Eötvös Loránd University

Faculty of Informatics

Department of Programming Languages and Compilers

High concurrency purchase system for e-commerce website's flash sale

Supervisor:
Dr. Viktória Zsók
Assistant Professor

Author:
Tang Shaozhe
Computer Science BSc

Budapest, 2022

EÖTVÖS LORÁND UNIVERSITY

FACULTY OF INFORMATICS

Thesis Registration Form

Student's Data:

Student's Name: Tang Shaozhe

Student's Neptun code: QS4OZ8

Course Data:

Student's Major: Computer Science BSc

I have an internal supervisor

Internal Supervisor's Name: *Zsók Viktória*

Supervisor's Home Institution: *ELTE IK PLC*

Address of Supervisor's Home Institution: *H-1117 Budapest, Pázmány Péter sétány 1/C*

Supervisor's Position and Degree: *Assistant Professor, Ph.D.*

Thesis Title: High concurrency purchase system for e-commerce website's flash sale

Topic of the Thesis:

(Upon consulting with your supervisor, give a 150-300-word-long synopsis os your planned thesis.)

Nowadays, there is a growing demand for consumers to purchase goods online. The purpose of online shopping can be for convenience, but multi-choice, low price and high-cost performance quality must be considered. Meanwhile, as for the merchants, they will likely attract more potential customers and advertise their goods by doing a sales promotion. Some consumers may not get their idea when there are some big sales promotions in the online markets.

This software will provide a good user experience for the consumers when they want to rush to purchase their ideal goods at a low price. This software will be able to purchase the goods at a low price, be stable, and solve the oversold problems when faced with high concurrency situations. To achieve these goals, the front-end web needs to withstand heavy traffic, while the back-end interface has to manage the high concurrency. The project would also scale horizontally.

The users will be able to execute operations which then will be processed by the software to achieve the users' goals. The software is planned to provide such functions as foregrounding, user login, goods exhibition, and background order management. To build such a system, I will use Go as the programming language and Iris as the web framework, MySQL and Redis for the databases, RabbitMQ for the message queue, and Docker for the service deployment. Combining these technologies, the project will address especially heavy traffic and high concurrency.

Budapest, 2021. 06. 01.

Contents

Chapter 1

Introduction

There are growing software requirements of high currency, high performance, and high availability with the development of information technology. When we are faced with such situations, how we can solve the technical requirements gets more and more important and popular nowadays. Recently, many IT lead companies have decided to use Go [1] in software infrastructure and business systems. Go supports millions of TCP [2] connections and costs low performance and memory consumption. Besides, Go supports concurrency from a programming language layer, using goroutines and channels [3]. The above Go features can help a lot when developing a high current software. With a properly good software architecture, we do not need to write many codes to cope with the large data flow in the high current situation, which justify the Go philosophy "Less is exponentially more" [4]. Similarly, a good selection and architecture in software development are likely to achieve more technical requirements with less cost of server resources and software development, operation, and maintenance.

Online shopping has become more and more popular nowadays with the development of society. Since there is no actual door to the online shops, the online shops can generally accommodate more shopping customers than the physical shops. However, we can feel how crowded it is at a shopping mall during a shopping festival, but can we imagine that the online shops might stand with more accommodating pressure with billions of customers at a time? If we want to avoid the online shops crashing or shutting down accidentally, we need to build a high current system for such websites so that they can stand with big data flow visiting and keep stable working all the time.

A high concurrent purchase system for e-commerce website flash sales will be developed to handle high current situations and deal with big data flow traffic in this thesis. Besides, it will be able to deal with oversold problems and have the features to scale out. With specific tools, effective algorithms, and nice software architecture, we will be able to implement the above functions and meet the requirements.

As the background and motivation of the software have been illustrated above, in the following chapters, we will get to know the software development process from a user and developer perspective in the second and third chapters.

In the second chapter, user documentation will be introduced with the software motivation and requirements. Besides, the main methods and tools will be discussed concerning the back-end, front-end, environment, and operations. The user will be able to run a successful high concurrency system with the detailed methods and procedures and the running result to check.

In the third chapter, the developer documentation will be about the shopping systems for the back-end administer and front-end customer. Firstly, the 3.1 Chapter is about developing an e-commerce website with the basic functions. Then we do optimization (3.2 Chapter) to equip the system with high concurrency and scaling out features. We will do testing to check if our system is usable, stable, secure, and meets all the requirements.

Chapter 2

User Documentation

This chapter will provide a short description of the requirements analysis. Besides, we will introduce the main methods and tools used in the software project with detailed information on their usage. The reader will understand the software and learn how to operate it with the correct instruction.

2.1 High Concurrency Purchase System

There is a growing tendency for people to shop online with the development of the internet. Generally, shopping online will provide more convenience and choices for the customers. Besides, customers are likely to get a lower price than the same products in the physical stores.

When shopping festivals like Black Friday or Christmas day come, the products generally have valuable discounts in all stores (physical and online). As situations where the physical stores are crowded always happen during such festivals, can we assume that the online stores are way more crowded? The answer is an absolute yes! The shopping websites will have much more visitors and customers with the attraction of high discounts on product sales. Then comes the second question, how does the shopping website deal with the big data traffic, and how does the shopping system deal with the huge orders?

This software will solve the above problems to deal with big data flow traffic and solve the oversold problems to the high current situations.

2.2 Main Tools and Methods

2.2.1 Back-end

The back-end system is designed with the Go [1] programming language with the Iris [5] framework of Model-View-Controller (MVC) [6] pattern and big databases with MySQL [7] and Redis [8]. The RabbitMQ [9] deals with the message queues [10] for message communication(in the send and receive modes).

Go

Go (also called Golang or Go language) is an open-source, multiparadigm, statically typed, compiled programming language designed by Google. Go, unlike other programming languages, provides easy and trackable concurrent features, which makes it convenient for application developers to interact with the requests among allocated resources, servers, databases, and networks. Besides, it provides a powerful standard library and useful tools like Gofmt, Gorun, Godoc. Go also has garbage collection features and supports testing capacities.

Iris Framework

Iris is a web framework that serves as a convenient and efficient cross-platform with a robust group of features. With Iris, we can build our public or private web applications and APIs with high performance, portability, and potential.

As an open-source Go web framework, Iris was created by Gerasimos Maropoulos for modern web applications development. With the assistance of the Go Iris framework, software developers are very likely to create super-fast web applications with less effort.

Compared to other go modules, Iris supports developers with upper-class model-view-controller MVC architectural pattern in web applications.

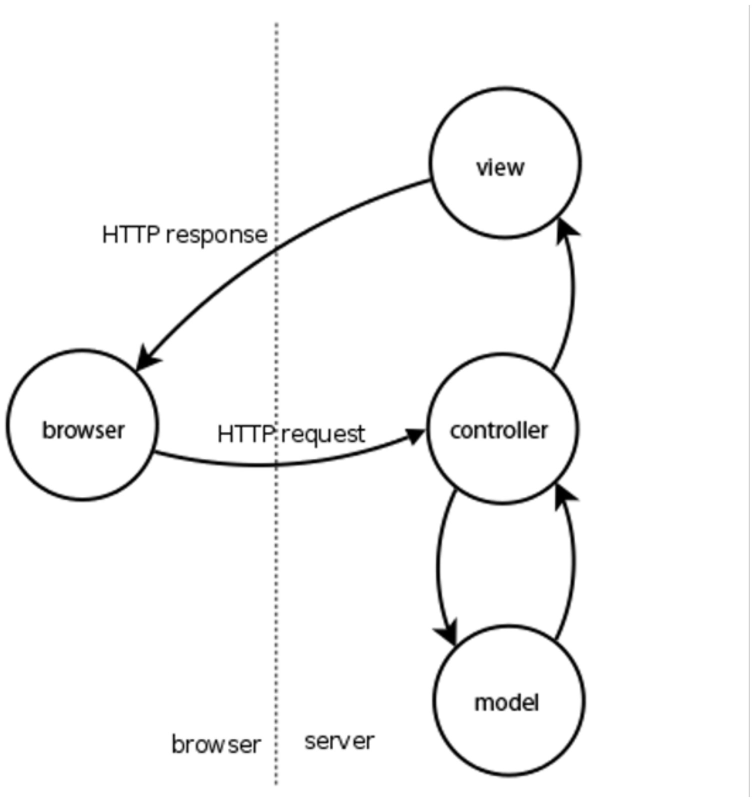


Figure 2.1: The structure of MVC

To use this MVC (Model View Controller) architectural pattern, You just need to import the MVC package in iris.

RabbitMQ

RabbitMQ is a popular open-source message broker as message-oriented middleware. It was developed to support the Advanced Message Queuing Protocol (AMQP) and subsequently expanded with a plug-in architecture to support Text Oriented Messaging Protocol STOMP, MQ Telemetry Transport (MQTT), and other protocols.

We need to install the abmq package to use the RabbitMQ. As the below diagram shows, "P" works as a producer, and "C" works as a consumer. The middle box is a queue, serving as a message buffer [10] that RabbitMQ keeps representing the consumer.

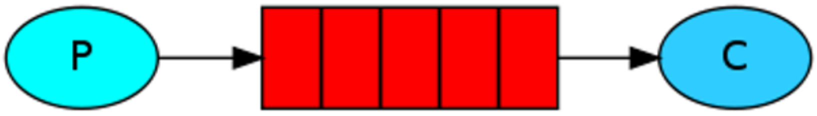


Figure 2.2: RabbitMQ

MVC

When we are learning about programming or in a software development process, common situations occasionally happen where we find some code bases are easy to check. At the same time, some are confusing and hard to follow, maintain, and update. In most cases, it depends on the organization and structures of the code [6].

Knowing the related part of codes will be greatly helpful once we would like to add new functions or features, fix a bug, or do any other operations on the program. If the program structures are well designed and presented, we are likely to guess the location of the codes to operate even though we have never seen them. MVC, which represents Model-View-Controller, is a software design pattern mainly used to develop graphical user interfaces (GUIs) and design web applications.

Model [11] is the central part of the MVC pattern, which presents the dynamic data structure and displays the application's user interface. It directly connects and manages the application's data, logic, and rules. [12] Typically, it communicates with the databases while interacting with data from other APIs or services in other cases.

View [13] represents the information of a chart, diagram, or table.[14] The main function of views is to render data, and that is to say, given a specific page with related data to render, we can use views to generate the correct output. If we want to implement an application with the MVC framework, we can return the server-rendered HTML to the end user's browser. Besides, we can also achieve that with the handle rendering data types such as XML, JSON, Etc. One important thing to notice is that we should put as little logic as possible in our codes to avoid unexpected problems. Since the view is mainly used to display data, it should keep a clear structure with loose coupling.

Controller [15] generally receives the input and converts it to commands as a signal to the models and views. Instead of directly writing to a database or establishing

HTML responses, controllers [16] will directly interact with the incoming data with the appropriate models, views, and other packages. As the main goal of a controller is to parse data and send it to other types and functions to handle, we should not put too much logic into the controller.

2.2.2 Front-end

We use HTML, CSS, and JavaScript as programming languages in the front-end development and Bootstrap as the framework.

HTML

Hypertext Markup Language (HTML) [17] is one of the most basic building components of web pages as a markup language. It defines the meaning, structure, and content of a web page. Besides HTML, we use CSS to display the appearance or presentation of a webpage and JavaScipt to describe the web page's functionality and behavior. "Hypertext" is a text used to generate links that connect web pages to other websites. "markup" is what HTML tags to annotate images, text, structure, and content of what a Web browser displays.

CSS

Cascading Style Sheets (CSS) [18] is a style sheet language that decides how HTML or XML presents in a web browser. It shows how the elements rendering is displayed on the screen [18] and can control multiple web page layouts at once.[19] As a core programming language for the open web, CSS is standardized and has three official specification versions CSS1, CSS2.1, and CSS3.

JavaScript

JavaScript (JS) [20] is an interpreted client-side script programming language with lightweight cross-platform features. As one of the most popular programming languages, JS is widely used in web development. Besides, it is also very popular in non-web browser environments software development such as Node.js, Apache CouchDB, and Adobe AcrobatBy. [21] What is more, JS can be used in both client-side and server-side development. On the client-side, the objects will be able to control a browser or Document Object Model (DOM). For example, useful client-side libraries like AngularJS, VueJS, and ReactJS can be used to place the HTML form elements and interacts with the user events.

Bootstrap

Bootstrap [22] is a powerful toolkit that collects HTML, CSS, and JavaScript tools for web development. It is hosted on GitHub and created by Twitter as an open-source and user-friendly framework. The advantage of responsive design keeps the browser working stably and can be designed consistently with re-usable components. With a rich extension in JavaScript, Bootstrap supports build-in functions for jQuery plugins and JS APIs. Since we can use Bootstrap in any IDE or editor, it provides web designers and developers with much convenience and interest to work.

2.2.3 Environment

The development environment generally serves as a workstation where developers can build and develop their programs. Here we will show detailed software development environment requirements for users to conduct. The specification of the system used in the project will be shown as follows:

- Operating System: Windows 10
- Ram: 16.0 GB
- CPU: 1.80 GHz
- Code Editor: Goland
- Go version: go1.15.8 windows/amd64
- Framework: iris12

2.2.4 Environment Preparation

The user must download the following application and packages to set up the environment and use this software.

Go download address: https://go.dev/dl/

Goland download address: https://www.jetbrains.com/go/download. Students can register for an educational account and use the free service during their studies.

RabbitMQ download address: https://www.rabbitmq.com/download.html

MySQL download address: https://www.mysql.com/cn/downloads

Navicat download address: https://www.navicat.com/en/products

2.2.5 Install and Run the Project

1. Install the project
2. Prepare MySQL database for the project: The database setting should be specified. There are many ways to create a specific database. Here we only introduce the way of creating a database by using MySQL Command Line Client. When we download the MySQL database, we set the root password to 1. In this case, we enter the database command prompt and make the commands create the database.
3. To run the project: We have to run the following codes together to see the whole interface and check the core functions of this software to run the project successfully.
4. To run RabbitMQ.
5. To run MySQL.
6. To run backend management system: Open a terminal, go to the backend directory, and run the main file (for the backend deployment).
7. To run fronted functions: Open a terminal, go to the frontend directory, and run the main file (for the user login).
8. To run getOne.go (Product quantities reduction, and avoid the oversold problem).
9. To run consumer.go (For the user consumption).
10. To run validate.go (Used for the consistent has, security verification, successful purchase validate).

2.2.6 Deployment of Docker

1. We need to compile the main file and getOne of the front and back ends into an executable Linux binary program. We need to set the GOARCH environment variable and GOOS environment variable, Then use the go build command to compile the program.

2. We need to download a net Ubuntu container, log in and copy our binary files into that container.

3. The command of cp in docker will be used to move our binary files in the last step to the net Ubuntu container.

4. The container with our project should be imported by the export command in docker. The export result will be named product.tar.

5. Lastly, we will use the import command in docker to import product.tar as a new docker image, which can be downloaded and implemented in other machines without extra configurations.

2.3 The Run Result of High Concurrency Purchase System

This section will show screenshots of the high concurrent purchase system after running. The results of both user and administrator runs will be displayed as follows.

2.3.1 The Perspective of User

If a user wants to access our highly concurrent shopping site, a legal user identity registration must be done first. Users need to submit their nickname, username, and password to complete the registration, as shown in the Figure 2.3.

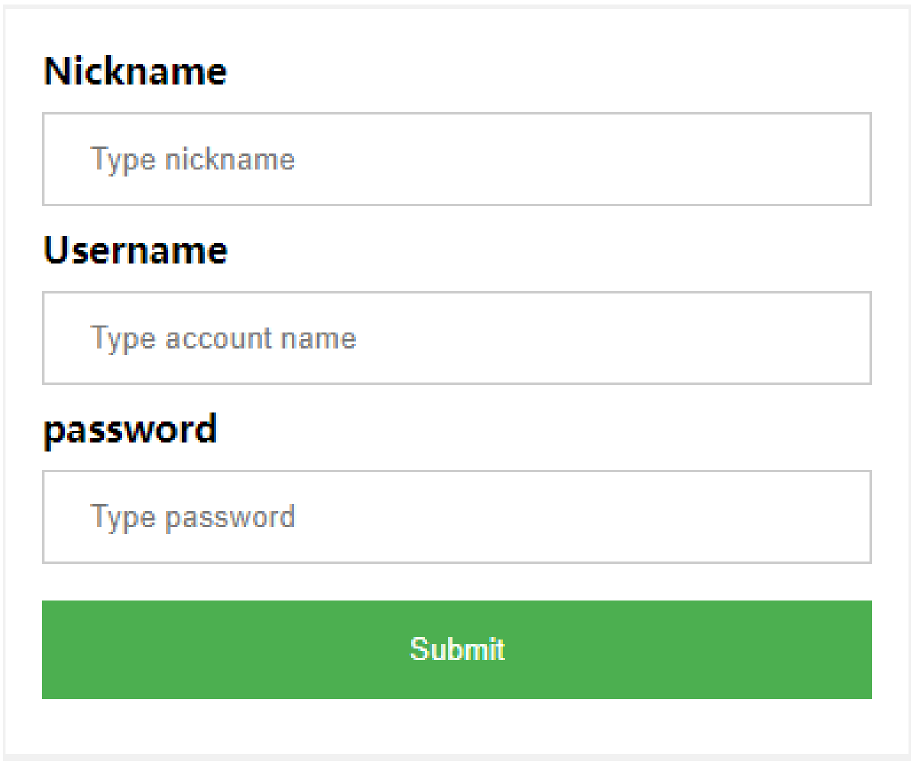


Figure 2.3: The figure of user register

The user proceeds to the login page and enters their account number and password to log in after completing registration. The back-end system will check and determine if the user is legitimate, and the corresponding logic will be shown later.

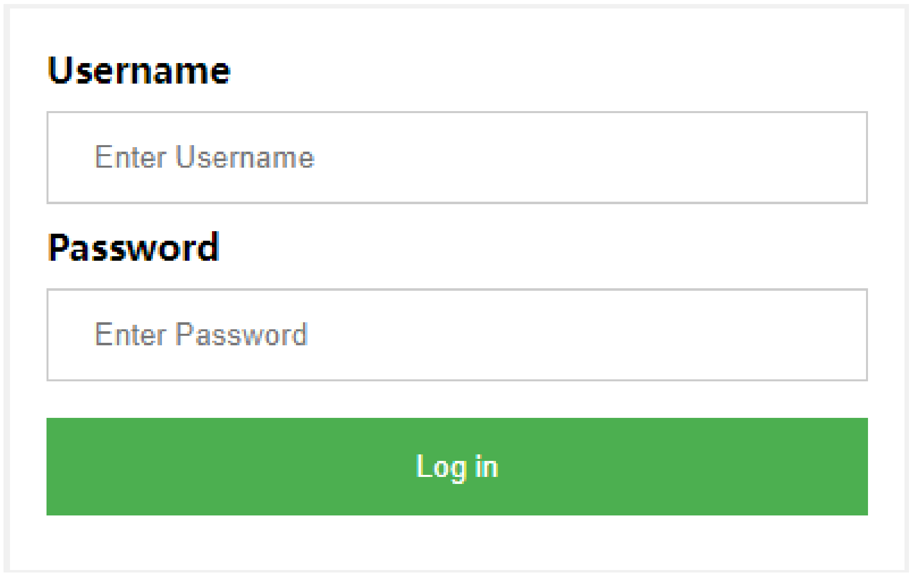
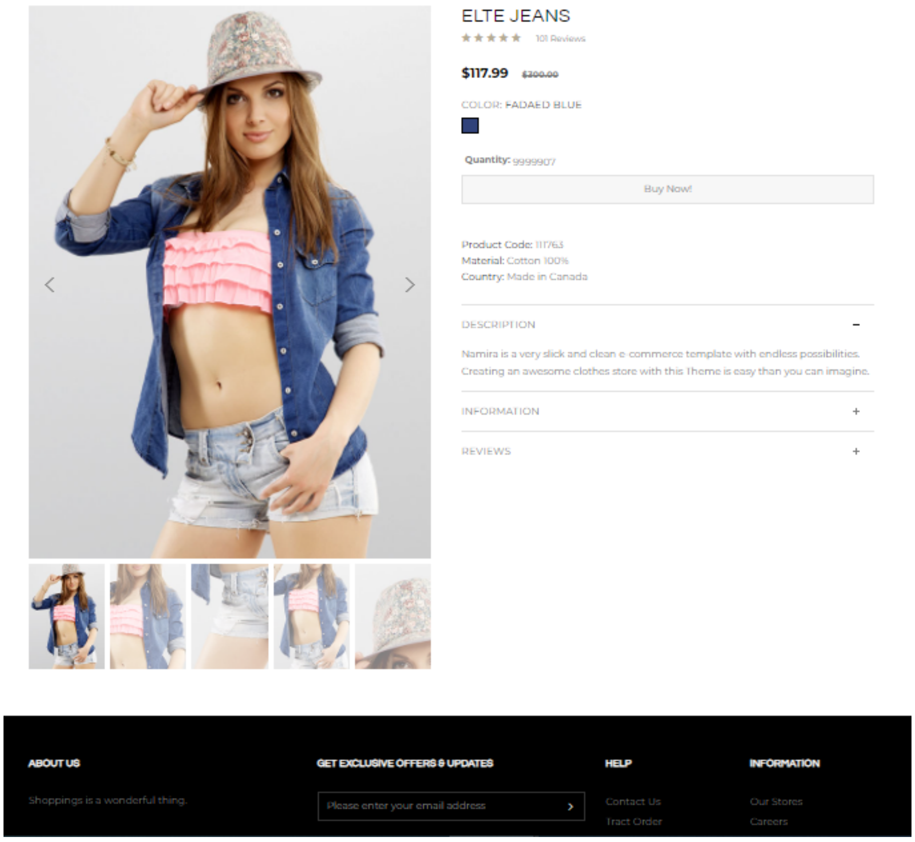


Figure 2.4: The figure of user login

The logged-in users will see the product details as shown in the Figure 2.5, and after observing the product details, they can click the button "Buy Now!" to purchase.


ELTE JEANS

★★★★★ 101 Reviews

$117.99 ~~$200.00~~

COLOR: FADAED BLUE

Quantity: 9999907

Buy Now!

Product Code: 111763
Material: Cotton 100%
Country: Made in Canada

DESCRIPTION −

Namira is a very slick and clean e-commerce template with endless possibilities. Creating an awesome clothes store with this Theme is easy than you can imagine.

INFORMATION +

REVIEWS +

ABOUT US

Shoppings is a wonderful thing.

GET EXCLUSIVE OFFERS & UPDATES

Please enter your email address

HELP

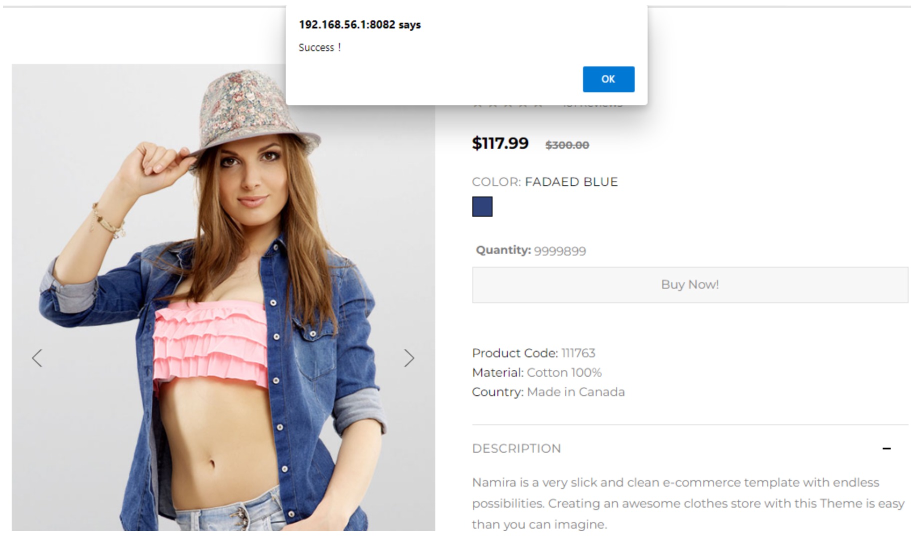
Contact Us
Tract Order

INFORMATION

Our Stores
Careers

Figure 2.5: The figure of product shopping page

The front end will have a 10 seconds interval purchase restriction to the next purchase request, as shown in the Figure 2.6.


192.168.56.1:8082 says

Success !

OK

$117.99 ~~$300.00~~

COLOR: FADAED BLUE

Quantity: 9999899

Buy Now!

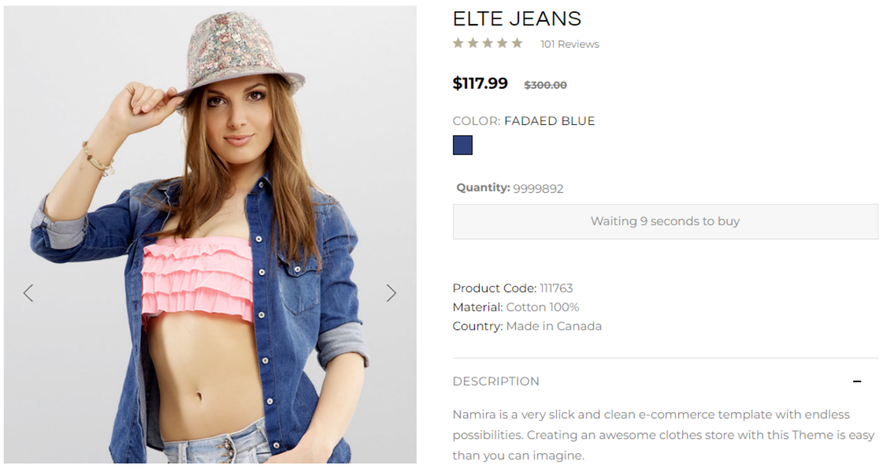
Product Code: 111763
Material: Cotton 100%
Country: Made in Canada

DESCRIPTION −

Namira is a very slick and clean e-commerce template with endless possibilities. Creating an awesome clothes store with this Theme is easy than you can imagine.

Figure 2.6: The figure of a successful purchase

The front end will restrict the user's next purchase request until the "Buy Now!" timer button exceeds 10 seconds, as shown in the Figure 2.7.



ELTE JEANS

★★★★★ 101 Reviews

$117.99 ~~$300.00~~

COLOR: FADAED BLUE

Quantity: 9999892

Waiting 9 seconds to buy

Product Code: 111763
Material: Cotton 100%
Country: Made in Canada

DESCRIPTION –

Namira is a very slick and clean e-commerce template with endless possibilities. Creating an awesome clothes store with this Theme is easy than you can imagine.

Figure 2.7: The figure of a pause for buy

2.3.2 The Perspective of Administrator

The administrator will be able to manage the products and orders in the system after running the main function in the backend directory as shown in the Figure 2.8.

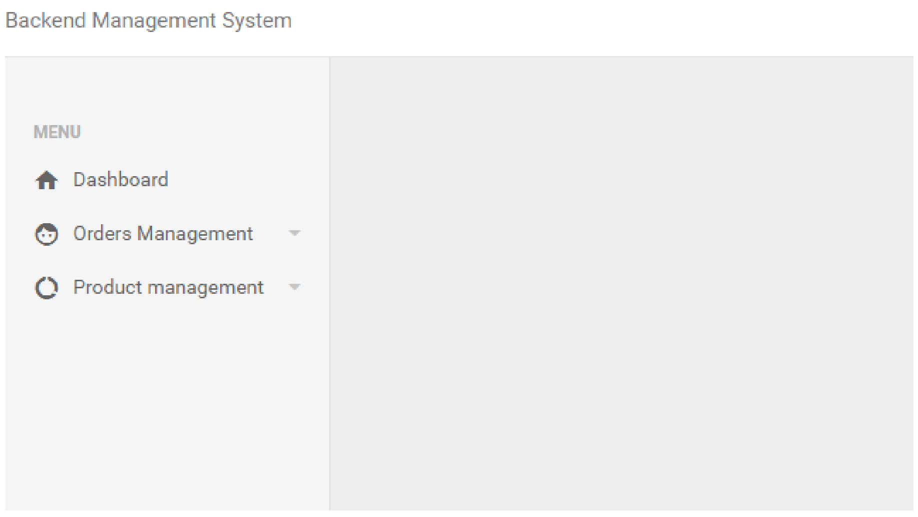
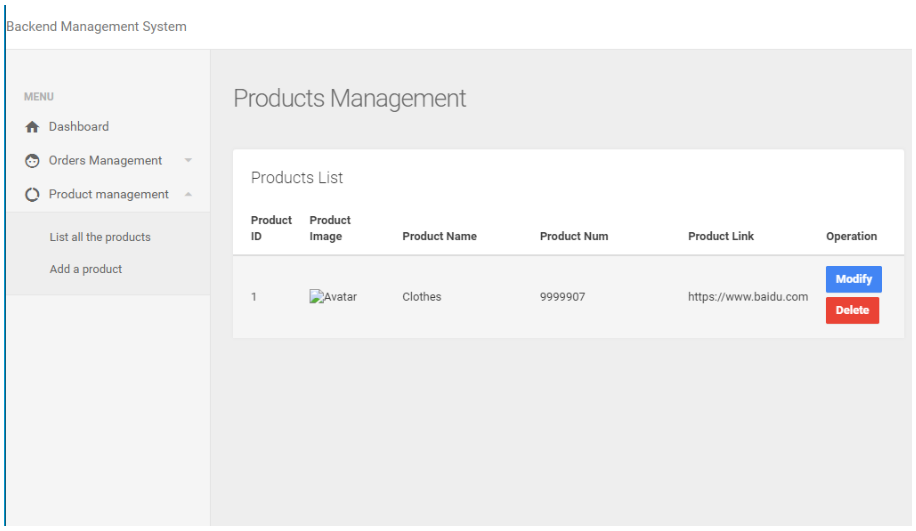


Figure 2.8: The figure backend interface

The administrator will see the list of products after clicking the product management button, which contains information about each product, such as product ID, product number, product picture, Etc. The administrator has the power to modify the product information.



Product ID	Product Image	Product Name	Product Num	Product Link	Operation
1	Avatar	Clothes	9999907	https://www.baidu.com	Modify Delete

Figure 2.9: The figure of product management

Administrators can add products by clicking the Add button. The product information includes product name, product number, product image address, and product link. As shown in the Figure 2.10, the administrator can type information in the blank and click the Add button to complete a product adding operation.

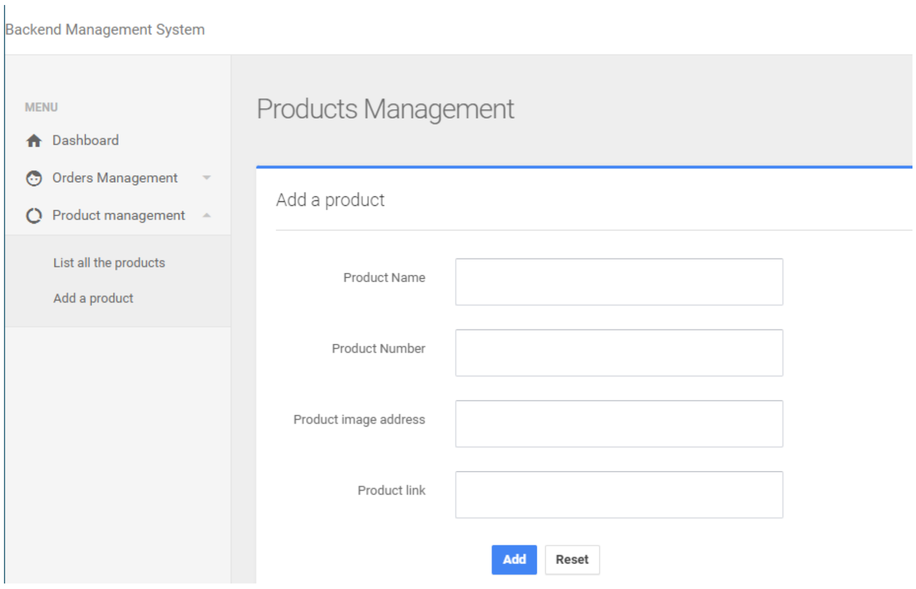
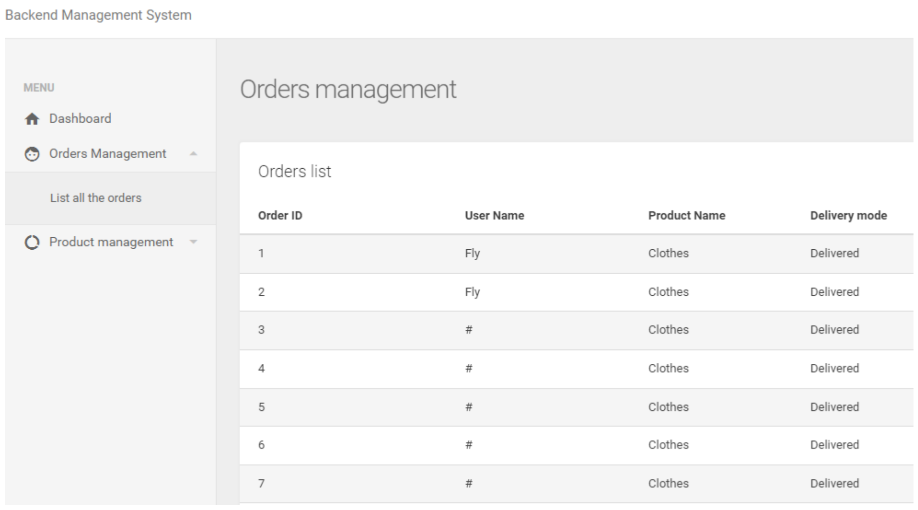


Figure 2.10: The figure of add product

The administrator will be able to view all order information after clicking the "List all the orders" button under the orders management block as shown in the Figure 2.11.



Orders list

Order ID	User Name	Product Name	Delivery mode
1	Fly	Clothes	Delivered
2	Fly	Clothes	Delivered
3	#	Clothes	Delivered
4	#	Clothes	Delivered
5	#	Clothes	Delivered
6	#	Clothes	Delivered
7	#	Clothes	Delivered

Figure 2.11: The figure of order

To conclude, we have shown the interfaces of every page on both the front-end and back-end. The customers have to register their accounts before shopping to enable users a better shopping experience and protect the web from being attacked. Besides, the front-end registration and login process will restrict the invalid visits with big data flow traffic sending many invalid requests, which will decrease the pressure on the server and databases. The valid customers will be able to shop and purchase their products with a 10 seconds interval restriction, and this is to control big data flow and protect other customers' purchase chances. The administrators will be able to operate on the products, like adding a new product, checking and deleting an existing product. Besides, the administrators can also operate the databases through MySQL to operate on the data.

The user documentation has been completed by now, and we introduced this software's background and motivation from a user perspective. Since the methods, tools, and running use cases are presented, The potential users will be able to know how to operate properly with the sufficient requirements fulfilled. The following chapter will introduce the detailed development process from a developer's perspective regarding the software's motivation, requirements analysis, and software architecture.

Chapter 3

Developer Documentation

This chapter is different from the user documentation, and we will introduce the development process from a developer's perspective. As the software requirements have been analyzed, we will start developing the software from the back-end management and front-end shopping parts with the Iris framework and MVC pattern. Moreover, the optimization development will equip the software with the features of standing with large data flow, high currency, and scaling out. The testing part will check if our software successfully meets all the requirements.

3.1 Development of the High Concurrency Purchase System

3.1.1 Back-end Products Management Development

In this chapter, we start developing this software by doing the development of the back-end products management system, which consists of three parts as :

1. Products models design development.
2. Products functions development (with insert, delete, modify, and check four basic functions).
3. Back-end products management interface.

We use the Iris framework with the MVC model and the Go programming language to design the back-end products management system.

We follow the MVC framework to go as follows to create the directory of the software.

Create the Directory Structure

Firstly, we put back-end product management functions in back-end/main.go. Inside the web directory, we put the controllers associated with the assets (css, img, js, lib) and views where the HTML locates. Secondly, we create the datamodels directory as the MVC model for the product models.

```
|
|__ backend
|   |
|   |__ web
|   |   |
|   |   |__ assets
|   |   |__ controllers
|   |   |__ views
|   |
|   |__ main.go
|__ data
|__ repositories
|__ services
```

Create data model

We create a models.go file to code the product's structure after creating the models directory. Assign the product ID, name, number, and image, URL with essential variables and types, then the product models part is done. The E-R diagram of the objects designed for our system is detailed in the Figure 3.1.

```go
package models

type Product struct {
  ID           int64  ‘json:"id"‘
  ProductName  string ‘json:"ProductName"‘
  ProductNum   int64  ‘json:"ProductNum"‘
  ProductImage string ‘json:"ProductImage"‘
  ProductUrl   string ‘json:"ProductUrl"‘
}
```

Code 3.1: models.go

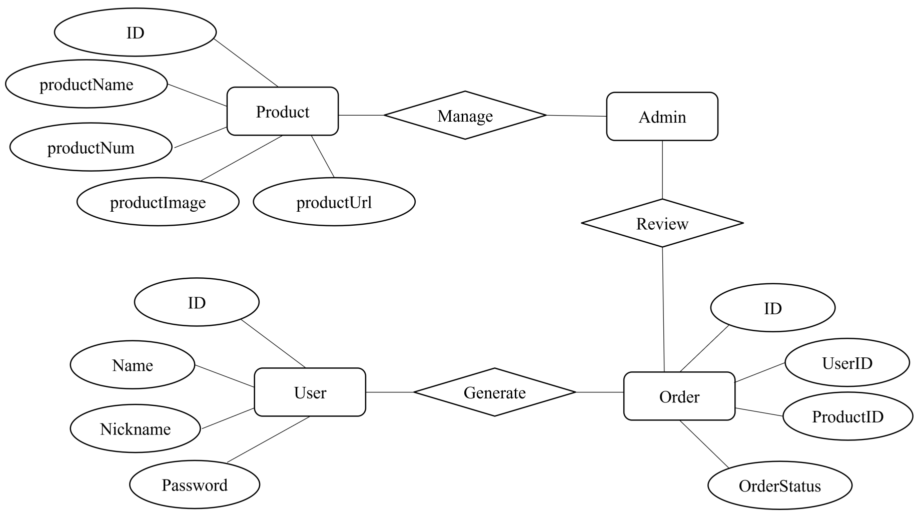


Figure 3.1: The E-R diagram

Create data interface

We need to create the interface for the development and then implement the related interface to create the database structure of the product. We add functions like insert, delete, update, and select after connecting to the databases.

```
type IProduct interface {
  Insert(*models.Product) error
  Delete(int64) bool
  Update(*models.Product) error
  SelectByKey(int64) (*models.Product, error)
  SelectAll() ([]*models.Product, error)
  SubProductNum(productID int64) error
}
```

Code 3.2: The interface of product

An interface is a type defined as a set of methods signatures, which is a generalization and abstraction of the behavior of other types. An interface is an abstract type that does not expose internal operations based on this precise layout.

The features of interface can be summarized as follows:

- An interface has only method declarations, no implementation, and data fields.
- Interfaces can be embedded anonymously into other interfaces or structs.

An interface is a type used to define behavior. These defined behaviors are not implemented directly by interfaces, but by user-defined types, and a concrete type that implements these methods is an instance of this interface type.

If a user-defined type implements a set of methods declared by an interface type, then the value of the user-defined type can be assigned to the value of the interface type. This assignment will store the value of the user-defined type into the interface type.

Create an initialization function

However, the implementation of the interface will not disappear in the Go programming language. We need to create a constructor function to check if the codes have implemented the needed structure as the following NewProductManager function.

```go
func NewProductManager(table string,db *sql.DB) IProduct{
  return &ProductManager{table: table,mysqlConn: db}
}
```

Code 3.3: NewProductManager

Create service logic based on the database operation structure

If we want to develop database service logic, the first step is to develop the corresponding interface. Inside the product interface, we connect to the database and add the relative insert, delete, update and select functions. After the interface is defined, we need to implement the interface.

```go
package services

import (
  "product/data"
  "product/repositories"
)

type IProductService interface {
  GetProductByID(int64) (*data.Product,error)
  GetAll() ([]*data.Product,error)
  DeleteProductByID(int64) bool
  InsertProduct(product *data.Product) (int64,error)
  UpdateProduct(product *data.Product) error
  SubProductNum(productID int64) error
}

```

```go
type ProductService struct {
  productRepos repositories.IProduct
}
```

Code 3.4: IProductService

Controller development

We have developed the models and views in the MVC architecture from the above. Now, the next step is to develop the controller part. After creating the controller's package, we create a ProductController structure to put the iris context and product service inside. Then we need to the action of the structure, that is, to get the products for the users. When a user is visiting our website, he or she probably wants to check the detailed product information and do shopping. Then the detailed product information and functions should be listed. Inside the controller, the views should be rendered to implement the above needs.

```go
type ProductController struct {
  Ctx iris.Context
  ProductService services.IProductService
}

func (p *ProductController) GetAll() mvc.View{
  products,_ := p.ProductService.GetAll()
  return mvc.View{
    Name: "product/view.html",
    Data: iris.Map{"productArray":products},
  }
}
```

Code 3.5: GetAll

Register the controller and connect it to the database in the backend/main.go.

```go
productRepository := repositories.NewProductManager("product",db)
productService := services.NewProductService(productRepository)
productParty := app.Party("/product")
product := mvc.New(productParty)
product.Register(ctx,productService)
product.Handle(new(controllers.ProductController))
```

Code 3.6: The register of product

Write the template file of the MVC model template

When creating a dynamic website or showing the user the customized output, we usually use the Golang template to implement. Generally, Golang has two template packages:

- text/template - it provides the needed functions to parse our program.
- html/template - it provides packages to support template rendering.

Generally, both these two template can generate the same interface while the html/template is safer as it implements data-driven templates to generating the output of HTML.

```
return mvc.View{
    Name: "product/manager.html",
    Data: iris.Map{
        "product": product,
    },
}
```

Code 3.7: MVC view

3.1.2 Back-end Orders Management Development

In the last chapter, we have already demonstrated how to develop the product back-end management system. In this chapter, we continue to develop the order back-end management system, which is similar to the product back-end management system, follows the MVC architecture, and has the three parts as below:

1. Orders models design development.
2. Orders functions development (with insert, delete, modify, and check four basic functions).
3. Back-end Orders management interface.

Create the orders management model in datamodels

We create a models.go file to code the structure of the product after creating the models directory. Assign the order ID, UserId, ProductID, and OrderStatus with an int64 type, then the order models part is done.

```
type Order struct {
  ID int64 ‘sql:"ID" order:"ID"‘
  UserId int64 ‘sql:"userID" order:"UserID"‘
  ProductId int64 ‘sql:"productID" order:"ProductId"‘
  OrderStatus int64 ‘sql:"orderStatus" order:"OrderStatus"‘
}
var(
  OrderSuccess int64 = 0
  OrderFail int64 = 1
)
```

Code 3.8: Order data model

Create order model database operation structure

Developing the order database service logic is similar to what we have done to the product, and the first step for us is to develop the corresponding interface. Inside the product interface, we connect to the database and add the relative insert, delete, update and select functions. After the interface is defined, we need to implement the interface.

```
type IOrderRepository interface{
    Insert(*models.Order) error
    SelectAllWithInfo() (map[int]map[string]string,error)
}

func NewOrderManagerRepository(table string) IOrderRepository{
  return &OrderManagerRepository{
    table: table,
  }
}

type OrderManagerRepository struct {
  table string
}
```

Code 3.9: Order interface

Write service logic based on the database operation structure

We need to implement the interface with the relative functions on the order service after creating the order constructor function. Firstly, we need to connect to the database using a Conn() function. Then, we do the basic insert, delete, update, and select functions.

```go
func (o *OrderMangerRepository) Insert(order *models.Order) error {
    sql := "INSERT ‘order‘ set userID=?,productID=?,orderStatus=?"
    err := Product.Exec(sql, order.UserId, order.ProductId, order.OrderStatus).Error
    if err != nil {
        log.Log.Error(err)
        return err
    }
    return nil
}

func (o *OrderMangerRepository) SelectAllWithInfo() (OrderMap map[int]map[string]string, err error) {
    sql := "SELECT\n\to.ID,\n\tu.userName,\n\tp.productName,\n\to.orderStatus \nFROM\n\t‘order‘ AS o\n\tLEFT JOIN product AS p ON o.productID = p.ID\n\tLEFT JOIN user AS u ON o.userID = u.ID"
    rows, err := Product.Raw(sql).Rows()
    if err != nil {
        return nil, err
    }
    return GetResultRows(rows), err
}
```

Code 3.10: Order databse operation

Controller development

The order controller development is similar to the product. We create an OrderController structure to put the iris context and product service inside after creating the controllers’ package. Then we need to the action of the structure, that is, to get the orders for the managers. Inside the controller, the views should be rendered to implement the above needs.

```
type OrderController struct {
  Ctx          iris.Context
  OrderService services.IOrderService
}

func (o *OrderController) Get() mvc.View {
  orderArray, err := o.OrderService.GetAllOrderInfo()
  if err != nil {
    log.Log.Errorf("Failed to check orders info %s", err.Error())
    return mvc.View{}
  }
  return mvc.View{
    Name: "order/view.html",
    Data: iris.Map{
      "order": orderArray,
    },
  }
}
```

Code 3.11: The MVC of order

Develop the register the order controller in the main function.

```
  orderRepository := mysql.NewOrderMangerRepository("order")
  orderService := services.NewOrderService(orderRepository)
  orderParty := app.Party("/order")
  order := mvc.New(orderParty)
  order.Register(ctx, orderService)
  order.Handle(new(controllers.OrderController))
```

Code 3.12: The MVC register of order

The order template is developed the same way as the product.

```
  return mvc.View{
    Name: "order/view.html",
    Data: iris.Map{
      "order": orderArray,
    },
  }
```

Code 3.13: The template of order

We have finished the back-end management development of the products and orders by now. After running the backend/main.go, we will be able to see a backend management interface.

```go
addr := host + ":18080"
err := app.Run(
  iris.Addr(addr),
  iris.WithoutServerError(iris.ErrServerClosed),
  iris.WithOptimizations,
)
```

Code 3.14: The running code of backend

3.1.3 Fronted-end Key Functions Development

In this chapter, we begin to develop the frontend functions. As the fronted pages are mainly for the customers, there should be no consideration for the manager compared with the designing of the backend interface. The user is the most important participant of the front-end, and from the user's perspective, we list the user interface and handler in Figure 3.2. Our core fronted functions development consists of three parts as follows:

1. User login interface development
2. Frontend products display functions development
3. Purchase data control development

However, this is not complete for our software which does not have the features of high currency and capacity to stand with the big data flow. We will talk bout the core system part in the later chapter, and now we begin with the basic user login interface.

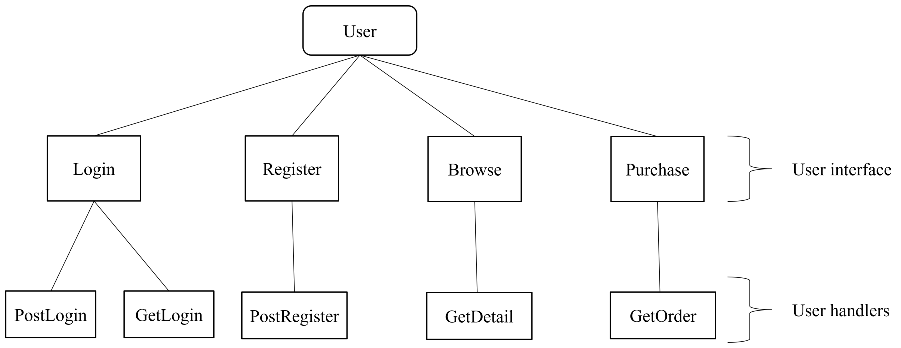


Figure 3.2: The interface and handlers of user

The implementation of AES

We will implement user login authentication in the following sections to authenticate the user. A cookie-based mechanism will be used to log the logged-in user, and since cookies are stored on the client-side, an attacker can modify them. Therefore, an AES encryption method to prevent attackers needs to be implemented before implementing the user login functionality.

We have implemented a 128bit AES padding pattern and encryption/decryption method. First, we define the AES padding byte as the length of the original text and then repeat this byte several times until the padded byte can be grouped successfully. Then call the AES interface to encrypt, and after a layer of base64 encryption as the encrypted value of the cookie sent to the client. We only list the three functions of encryption for space reasons, and decryption is the reverse process of the above steps.

Since the attacker cannot know the AES encryption key, he wants to impersonate the legitimate user identity through a fake cookie is required to enumerate the encrypted string, and this is very costly. We will give the security in the later test section after using AES encryption.

```go
func pKCS7Padding(ciphertext []byte, blockSize int) []byte {
  padding := blockSize - len(ciphertext)%blockSize
  return append(ciphertext, bytes.Repeat([]byte{byte(padding)},
     padding)...)
}

func aesEcrypt(origData []byte, key []byte) ([]byte, error) {
  block, err := aes.NewCipher(key)
  if err != nil {
    return nil, err
  }
  blockSize := block.BlockSize()
  origData = pKCS7Padding(origData, blockSize)
  blocMode := cipher.NewCBCEncrypter(block, key[:blockSize])
  crypted := make([]byte, len(origData))
  blocMode.CryptBlocks(crypted, origData)
  return crypted, nil
}

func EnCode(pwd []byte) (string, error) {
  result, err := aesEcrypt(pwd, PwdKey)
  if err != nil {
    return "", err
  }
  return base64.StdEncoding.EncodeToString(result), err
}
```

Code 3.15: The implementation of AES

User registration main process

The User login interface also follows the MVC architecture, similar to the product order management system. The detailed process of developing user login MVC will not be introduced. Now we start from the user registration main process.

Since the user interface is different from the backend pages with different functions, the static web pages, templates, and patterns we use will also be different. We code all the fronted pages into the fronted folder to distinguish the above point while the backend functions are inside the backend folder.

```go
func (c *UserController) GetRegister() mvc.View {
    return mvc.View{
        Name: "user/register.html",
    }
}
```

Code 3.16: The MVC of user register

We put all the static resources into the public folder where the CSS, fonts, js, and images are stored. Besides, we need a views folder to store the template files and the HTML files. We need to prepare a registration form and create a new Get type controller. We need to register the user with the register function, and the interface will be displayed on the register.html page after the user controller is created. Then we create a fronted/main.go to run the fronted pages.

Then, we create an accepting form controller, and the controller method name must start with Post. We can get a login interface as the following after finishing the login.html and register.html. However, we need to deal with the post request on the PostRegister() function. The execution logic diagram of register is shown in 2.3.

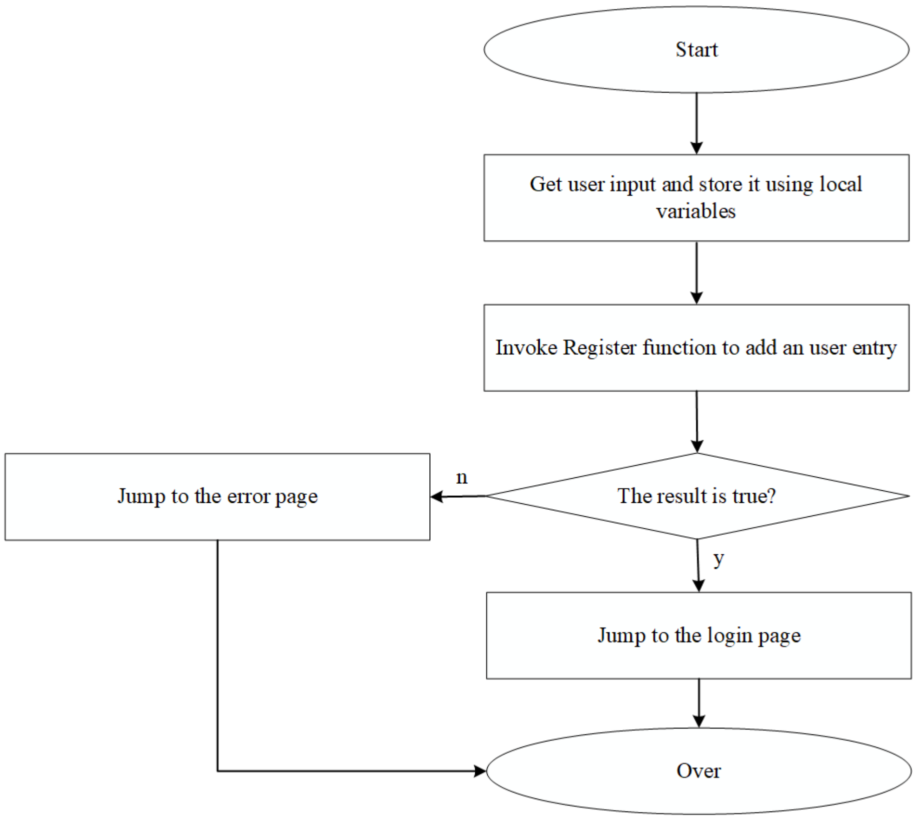

Start
Get user input and store it using local variables
Invoke Register function to add an user entry
The result is true?
n
Jump to the error page
y
Jump to the login page
Over

Figure 3.3: The execution logic diagram of register

We get the form content and create an instance of the user structure after the above operations.

```
    var (
        nickName = c.Ctx.FormValue("nickName")
        userName = c.Ctx.FormValue("userName")
        password = c.Ctx.FormValue("password")
    )
    user := &models.User{
        UserName:     userName,
        NickName:     nickName,
        HashPassword: password,
    }
```

Code 3.17: Get form content

We have done the user models, mapping relations, and services till now. The database development is not done yet, but the method is similar to the product and orders. We will encrypt the password while registering.

```go
func GeneratePassword(userPassword string) ([]byte,error){
    return bcrypt.GenerateFromPassword([]byte(userPassword),bcrypt.
       DefaultCost)
}
```

Code 3.18: Get encrypted password

User login function process

The user login process can be concluded as the following four steps: The first step will be rendering the user login page. Then user controller will be started to process the user request. In the third step, the input, including username and password, will be verified by the controller. Finally, the controller will jump to the user based on the authentication result. If the user is logged in, the user information will be encrypted and saved in a cookie to return and jump to the purchase page. If the user fails to log in, the user will be redirected to the registration page when the user ID is 0 or to the error page when the user password is not equal to the password saved in the database. The whole execution logic diagram is shown in 3.4.

```go
func (c *UserController) GetLogin() mvc.View {
    return mvc.View{
        Name: "user/login.html",
    }
}
```

Code 3.19: The MVC of login

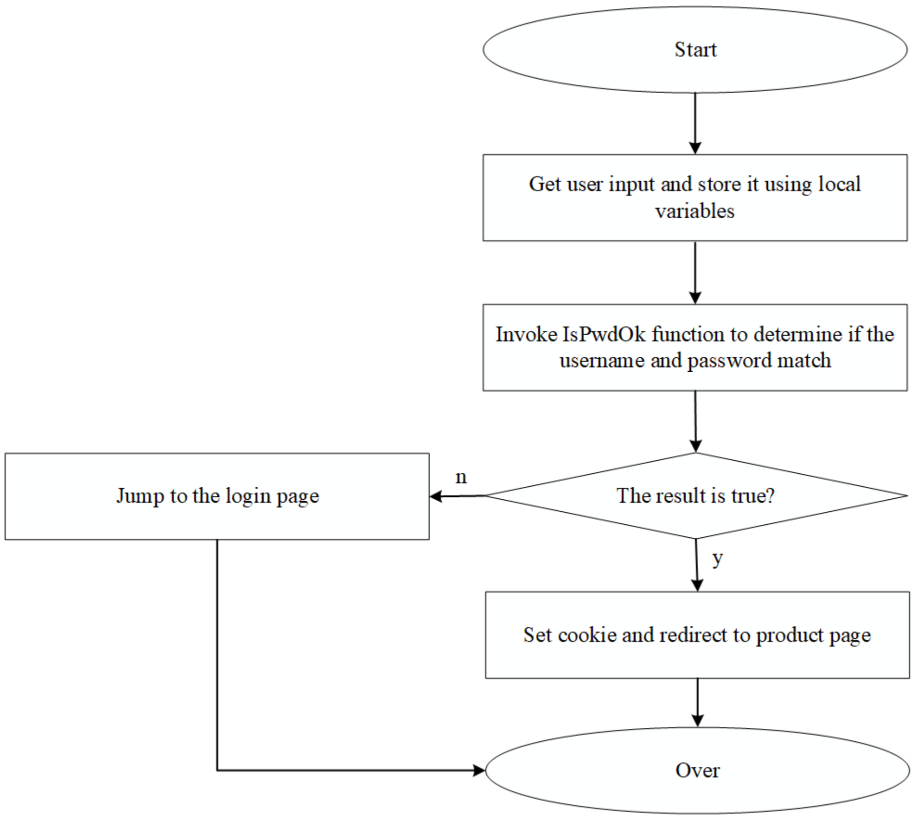

Start
Get user input and store it using local variables
Invoke IsPwdOk function to determine if the username and password match
The result is true?
n
Jump to the login page
y
Set cookie and redirect to product page
Over

Figure 3.4: The execution logic diagram of login

Product details display

We can use the back-end services to develop the fronted product details interface based on the previous back-end management development. Similarly, the development has three main parts as follows:

1. Query information based on the product service.
2. Product controller development.
3. Product fronted display interface development.

Firstly, we need to create a product controller inside the fronted/web/controllers folder. We need the Iris context and make the product service consistent with the previous interface. Besides, we need a temporary session to run the program. However, the session here will cause high pressure on the server and is hard for the server to manage, which would not achieve our high current software requirements. Hence, we will change and update the session to the cookie in the later chapter.

We need to query our product information from the database with a query function after developing the database.

3.1.4 Quantity Logic Control

We need to do the purchase data control development after the product's detailed information is developed, which is one of the core functions of our software and has three main parts as follows:

1. Get product information
2. Product ID conversion
3. Inquire about the products

Firstly, we need to get the product ID from the URL parameter and assign it to our program. Secondly, we use a cookie to get the user ID. In addition, we need to do product ID conversion to achieve the above needs. After getting the product and user ID, we need to query the product information to check if we still have enough products in stock through product service.

Now, let us check the quantity of the products to see if the quantity satisfies the selling requirements. If the product number is bigger than zero, the user will be able to purchase. Every time a user successfully purchases the product, we deduct the product quantity by one and keep the quantity updated with the database. Similarly, if we create an order, the number of orders increases.

Redis for product number management

Redis is a high-performance key-value database that has the following advantages comparing other databases.

1. Redis has very high performance, and reading and writing speed is better than other databases.
2. Redis supports rich data structures like sets, strings, and lists.
3. Database operations in Redis are all atomic.

We use Redis to cache the product data to achieve high-performance judgment when users grab the product. We implement the operation of adding and removing keys for Redis. We define a global Redis connection variable for Redis and then create the initialization function to perform connection tests on Redis. Next, we create SetCache, DelCache, and GetCache functions for a specific key as follows.

Note that in Redis, the effect of adding and updating is the same, so the SetCache is used uniformly for implementation.

The execution logic diagram for our high concurrency purchase system using Redis is shown in 3.5.

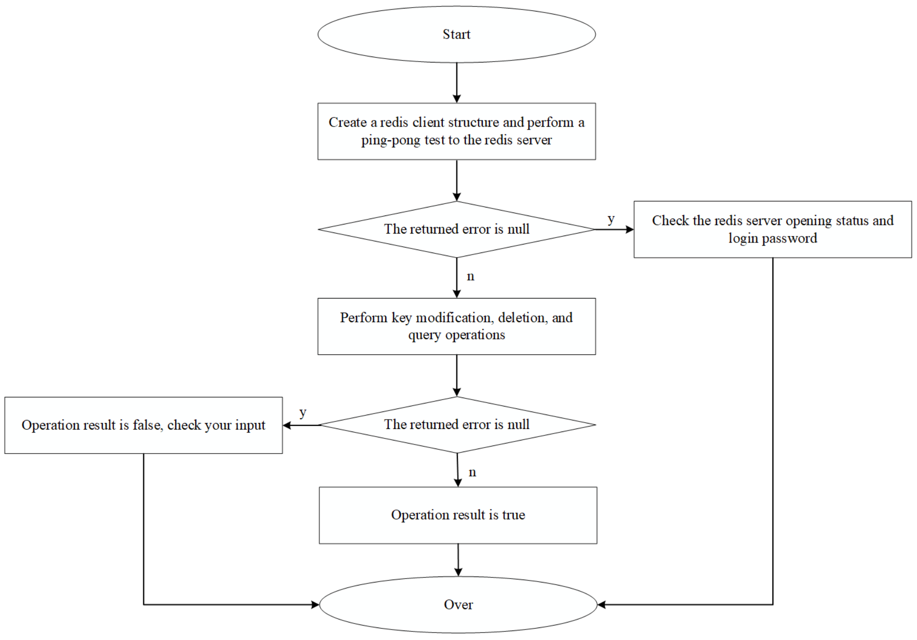

Start
Create a redis client structure and perform a ping-pong test to the redis server
The returned error is null
y
Check the redis server opening status and login password
n
Perform key modification, deletion, and query operations
Operation result is false, check your input
y
The returned error is null
n
Operation result is true
Over

Figure 3.5: The execution logic diagram of redis

3.2 Optimization for High Concurrency Purchase System

3.2.1 Front-end Optimization

During the process of developing the front-end part, we already analyzed the core functions architecture as follows:

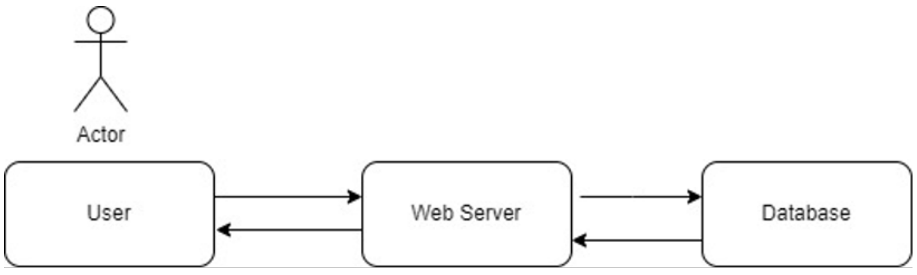


Figure 3.6: frontend-opt

When a user visits our website, the first visit will be to our web server, where all the product information is listed. Inside the web server (the detailed products information page), query functions are closely connected with the databases. After the web server interacts with the database, the database will get the detailed product information and then send data back to the web server to do the HTML page rendering for the users.

Optimize the architecture

However, this simple fundamental architecture cannot meet the needs of our software's high current system. The main problems will be shown as follows:

1. Web server will have high pressure, and the security verification cost will be expensive. The server needs to deal with a large number of users' requests and do action for the security verification. When all the actions need to be done on one server, the high pressure increases the financial cost.

2. Getting real-time data will cause high pressure on the databases. The web server will receive one request when one user does actions on the front-end page. When there are many users, like 10000 users visiting our web page and doing relatives operations, the server will receive 10000 requests and do 10000 queries.

3. It is hard to keep the data consistent in the high current situations. In the high current situation, the oversold problem of the products are generally taken place, and we have to do actions to every pressure node.

We build the following architecture as shown below to solve the above problems.

1. When a user visits our system, he will visit all the static resources where the js, CSS, product images, and the HTML files locate.

2. When a user wants to purchase a product, and after clicking the "Purchase" button, real-time data will be sent to the SLB(Server Load Balancer). The SLB will distribute the requests from the web servers.

3. The distributed security verification is mainly used to do the data interception checking if the request is valid. Meanwhile, the distributed system can do horizontal scaling.

4. There will be a product purchase quantity control after the distributed verification is done to avoid the oversold problems. Meanwhile, this will also improve the performance of the software.

5. When the product purchase quantity control receives the request to purchase a product, it will send this request to the web server.

6. The web server will write the purchase message to the RabbitMQ after receiving the request.

7. RabbitMQ is a messaging broker. When data traffic occurs, MySQL will receive multiple requests and get busy. RabbitMQ will serve as a middleware to store all the requests helping to decrease database pressure.

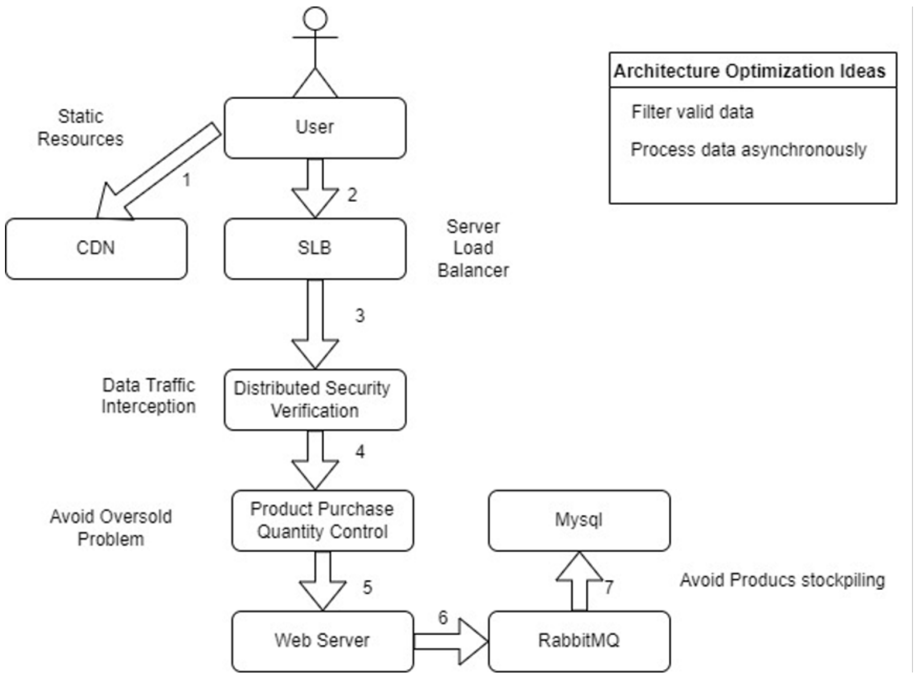

User
Static Resources
1
2
CDN
SLB
Server Load Balancer
3
Data Traffic Interception
Distributed Security Verification
4
Avoid Oversold Problem
Product Purchase Quantity Control
5
Web Server
6
RabbitMQ
7
Mysql
Avoid Producs stockpiling
Architecture Optimization Ideas
Filter valid data
Process data asynchronously

Figure 3.7: system

Static page design and development

We can display our pages according to the requests after a web page is developed in a static method. The static webs' display will be much faster than the dynamic ones. A dynamic website will always change and present different interfaces concerning real-time data. Every time we click a link or visit a dynamic website, the server will check a series of dynamic data and do relative logic judges to check if the service logic is legal. If it is legal and consistent with our services and logic, the pages need to do rendering to respond to these logic and eventually display the relative pages. For dynamic pages, if the query functions and logic judgments are not well developed or the web pages receive too many visits, the server behavior will be out of expectation, even close, shut down automatically, or crash. However, a static web page will reduce such negative impacts and improve speed and efficiency.[23]

Since the dynamic web pages need to query a series of data which will cause heavy pressure even though using cache or Redis will not help a lot, only displaying a static web will comparably cost a few resources and reduce the server working pressure.

In most cases, static sites are way more secure than dynamic sites. As the dynamic sites generally are more exposed to the internet and every request from the server needs to interact with the application and database, an unexpected malicious request could cause uncountable mistakes or errors.

Typically, there are several ways to develop static webs, like using a static site generator, buying a domain, and doing the deployment. Instead, we choose to implement it through coding.

Firstly, we need to create the directories to store the generated HTML file and the static file templates. Then we need a function to generate the HTML static files.

We still need a controller to generate the HTML after the HTML function is created. Similarly, we need to properly handle the files and template path and deal with the error.

We use the ParseFiles function to generate the template path inside the GetGenerateHtml function. Besides, the HTML file generated path should also be inserted. After the path setting, we need to do the template data rendering. The last step is to generate the static files with the necessary parameters assigned.

Javascript security control

We have implemented the fronted web optimization in the above chapters. Now we will do the fronted JS data flow control in this chapter.

Many users will visit our websites simultaneously in a real-time situation for our software. If we do not set a limit to the data flow, the web server will receive a request every time a user visits our website. Similarly, the web server will send a query request to the database. The above series of impacts will cause high pressure for the databases and generate some unexpected resource waste, affecting the stability of our webs and applications. If we set the data flow limitation layer by layer, only valid data will be sent from the fronted web to the server, and accordingly, valid requests will be recorded in the database. Eventually, the database will receive countably affordable requests and keep stable interaction with the server.

The common Javascript security control can be listed as follows:

- JS set the limit that the user can only submit the request within ten seconds. If the user has already sent the request to buy, either successfully or not, he/she can only wait ten seconds for another operation.
- JS set the limitation to users who did not log in. Only our members can operate on our websites because we receive such requests from the user ID, and this limitation will also avoid invalid requests.

We will be able to run the front-end page after setting the static files and resources directories. However, we should set the page parameters with a script in the htmlProduct.html file. Then we need to code the user rush to buy function with related parameters and interact with the button.

Once a user rushes to buy a product and has clicked the button, there should be a timer that will limit the user's operation. The user will be able to do another operation ten seconds later after the last operation whenever the user successfully purchases the product.

```
    function timeSub(){
        inter = setInterval("timeFunc()",1000)
    }
    function timeFunc(){
        count--;
        if (count<=0){
            count = interval
            document.getElementById(BuyButtonID).removeAttribute("
               disabled")
            document.getElementById(BuyButtonID).value="Buy now!";
            clearInterval(inter);
        }else{
            document.getElementById(BuyButtonID).value="waiting"+
               count+"seconds to buy";
        }
    }
```

Code 3.20: The code of js security control

We need to send asynchronous requests after executing the timeSub() function. Firstly, we need to create an ajax engine object that will support receiving requests from IE7+, Firefox, Chrome, Opera, and Safari. Secondly, we can bind the listener event to the engine object.

3.2.2 Back-end Optimization

Consistent hash

We design to support multiple machines in our high concurrent purchase system, and by deploying multiple machines, we can expand the processing performance of the whole system horizontally. All distributed machines will constitute a distributed storage cluster. For distributed storage, data of different objects are stored on different machines, so we use consistent hashing to establish the mapping relationship from data to servers. The consistent hash algorithm ensures that when machines are increased or decreased, the data migration between nodes is limited to between two nodes and does not cause global network problems.

The hash algorithm comes to hash the corresponding key into a space with 2^{32} times the number of buckets, and the number space from 0 to $2^{32} - 1$. Now we can connect these numbers head to tail to form a closed loop.

Since the number of machines is finite and does not completely fill the entire hash ring, we construct virtual nodes for each machine and distribute them over the hash ring to prevent machine overload.

The implementation of consistent hash

First, we need to define a hash structure, which includes a hash ring, a data structure to store sorted keys, the number of virtual nodes, and a read/write lock to control access by multiple machines. Then create the initialization function NewConsistent, which returns a hash structure of pointer type.

```go
type Consistent struct {
  circle map[uint32]string
  sortedHashes hashes
  virtualNodeNums int
  sync.RWMutex
}

func NewConsistent() *Consistent{
  return &Consistent{
    circle: make(map[uint32]string),
    virtualNodeNums: 20,
  }
}
```

Code 3.21: Definition of consistent hash

Secondly, we generate the corresponding key for each virtual node and define a good hash function, taking 127.0.0.1 as an example, then its key for each virtual is 127.0.0.1-1, the symbol of following is the number of this virtual node. Then we define the hashKey function to hash the keys so that they are mapped to the hash ring.

```go
func (c *Consistent) generateVirtualNodes(s string,index int)
   string{
  return s + "-" + strconv.Itoa(index)
}

func (c *Consistent) hashKey(key string) uint32{
  if len(key) < 64{
    var extendKey [64]byte
    copy(extendKey[:],key)
    return crc32.ChecksumIEEE(extendKey[:len(key)])
  }
  return crc32.ChecksumIEEE([]byte(key))
}
```

Code 3.22: hash function of consistent hash

Third, set up the add and remove node function for the hash ring, whose function is to adapt to the needs of adding and removing servers in distributed scenarios. Adding and removing a certain server on the ring is done by calling the Add and Remove functions.

```go
func (c *Consistent) add(s string){
  for i := 0; i < c.virtualNodeNums; i++{
    c.circle[c.hashKey(c.generateKey(s,i))] = s
  }
  c.updateSortedHashes()
}
func (c *Consistent) remove(s string){
  for i := 0;i<c.virtualNodeNums;i++{
    delete(c.circle,c.hashKey(c.generateKey(s,i)))
  }
  c.updateSortedHashes()
}
```

Code 3.23: Adding and removing for consistent hash

Fourth, the update function needs to be implemented, which is used to sort the existing keys on the hash ring. The request will generate a hash value after hashing and find the server node it should access through this sorted hash key.

```
func (c *Consistent) updateSortedHashes(){
  var hashes hashes
  for k := range c.circle{
    hashes = append(hashes,k)
  }
  sort.Sort(hashes)
  c.sortedHashes = hashes
}
```

Code 3.24: update for consistent hash

Fifth, we implement the search function, which finds the nearest server node clockwise from sortedHashes after a certain hash key.

```
func (c *Consistent)search(key uint32) int{
  f := func(x int) bool{
    return c.sortedHashes[x] > key
  }
  i := sort.Search(len(c.sortedHashes),f)
  if i >= len(c.sortedHashes){
    i = 0
  }
  return i
}
```

Code 3.25: search for consistent hash

Finally, we implement a get function that wraps the search function and returns information about the nearest server for a key.

```go
func (c *Consistent) Get(name string) (string,error){
  c.RLock()
  defer c.RUnlock()
  if len(c.circle) == 0{
    return "",noDataErrorMsg
  }
  key := c.hashKey(name)
  index := c.search(key)
  return c.circle[c.sortedHashes[index]],nil
}
```

Code 3.26: get for consistent hash

The use of RWlock

Multiple threads may simultaneously use a resource in a highly concurrent environment. Such resources are called shared resources, and multi-threaded access to shared resources may lead to inconsistent data. Therefore, locks are introduced to protect the shared resource so that only one thread can access the shared resource simultaneously, thus ensuring data consistency.

The implementation of RWlock

Locks are implemented in the sync package in the Go programming language. We first need to define a lock of type sync to lock a shared resource. Mutex then calls the Lock and Unlock methods to complete access and unlock the shared resource, using defer to unlock the data at the end of the function automatically. Other threads cannot access the shared resource until it is unlocked to ensure data consistency.

```go
import "sync"

var mutex sync.Mutex
mutex.Lock()
defer mutex.Unlock()
```

Code 3.27: get for consistent hash

The use of cookie

We use cookies as a user identity token to authenticate the client and prevent unauthorized users from accessing the system. Cookie has the following advantages:

1. The cookie is stored on the client-side, and the client places the cookie in the request for each request to mark the user. Compared to sessions, cookies are easier to implement.

2. The cookie can reduce the storage burden on the server because it is stored on the client-side, and its storage size is limited.

The implementation of cookie

Cookies can be implemented by context package in the Go. We set an http.cookie to the specified key-value pair form using the Setcookie function in the context package.

We then modify the user login function to authenticate the user. First, we get the user's account number and password, then get the results by comparing them. If the user passes the authentication, a cookie named auth is added to identify him as a legitimate user. The user is then redirected to the product details page.

```go
func (u *UserController) PostLogin() mvc.Response{
  userName := u.Ctx.FormValue("userName")
  passWord := u.Ctx.FormValue("passWord")
  user, ok := u.Service.IsPwdEqual(userName,passWord)
  if !ok{
    return mvc.Response{
      Path: "/user/login",
    }
  }
  tool.SetGlobalCookie(u.Ctx,"id",strconv.FormatInt(user.ID,10))
  idByte := []byte(strconv.FormatInt(user.ID,10))
  idString,_ := tool.StrEncode(idByte)
  tool.SetGlobalCookie(u.Ctx,"auth",idString)
  return mvc.Response{
    Path: "/product",
  }
}
```

Code 3.28: The implementation of cookie

3.3 Testing

Our application will be safe and credible under full tests. In the Go language, the automated test can be very useful for finding bugs. Our test will keep validating the updated code after adding new code and deleting old code. We will use the go test tool to check our application in our application. In addition, we will perform a simple manual security test. All in all, four-layer tests will be utilized by us to demonstrate the robustness and security of the high concurrency purchase system as follows:

1. Unit tests - it will test some part of the application independently.
2. Security tests - it will check the effectiveness of our security measures.
3. Functional tests - it will test our application works from the user's view.
4. Availability tests - it will test our high concurrency purchase application to see if it works under high traffic conditions.

3.3.1 Testing Plans

In the security test, we first checked the system's AES encryption algorithm to check whether an attacker could forge the AES encryption result to pass the system's verification. Secondly, we verified the cookie mechanism used to see whether a non-logged-in user could directly grab the product.

In Functional tests, we simulate access to URLs with the curl tool to check whether the corresponding results are obtained. Because access to a single URL triggers the application's judgment, functional tests can check whether the whole system is running correctly.

In the availability test, we conduct a stress test on the externally exposed robocall port of our high concurrent purchase system via wrk to check the usability of our system in dealing with high concurrency and prove that our system can withstand the pressure of high concurrent and heavy traffic after using RabbitMQ and consistent hashing.

3.3.2 Writing Tests

Unit tests

The unit test mainly checks the functions of orders, users, and products in the system. Here, since the three functions are very similar, we choose orders as an example to test. We write the following four functions to test adding, deleting, and checking orders.

```go
func TestOrderService_GetOrderByID(t *testing.T) {
  db,_ := common.NewMysqlConn()
  order := repositories.NewOrderManagerRepository("order",db)
  orderService := NewOrderService(order)
  if orderService == nil{
    t.Errorf("Get null!")
  }
  if _,err := orderService.GetOrderByID(int64(1)); err !=nil{
    t.Errorf("Get null!")
  }
}
```

Code 3.29: Test for GetOrderByID

```go
func TestOrderService_DeleteOrderByID(t *testing.T) {
  db,_ := common.NewMysqlConn()
  order := repositories.NewOrderManagerRepository("order",db)
  orderService := NewOrderService(order)
  if orderService == nil{
    t.Errorf("Get null!")
  }
  if err := orderService.DeleteOrderByID(int64(1)); err == false{
    t.Errorf("Delete order false!")
  }
}
```

Code 3.30: Test for DeleteOrderByID

```go
func TestOrderService_UpdateOrder(t *testing.T) {
  db,_ := common.NewMysqlConn()
  order := repositories.NewOrderManagerRepository("order",db)
  orderService := NewOrderService(order)
  if orderService == nil{
    t.Errorf("Get null!")
  }
  testOrder := &data.Order{int64(1),int64(1),int64(1),0}
  if err := orderService.UpdateOrder(testOrder); err != nil{
    t.Errorf("Update order false!")
  }
}
```

Code 3.31: Test for UpdateOrder

```go
func TestOrderService_InsertOrder(t *testing.T) {
  db,_ := common.NewMysqlConn()
  order := repositories.NewOrderManagerRepository("order",db)
  orderService := NewOrderService(order)
  if orderService == nil{
    t.Errorf("Get null!")
  }
  testOrder := &data.Order{int64(2),int64(1),int64(1),0}
  if _,err := orderService.InsertOrder(testOrder); err != nil{
    t.Errorf("Insert order false!")
  }
}
```

Code 3.32: Test for InsertOrder

In addition to the above unit test, we also checked the standalone get IP function, and the code is shown below.

```go
func TestGetLocalIp(t *testing.T) {
  if ans,_ := GetLocalIp(); ans != "127.0.0.1" {
    t.Errorf("Get %s wrong", ans)
  }
  if ans,_ := GetLocalIp(); ans == "127.0.0.2" {
    t.Errorf("Get %s wrong", ans)
  }
}
```

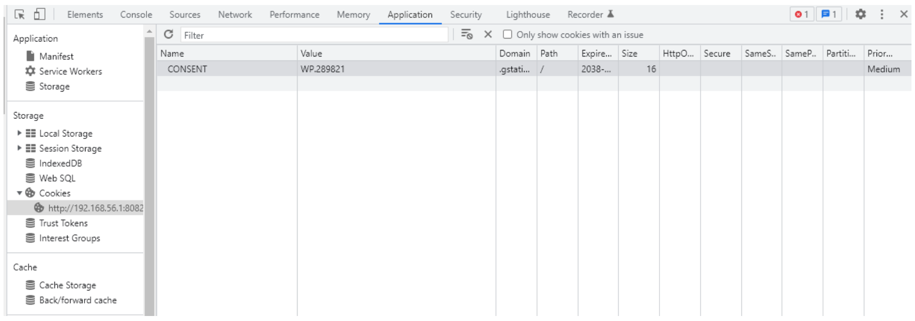
Code 3.33: Test for IP address

Security tests

We will test two aspects of our high concurrency spike system. First, we use an empty cookie to simulate an unauthorized user with direct access to access.

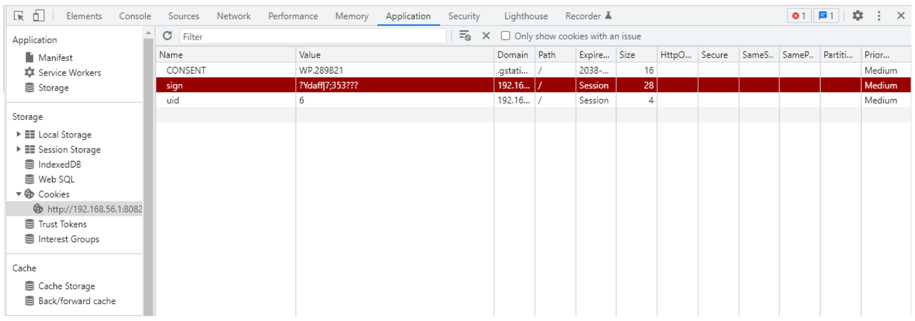
In our system, if it jumps directly to the login page, it means it passes the test. Moreover, if the access is successful, it means the test fails. Second, we use a modified cookie to simulate an attacker's access to the system. With the help of the AES encryption algorithm, our system checks the value of the cookie modified by the attacker to identify that the current user is trying to tamper with the cookie to bypass authentication.

As shown in the Figure 3.8 and Figure 3.9, we test this by constructing a window directly in the browser that does not contain cookie information and a window that contains modified cookie information and accessing them separately.



Name	Value	Domain	Path	Expire...	Size	HttpO...	Secure	SameS..	SameP..	Partiti...	Prior...
CONSENT	WP.289821	.gstati...	/	2038-...	16						Medium

Figure 3.8: No auth cookie



Name	Value	Domain	Path	Expire...	Size	HttpO...	Secure	SameS..	SameP..	Partiti...	Prior...
CONSENT	WP.289821	.gstati...	/	2038-...	16						Medium
sign	?Ydaff|7;353???	192.16...	/	Session	28						Medium
uid	6	192.16...	/	Session	4						Medium

Figure 3.9: Modified auth cookie

Functional tests

We test against the getOne interface and the records generated by RabbitMQ in the functional tests. The getOne interface request is accessed directly using the curl tool, the former via the following command.

curl http://127.0.0.1:8084/getOne

Then we will construct the request via curl and then see if RabbitMQ accepts the request.

Availability tests

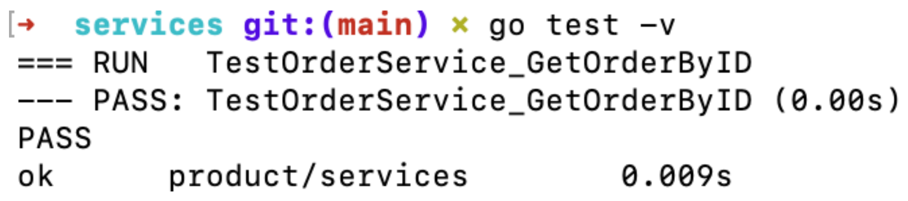
We have given our test cases with the wrk tool's help as follows: the parameter t represents the number of threads, the parameter c represents the number of connections established, -d represents the duration of the stress test, and –latency represents the output latency. We can change these parameters to put more pressure on the system to check the system's operation.

wrk -t40 -c300 -d10s –latency "http://127.0.0.1:8084/getOne"

3.3.3 Testing Results

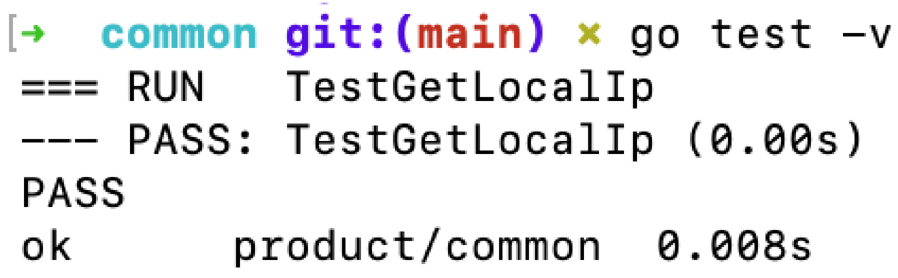
Unit tests

Run the go test -v command so that we can test the file in the corresponding test directory. Here are the test results of the function of getting the order according to the ID and the test results of getting the local IP. Since the other test results are similar to the following Figure, so we will not repeat them.



```
[➜  services git:(main) ✗ go test -v
=== RUN   TestOrderService_GetOrderByID
--- PASS: TestOrderService_GetOrderByID (0.00s)
PASS
ok      product/services        0.009s
```

Figure 3.10: Test for order



```
[➜  common git:(main) ✗ go test -v
=== RUN   TestGetLocalIp
--- PASS: TestGetLocalIp (0.00s)
PASS
ok      product/common  0.008s
```

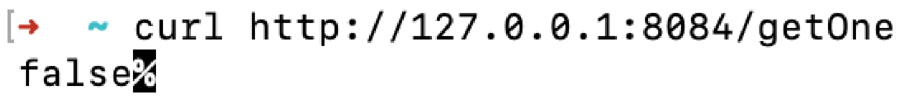
Figure 3.11: Test for getLocalIP

Security tests

The result of the security test is that the attacker will automatically jump to the landing page when he visits the page with an empty cookie and a modified cookie, which is because our cookie validation works.

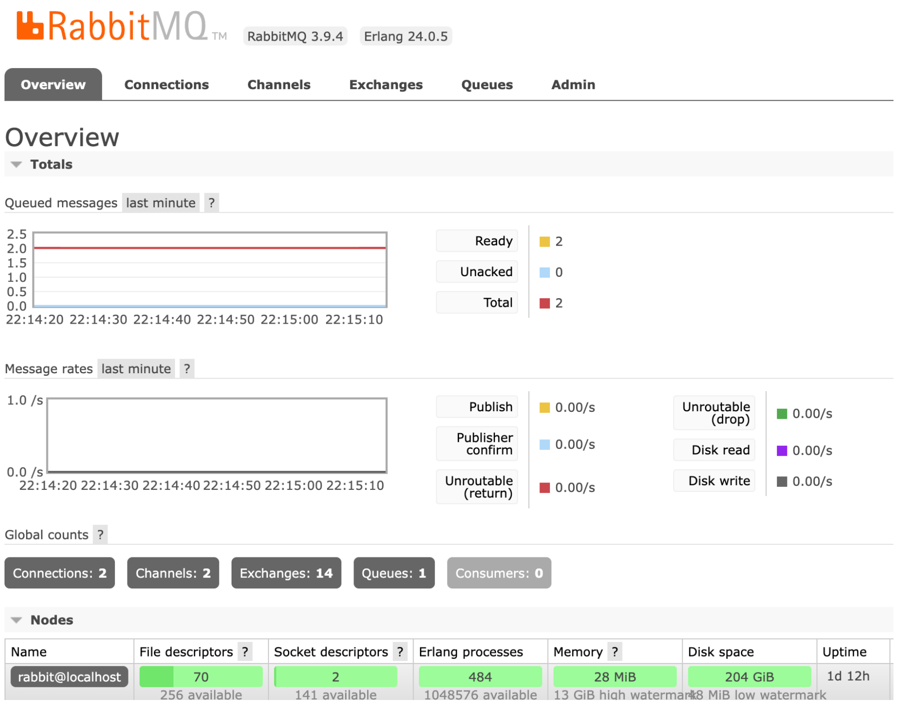
Functional tests

We performed functional tests on getOne, tested RabbitMQ, and found that getOne would return a failed snatch result with no cookie-carrying access. After our two click for buying through the browser, RabbitMQ received two requests to add order and waited to consume it.



```
[➜  ~ curl http://127.0.0.1:8084/getOne
false%
```

Figure 3.12: The result of curl



Overview

Totals

Queued messages last minute ?

2.5 2.0 1.5 1.0 0.5 0.0
22:14:20 22:14:30 22:14:40 22:14:50 22:15:00 22:15:10

Ready	🟨 2
Unacked	🟦 0
Total	🟥 2

Message rates last minute ?

1.0 /s
0.0 /s
22:14:20 22:14:30 22:14:40 22:14:50 22:15:00 22:15:10

Publish	🟨 0.00/s	Unroutable (drop)	🟩 0.00/s
Publisher confirm	🟦 0.00/s	Disk read	🟪 0.00/s
Unroutable (return)	🟥 0.00/s	Disk write	⬛ 0.00/s

Global counts ?

Connections: 2 Channels: 2 Exchanges: 14 Queues: 1 Consumers: 0

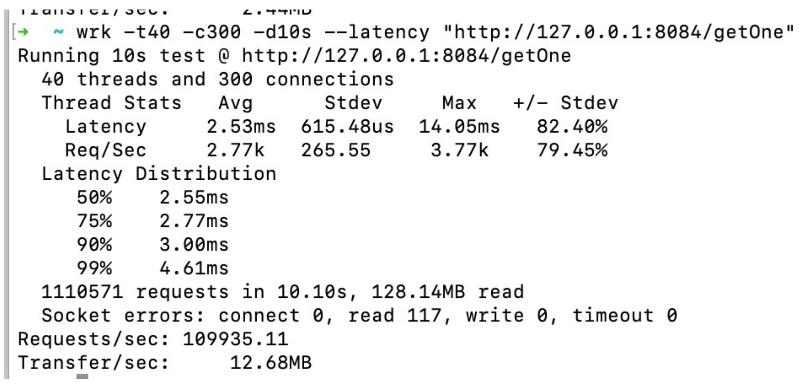
Nodes

Name	File descriptors ?	Socket descriptors ?	Erlang processes	Memory ?	Disk space	Uptime
rabbit@localhost	70 256 available	2 141 available	484 1048576 available	28 MiB 13 GiB high watermark	204 GiB 48 MiB low watermark	1d 12h

Figure 3.13: The functional result of RabbitMQ

Availability tests

We conducted a concurrency test on our getone interface as the Figure 3.14 shows, and the wrk software used 40 threads to snatch the product and 300 connections in 10 seconds. We can find that the average latency of the getone interface is about 2.53ms, 99% of the latency is distributed within 5ms. The volume of requests processed per second can reach 110,000, proving that our snatch interface can be used in the RabbitMQ. Consistent hashing has improved the system's ability to withstand large traffic volumes and cope with high concurrency scenarios.



```
➜  ~ wrk -t40 -c300 -d10s --latency "http://127.0.0.1:8084/getOne"
Running 10s test @ http://127.0.0.1:8084/getOne
  40 threads and 300 connections
  Thread Stats   Avg      Stdev     Max   +/- Stdev
    Latency     2.53ms  615.48us  14.05ms   82.40%
    Req/Sec     2.77k   265.55     3.77k    79.45%
  Latency Distribution
     50%    2.55ms
     75%    2.77ms
     90%    3.00ms
     99%    4.61ms
  1110571 requests in 10.10s, 128.14MB read
  Socket errors: connect 0, read 117, write 0, timeout 0
Requests/sec: 109935.11
Transfer/sec:     12.68MB
```

Figure 3.14: Availability test for purchase interface

It is worth noting that we have only built the service locally here, and if it is deployed on a cloud server, it will still be able to handle a large number of requests normally, although its latency will increase.

3.3.4 Testing Conclusion

We can conclude that after the above four types of tests, our high concurrent purchase system has better usability, security and meets the purchase needs of users under high concurrency conditions.

We have given the interface tests for adding, deleting, modifying, and querying orders as an example to check that all system parts are operating properly. We can conclude that our system has no problems in terms of functionality.

We tested the system's security from a hacker's perspective by simulating two different types of attacks with an empty cookie and a tampered cookie and found that our cookie mechanism can effectively differentiate between illegal users, and the AES encryption mechanism makes it difficult for attackers to crack encrypted cookies.

Our high concurrent purchase system needs to withstand high traffic access. Therefore, in the usability testing, we simulated several connections and tested them for a period of time with the help of an automated wrk testing tool, and found that our system can handle up to 100,000 accesses per second, with the potential to withstand more if we use a distributed system.

Eventually, we completed the software work and now it comes to an end of our project. The next conclusion part will summarize the work and offer future work ideas.

Conclusion

We have already finished developing the high concurrency purchase system for the e-commerce website's flash sale. To sum it up, we built the backend products and orders management system for the managers and the fronted shopping purchase system for the customers. Since we used such proper technical tools and nice software architecture, our system is stable, secure, and efficient for the users. The customers will get a joyful and fair experience during the shopping. The managers will benefit greatly by receiving many orders, making big profits, and not worrying about the oversold problems from such a stable and convenient management system.

Since we use the Go programming language and Iris framework with MVC pattern, the software architecture is clear and easy to understand for the developers to do the future development. Besides, with the distributed verification and consistent hash algorithm, our software will be able to scale out, which will also benefit future developers a lot.

The future work of the software can be concluded as follows: 1. Add verification code in the front-end web to avoid malicious spam registration or login. 2. We can add more patterns into the product models like stock keeping unit[24] (SKU) or attributes of the products to provide the users with more choices and generate a friendly shopping experience. 3. We can do front-end performance optimization from the page level [25]. For example, reduce external HTTP requests, minify CSS, enable prefetching, Increase speed with a CDN and caching, use a minimalistic framework, Etc. 4. We can optimize the consistent hash algorithm to solve the problem of precise mapping between requests and cache servers and eliminate cache server issues.

Bibliography

[1] Avinash Sharma. *A Mini-Guide on Go Programming Language*. https://appinventiv.com/blog/mini-guide-to-go-programming-language/. 2022.05.10.

[2] *Introduction to TCP/IP*. https://w3schools.sinsixx.com/tcpip/tcpip_intro.asp.htm. 2022.05.10.

[3] Golang. *Concurrency*. https://www.golang-book.com/books/intro/10. 2022.05.10.

[4] *go philosophy*. https://commandcenter.blogspot.com/2012/06/less-is-exponentially-more.html. 2022.05.10.

[5] *Iris Web Framework*. https://www.iris-go.com/. 2022.05.10.

[6] Jon Calhoun. *Using MVC to Structure Go Web Applications*. https://www.calhoun.io/using-mvc-to-structure-go-web-applications. 2022.05.10.

[7] *MySQL 8.0 Reference Manual*. https://dev.mysql.com/doc/refman/8.0/en/introduction.html. 2022.05.10.

[8] *Redis Documentation*. https://redis.io/docs/. 2022.05.10.

[9] *RabbitMQ*. https://en.wikipedia.org/wiki/RabbitMQ. 2022.05.10.

[10] *RabbitMQ Message Queue*. https://www.rabbitmq.com/tutorials/tutorial-one-elixir.html. 2022.05.10.

[11] Jon Calhoun. *Using MVC to Structure Go Web Applications*. https://www.calhoun.io/using-mvc-to-structure-go-web-applications. 2022.05.10.

[12] *Model–view–controller*. https://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93controller. 2022.05.10.

[13] Jon Calhoun. *Using MVC to Structure Go Web Applications*. https://www.calhoun.io/using-mvc-to-structure-go-web-applications. 2022.05.10.

[14] *Model–view–controller*. `https://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93controller`. 2022.05.10.

[15] Jon Calhoun. *Using MVC to Structure Go Web Applications*. `https://www.calhoun.io/using-mvc-to-structure-go-web-applications`. 2022.05.10.

[16] *Model–view–controller*. `https://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93controller`. 2022.05.10.

[17] MDN Web Docs. *HTML: HyperText Markup Language*. `https://developer.mozilla.org/en-US/docs/Web/HTML`. 2022.05.10.

[18] MDN Web Docs. *CSS: Cascading Style Sheets*. `https://developer.mozilla.org/en-US/docs/Web/CSS`. 2022.05.10.

[19] MDN Web Docs. *CSS Introduction*. `https://www.w3schools.com/css/css_intro.asp`. 2022.05.10.

[20] lakshita. *Introduction to JavaScript*. `https://www.geeksforgeeks.org/introduction-to-javascript/`. 2022.05.10.

[21] MDN Web Docs. *JavaScript*. `https://developer.mozilla.org/en-US/docs/Web/JavaScript`. 2022.05.10.

[22] Tomislav Bacinger. *What is Bootstrap? A Short Bootstrap Tutorial on the What, Why, and How*. `https://www.toptal.com/front-end/what-is-bootstrap-a-short-tutorial-on-the-what-why-and-how`. 2022.05.10.

[23] srandby. *Dynamic and Static Website Security*. `https://srandby.org/digital-writing/index.html`. 2022.05.10.

[24] *Stock Keeping Unit (SKU)*. `https://www.shopify.com/encyclopedia/stock-keeping-unit-sku`. 2022.05.10.

[25] Arsenault. *frontend optimization*. `https://www.keycdn.com/blog/frontend-optimization`. 2022.05.10.

List of Figures

List of Codes