity of a user while signing in. The authentication can be done both through POST requests or HTTP Basic Authentication.

Authorization enforcer	
Why we choose this pattern	• Authorization enforcer eliminates the chance for repetitive code and supports greater reuse by encapsulating disparate access-control mechanisms through common interfaces. • Authorization enforcer improves the maintainability by supporting a change to new authorization policies and if the business policies change, it is easy to change permissions of roles. • Authorization enforcer partitions the authentication and authorization responsibilities.
How to embody in our code	• In our code as a ruby on rails project, a gem 'role_model' is introduced to support role features. The user model is modified to support role features as well. Additionally, table user is modified by adding a column 'roles_mask', this column determines the role of a specific user. In this project, there are two roles, one is 'participant' who can sell and purchase goods, another role is administrator who can handle the cancelation request about a order. The administrator features can only be seen by a signed-in administrator user[1]. Furthermore a unauthorized user should not be able to access admin feature through malicious url[2]. After the role is assigned to users, our project now have capability to justify whether current user has authorization to access administrator features. These authorization process can be put in a single place. • [1]In terms of prohibit showing features to unauthorized user, the super class 'erb' file '_header' is modified to adding capability of determine items to show according to different roles. • [2]In terms of prohibit malicious url access, the super class of controller ApplicationController is modified to adding capability of verifying access authorisation of current user.