

# Advance Data Management - Programming Project

Development of an application supported by a NoSQL Database

Giacomo Di Fabrizio

Department of Electrical and Computer Engineering  
University of Thessaly

April, 2024



## Selection of the topic

The objective of this project is to develop a MERN (MongoDB, Express.js, React.js, Node.js) application tailored for an e-commerce website.

## Description of the topic

The aim of this project is to design and implement a robust e-commerce website using the MERN (MongoDB, Express.js, React.js, Node.js) stack.

The website will serve as a platform for users to browse, search, and purchase products online.

It will provide an intuitive and seamless shopping experience for customers.

## Specification of the topic

- User Authentication
- Product Catalog Management
- Shopping Cart Management
- Order Tracking
- Responsive Design
- Performance Optimization

## Selection of NoSQL DB system suitable for the above topic.

For this project, MongoDB has been selected as the NoSQL database system.

MongoDB's flexibility, scalability, and ability to handle unstructured data make it well-suited for the requirements of an e-commerce website. Its document-oriented data model allows for easy storage and retrieval of product information, user data, and transaction records.

Additionally, MongoDB's support for sharding and replication ensures high availability and fault tolerance, crucial aspects for an e-commerce platform handling large volumes of data and transactions.

[Sign up for MongoDB Atlas](#)

I visited the [MongoDB Atlas website](#) and signed up for an account. MongoDB Atlas offers a free tier, which is suitable for getting started with small-scale projects.



# Create a Project

After signing in to MongoDB Atlas, I created a new project.

The screenshot shows the MongoDB Atlas web interface for creating a new project. The URL in the browser is <https://cloud.mongodb.com/v2#/org/63bdad0d58d1141f6d0a886/projects/create>. The page title is "Create a Project". On the left, there's a sidebar with "ORGANIZATION" selected, showing links for Projects, Alerts, Activity Feed, Settings, Integrations, Access Manager, Billing, Support, and Live Migration. The main content area has a header "GIACOMO'S ORG - 2023-01-10 > PROJECTS". It contains two tabs: "Name Your Project" (selected) and "Add Members". Under "Name Your Project", there's a text input field with "EcommerceAdvanceDataManagement" typed in. Below it, there's a section for "Add Tags (Optional)" with a table for adding tags. The table has columns "Key" and "Value", both with dropdown menus. A button "+ Add tag" is at the bottom. At the bottom right of the form are "Cancel" and "Next" buttons. At the very bottom of the page, there's a footer with "System Status: All Good" and "Last Log in: 078.59.47.120", along with links for "Status", "Terms", "Privacy", "Atlas Blog", and "Contact Sales".

# Create a Project

The screenshot shows the MongoDB Cloud interface for creating a new project. The URL in the browser is <https://cloud.mongodb.com/v2/org/63bdad0d58d1141f6d0af886/projects/create>. The interface includes a left sidebar with 'ORGANIZATION' and 'Projects' selected, and various other tabs like Alerts, Activity Feed, Settings, Integrations, Access Manager, Billing, Support, and Live Migration. The main content area is titled 'Create a Project' and has two tabs: 'Name Your Project' (selected) and 'Add Members'. Below these tabs is a section for 'Add Members and Set Permissions'. A text input field says 'Invite new or existing users via email address...'. A dropdown menu lists a user: 'giacomo@fabrizio@gmail.com (you)'. To the right of this is a 'Project Owner' dropdown set to 'Project Owner'. At the bottom are 'Back', 'Cancel', and 'Create Project' buttons. On the right side, there's a sidebar titled 'Project Member Permissions' listing several roles with their descriptions:

- Project Owner**: Has full administration access.
- Project Cluster Manager**: Can update clusters.
- Project Data Access Admin**: Can access and modify a cluster's data and indexes, and kill operations.
- Project Data Access Read/Write**: Can access a cluster's data and indexes, and modify data.
- Project Data Access Read Only**: Can access a cluster's data and indexes.
- Project Search Index Editor**: Can view and manage a cluster's search indexes.
- Project Read Only**: May only modify personal preferences.
- Project Stream Processing Owner**: Has full administration access.

# Create a Cluster

After creating a new project, I created a new cluster.

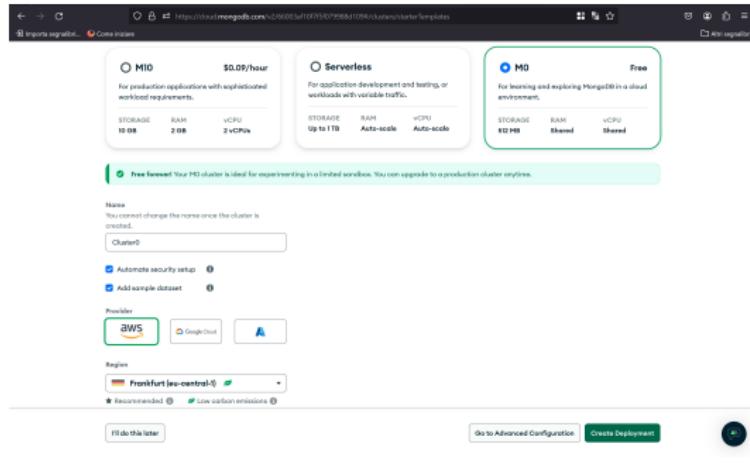
A cluster in MongoDB Atlas consists of one or more MongoDB instances running on the cloud.

The screenshot shows the MongoDB Atlas interface. On the left, there's a sidebar with sections for Deployment (Database, Data Lake), Services (Device Sync, Triggers, Data API, Data Federation, Atlas Search, Stream Processing, Migration), Security (Backup, Database Access, Network Access, Advanced), and a 'New On Atlas' section. The main area is titled 'Overview' and shows a large green button labeled '+ Create'. To the right, there's a 'Toolbar' with a 'Featured Resources' section containing links to 'Get Started with Atlas', 'Reference MongoDB Documentation', 'Develop Applications with the Developer Center', and 'Ask the MongoDB Community'. Below that is a 'New On Atlas' section with a link to 'Learn about the latest feature enhancements on Atlas'. At the bottom, there's a footer with links to 'System Status: All Good', 'Status', 'Terms', 'Privacy', 'Atlas Blog', 'Contact Sales', and 'Data'.

## Configure Cluster Settings

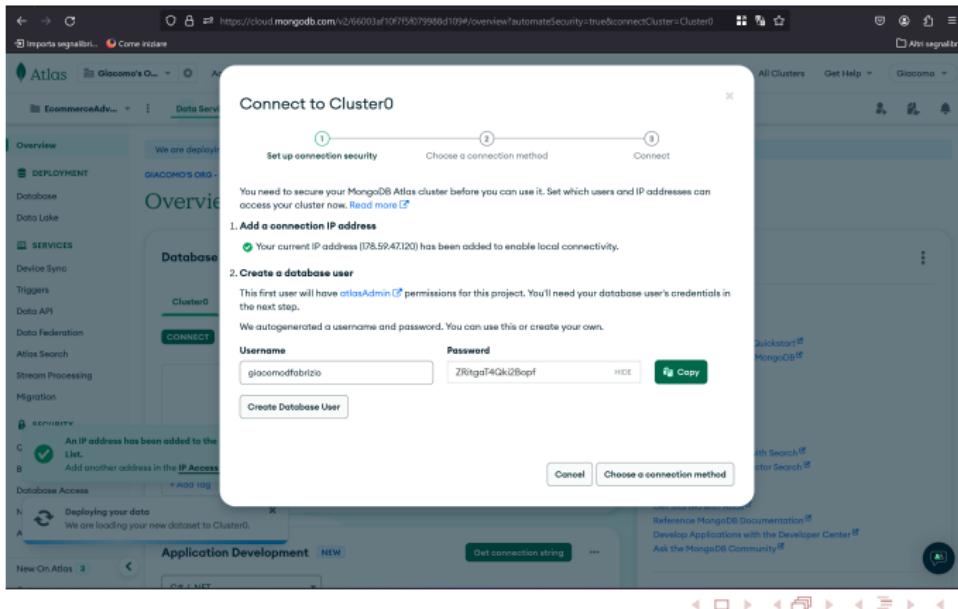
During the cluster creation process, I configured settings such as the cloud service provider (e.g., AWS, Azure, GCP), region, and cluster tier based on my requirements and preferences.

MongoDB Atlas provides options for high availability, automatic scaling, and backups to ensure data reliability and availability.



# Set Up Security

I configured security settings for the cluster, including setting up IP whitelisting, creating database users with appropriate privileges, and enabling encryption at rest and in transit to protect data security.



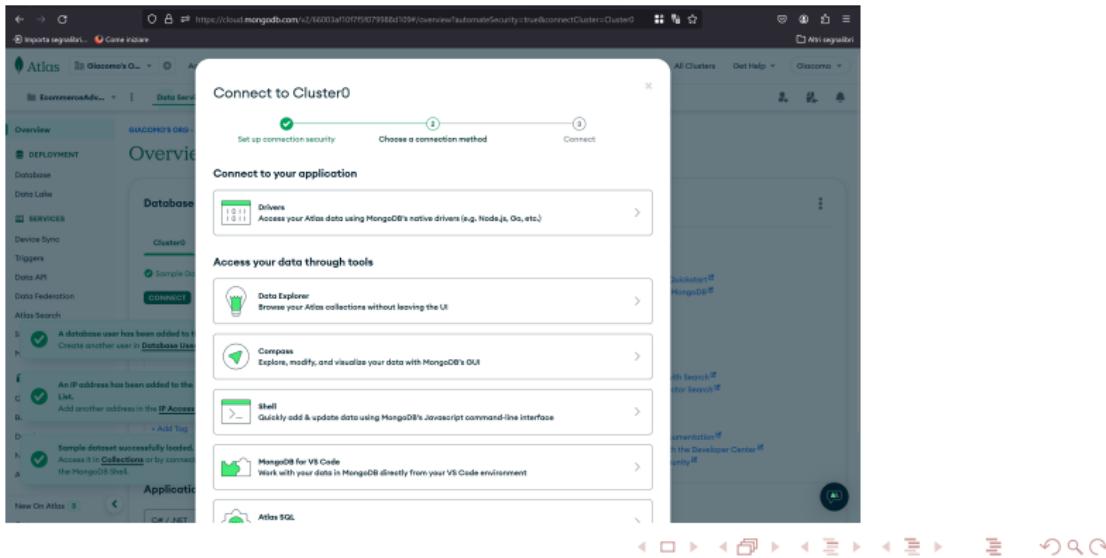
# Set Up Security

The screenshot shows a browser window for MongoDB Atlas. The main interface on the left has tabs for Overview, Deployment, Services, and Database. The Database tab is active, showing a Cluster0 section with a 'Sample Dataset' button. A modal dialog titled 'Connect to Cluster0' is open in the center. The dialog has three steps: 1. Set up connection security (completed with a green checkmark), 2. Choose a connection method (button), 3. Connect (button). Step 1 contains instructions to secure the cluster and add a connection IP address. Step 2 contains instructions to create a database user. Step 3 is a 'Choose a connection method' button. Below the modal, a message says 'An IP address has been added to the IP Access List.' and 'Sample dataset successfully loaded.' At the bottom of the page, there are links for Application Development, Get connection string, and a navigation bar.

## Connect to the Cluster

MongoDB Atlas provides a connection string that I used to connect to the cluster from my application code.

This connection string includes authentication credentials and other necessary information to establish a connection securely.



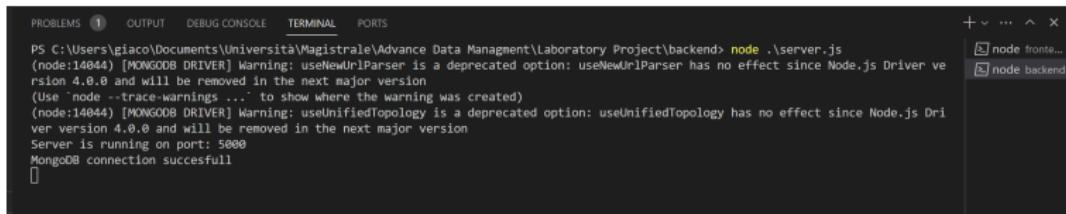
# Connect to the Cluster

```
cartRoute.js Orderpage.js ChartCard.js Chartpage.js 1 NavBar.js db.js X
backend > JS db.js > ...
1  require('dotenv').config()
2  const mongoose = require('mongoose')
3  mongoose.set('strictQuery', false)
4  const uri = process.env.MONGO_CONNECTION
5  mongoose.connect(uri, { useUnifiedTopology: true, useNewUrlParser: true })
6  var connection = mongoose.connection
7  connection.on('error', () => {
8    console.log('MongoDB connection failed')
9  })
10 connection.on('connected', () => {
11   console.log('MongoDB connection successful')
12 })
13 module.exports = mongoose
14
```

## Verify Connection

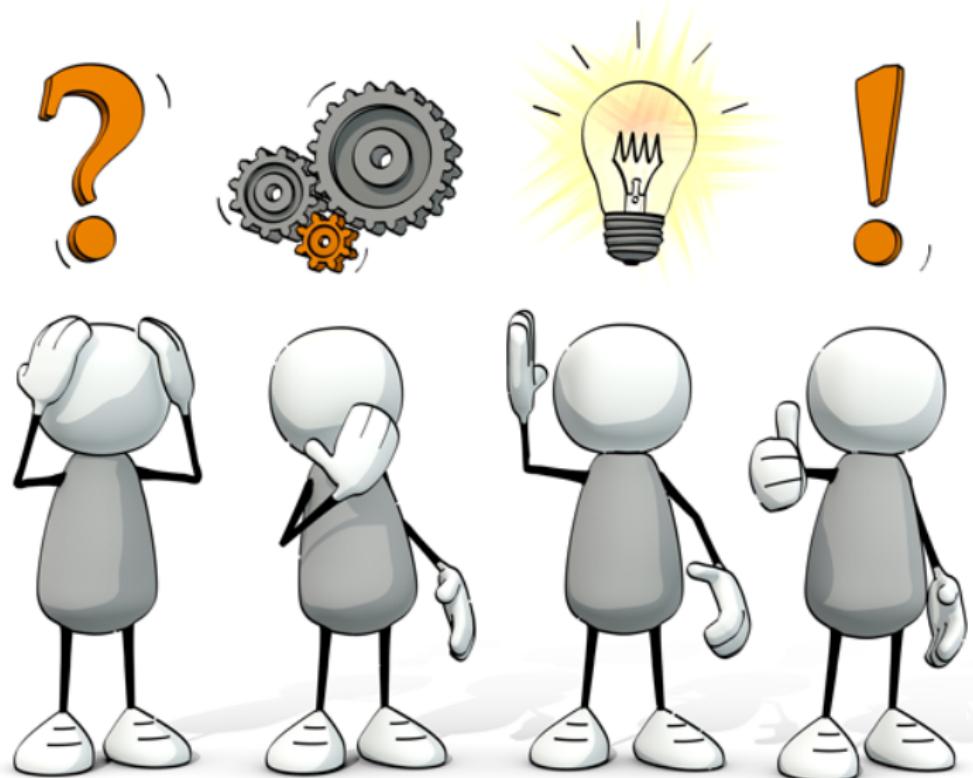
After configuring the connection, I tested the connection from my application to MongoDB Atlas to ensure that it was working correctly.

This involved connecting to the cluster using the provided connection string and performing basic database operations to verify functionality.



```
PROBLEMS 1 OUTPUT DEBUG CONSOLE TERMINAL PORTS + v ... ^ x
PS C:\Users\giaco\Documents\Università\Magistrale\Advance Data Management\Laboratory Project\backend> node ./server.js
(node:14044) [MONGODB DRIVER] Warning: useNewUrlParser is a deprecated option: useNewUrlParser has no effect since Node.js version 4.0.0 and will be removed in the next major version
(Use `node --trace-warnings ...` to show where the warning was created)
(node:14044) [MONGODB DRIVER] Warning: useUnifiedTopology is a deprecated option: useUnifiedTopology has no effect since Node.js Driver version 4.0.0 and will be removed in the next major version
Server is running on port: 5000
MongoDB connection successful
```

## Identification of useful queries for the application.



## User Authentication and Authorization

- Verify user credentials during login: Find a user by email and compare hashed passwords.
- Register a new user: Check if the user already exists and then save the user details with a hashed password.
- Verify user role: Decode the JWT token and extract the isAdmin field to determine user role.

## User Profile

- Retrieve user details: Decode the JWT token to extract user information for profile display.

## Product Management

- Get specific product details: Find a product by its unique identifier.
  - Filter products by category: Retrieve products belonging to a specific category.
  - Get all products: Retrieve all products available in the catalog.

# Order Management

- Add a new order: Create a new order by retrieving items from the user's cart, updating product quantities, and calculating the total cost.
- Get user's order history: Find orders associated with a specific user and include product details for each order.

## Cart Management

- Retrieve user's cart: Find the user's cart and retrieve details of products added to it.
  - Add object to cart: Update the user's cart by adding a new object or increasing the quantity of an existing object.
  - Delete object from cart: Remove an object from the user's cart based on the product ID.
  - Delete user's entire cart: Delete all objects from the user's cart.

# Catalog Management

- Add object to catalog: Add a new product to the catalog or update the quantity of an existing product.
- Get quantity available for a product: Retrieve the quantity available for a specific product in the catalog.

Creating data using data generator or finding ready-made data.

For the creation of data in the e-commerce application, I utilized original product data obtained from an [external source](#).

This data consists of various products with details such as title, price, description, category, image, and ratings.

To prepare this data for insertion into MongoDB, I filtered and transformed it into a format suitable for database insertion.

The filtered data includes essential product details such as name, description, price, category, and image

## Designing the DB model and implementing it in the NoSQL DB system.

For designing the database model and implementing it in the NoSQL DB system, I structured the data using Mongoose schemas in MongoDB.

# Cart Model

```
JS Cart.js X JS Catalog.js JS Order.js JS Products.js
backend > model > JS Cart.js > ...
1 const mongoose = require('mongoose')
2 const Schema = mongoose.Schema
3
4 const cartSchema = new Schema({
5   userid: {
6     type: Schema.Types.ObjectId,
7     ref: 'User',
8   },
9   items: [
10   {
11     productid: {
12       type: Schema.Types.ObjectId,
13       ref: 'Product',
14     },
15     quantity: {
16       type: Number,
17       required: true,
18     },
19   },
20 }, {
21 })
22
23 module.exports = mongoose.model('Cart', cartSchema)
24 |
```

# Catalog Model

```
Cart.js          JS Catalog.js X   JS Order.js      JS Products.js      JS
backend > model > JS Catalog.js > ...
1  const mongoose = require('mongoose')
2  const Schema = mongoose.Schema
3
4  const catalogSchema = new Schema({
5    product: {
6      type: Schema.Types.ObjectId,
7      ref: 'Product',
8    },
9    quantity: {
10      type: Number,
11      required: true,
12    },
13  })
14
15 module.exports = mongoose.model('Catalog', catalogSchema)
16
```

# Order Model

```
JS Cart.js      JS Catalog.js    JS Order.js  X  JS Products.js
backend > model > JS Order.js > [o] orderSchema
1  const mongoose = require('mongoose')
2  const Schema = mongoose.Schema
3
4  const orderSchema = new Schema({
5    userid: {
6      type: Schema.Types.ObjectId,
7      ref: 'User',
8    },
9    items: [
10      {
11        productid: {
12          type: Schema.Types.ObjectId,
13          ref: 'Product',
14        },
15        quantity: {
16          type: Number,
17          required: true,
18        },
19      },
20    ],
21    cost: {
22      type: Number,
23      required: true,
24    },
25    status: {
26      type: String,
27      required: true,
28      default: 'Processing',
29    },
30  })
31
32 module.exports = mongoose.model('Order', orderSchema)
```

# Product Model

```
Cart.js          JS Catalog.js      JS Order.js      JS Products.js X    JS User.js
backend > model > JS Products.js > [e] productSchema
1  const mongoose = require('mongoose')
2
3  const productSchema = mongoose.Schema(
4      {
5          name: { type: String, required: true },
6          description: { type: String, required: true },
7          price: { type: String, required: true },
8          category: { type: String, required: true },
9          image: { type: String, required: true },
10     },
11     [
12         {
13             timestamp: true,
14         }
15     ]
16 )
17 const productModel = mongoose.model('Product', productSchema)
18 module.exports = productModel
```

# User Model

```
S Cart.js          JS Catalog.js    JS Order.js    JS Products.js   JS User.js    X
backend > model > JS User.js > ...
1  const mongoose = require('mongoose')
2
3  const userSchema = mongoose.Schema(
4      {
5          firstname: { type: String, required: true },
6          lastname: { type: String, required: true },
7          email: { type: String, required: true },
8          password: { type: String, required: true },
9          street: { type: String, required: true },
10         money: { type: Number, default: 1000 },
11         isAdmin: { type: Boolean, default: false },
12     },
13     {
14         timestamp: true,
15     }
16 )
17
18 const userModel = mongoose.model('User', userSchema)
19 module.exports = userModel
20
```

Loading the data into the DB.

To load the data into the NoSQL database, I utilized a JSON file containing the product information and implemented a route in the backend to insert this data into the database.

## Creating code for CRUD data operations.

To implement CRUD (Create, Read, Update, Delete) operations for the data in the MongoDB database, several routes have been created in the backend server using Express.js.

## Products Routes ('products.js')

- Read Product by ID:
    - Route: 'GET /product/:id'
    - Retrieves a product by its ID from the database.
  - Read Products by Category
    - Route: 'GET /product/category/:category'
    - Retrieves all products belonging to a specific category.
  - Read All Products:
    - Route: 'GET /getallproduct'
    - Retrieves all products from the database.
  - Create Products from JSON File:
    - Route: 'GET /addallproduct'
    - Inserts all products from a JSON file into the database.

## Cart Routes ('cart.js')

- Read Cart Items:
  - Route: 'POST /getcart'
  - Retrieves the items in the user's cart.
- Delete Cart:
  - Route: 'POST /deletecart'
  - Deletes the entire cart of the user.
- Delete Item from Cart:
  - Route: 'POST /deleteobjectcart'
  - Deletes a specific item from the user's cart.
- Add Item to Cart:
  - Route: 'POST /addobjectcart'
  - Adds a new item (and its quantity) to the user's cart.

## Catalog Routes ('catalog.js')

- Add Product to Catalog:
    - Route: 'POST /addtocatalog'
    - Adds a product to the catalog with its quantity.
  - Read Product Quantity:
    - Route: 'GET /quantity/:id'
    - Retrieves the available quantity of a product.

## Order Routes ('orders.js')

- Add Order:
  - Route: 'POST /addorder'
  - Creates a new order with the user's cart items, deducts the total cost from the user's balance, and updates the catalog quantities.
- Read User Orders:
  - Route: 'POST /getorders'
  - Retrieves all orders made by the user along with product details.

## User Authentication Routes ('auth.js')

- User Login:
  - Route: 'POST /login'
  - Authenticates the user with email and password, generates a JWT token.
- User Registration:
  - Route: 'POST /register'
  - Registers a new user with hashed password and generates a JWT token.
- Verify User Role:
  - Route: 'POST /verifyrole'
  - Verifies if the user is an admin based on the JWT token.
- Get User Info:
  - Route: 'POST /user'
  - Retrieves user information from the JWT token.
- Get User Money Balance:
  - Route: 'POST /getmoneyavailable'
  - Retrieves the available money balance of the user from the JWT token.

# Graphical User Interface

**ESHOP by GDF**

All Products | Men Products | Women Products | Electronics | Jewelry | Sign In | Login | Chart

<b>Title:</b> WD 4TB Gaming Drive Works with Playstation 4 Portable External Hard Drive <b>Price:</b> 114€	<b>Title:</b> Fjallraven - Foldsack No. 1 Backpack, Fits 15 Laptops <b>Price:</b> 109.95€	<b>Title:</b> Mens Cotton Jacket <b>Price:</b> 55.99€	<b>Title:</b> WD 2TB Elements Portable External Hard Drive - USB 3.0 <b>Price:</b> 64€

# Graphical User Interface

The screenshot shows a 'Sign In' page with the following fields:

- First Name: Enter First Name
- Last Name: Enter Last Name
- Street: Enter Street
- Email address: Enter email

We'll never share your email with anyone else.
- Insert Password: Password
- Conform Your Password: Password

A 'Submit' button is located at the bottom of the form.

# Graphical User Interface

The screenshot shows a login form titled "Login". At the top right are links for "Sign In", "Login", and "Chart". The main form has fields for "Email address" (placeholder "Enter email") and "Password" (placeholder "Password"). Below the password field is a "Submit" button. A note below the email field states: "We'll never share your email with anyone else." The footer contains the copyright notice: "© 2024 Copyright: Giacomo Di Fabrizio".

# Graphical User Interface

ESHOP by GDF

Sign In [a](#) Chart Logout

Product

WD 4TB Gaming Drive Works with Playstation 4 Portable External Hard Drive

A black and blue rectangular external hard drive with a carbon fiber texture pattern on the front panel. The WD logo is visible on the top left corner.

Quantity: 2  
Price per unit: 114€

Remove from the chart

Total Price: 228€

Buy now

# Graphical User Interface

ESHOP by GDF

Sign In   a   Chart   Logout

## Orders

**Order id:** 66001008d01588d2c1c53fa3

**Order status:** Processing

**Products:**



Quantity: 1

Price: 114€



# Graphical User Interface

ESHOP by GDF

Sign In   a -   Chart   Logout



**Name:** WD 4TB Gaming Drive Works with Playstation 4 Portable External Hard Drive

**Description:**  
Expand your PS4 gaming experience. Play anywhere Fast and easy, setup Sleek design with high capacity, 3-year manufacturer's limited warranty

**Category:** electronics

**Price:** 114€

**Quantity available:** 69

**Desired quantity:**

[Add to the Chart](#)

© 2024 Copyright: Giacomo Di Fabrizio

# Graphical User Interface

ESHOP by GDF

Sign In [a](#) Chart Logout

---

Profile

**Firstname:**a

**Lastname:**a

**Email:**a

**Street:**a

**Money:**666.1€

# Implementation of the queries in the DB.

```
js cartRoute.js X
backend > routes > js cartRoute.js > router.post('/addobjectcart') callback
88 //Route to add object (and his quantity) in the cart
89 router.post('/addobjectcart', async (req, res) => {
90   try {
91     const { id, productid, quantity } = req.body
92     const user = await User.findOne({ _id: id })
93     const userId = user.id
94
95     let cart = await Cart.findOne({ userId: userId })
96
97     if (!cart) {
98       cart = new Cart({
99         userId: userId,
100        items: [{ productid: productid, quantity }],
101      })
102    } else {
103      const existingItem = cart.items.find(
104        (item) => String(item.productid) === String(productid)
105      )
106
107      if (existingItem) [
108        existingItem.quantity += parseInt(quantity)
109      ] else {
110        cart.items.push({ productid, quantity: parseInt(quantity) })
111      }
112    }
113
114    await cart.save()
115    res.status(200).json({ message: 'Object added to cart successfully', cart })
116  } catch (error) {
117    console.log(error)
118    res.status(500).json({ error: 'Error adding object in the cart' })
119  }
120})
```

# Implementation of the queries in the DB.

```

@orderRoutes.get('/addorder', async (req, res) => {
  try {
    const { id } = req.query;
    const user = await User.findById(id);
    if (!user) {
      return res.status(404).json({ error: 'User not found' });
    }
    const userId = user._id;
    const itemsInCart = await Cart.find({ userId: userId });
    if (!itemsInCart) {
      return res.status(404).json({ error: 'Cart not found' });
    }
    let productsInfo = [];
    for (const cart of itemsInCart) {
      for (const item of cart.items) {
        const product = await Product.findById(item.productId);
        if (product) {
          productsInfo.push({
            productId: product._id,
            quantity: item.quantity,
            price: product.price,
          });
        }
        // Aggiorna la quantità disponibile nel catalogo
        await Catalog.findByIdAndUpdate(
          { product: product._id },
          { $inc: { quantity: -item.quantity } }
        );
      }
    }
    let totalCost = 0;
    for (const item of productsInfo) {
      totalCost += parseFloat(item.quantity) * parseFloat(item.price);
    }
    // Create new order
    const newOrder = new Order({
      userId: userId,
      items: productsInfo,
      cost: totalCost,
    });
    if (user.money < totalCost) {
      res.status(404).json({ error: "You don't have enough money" });
    } else {
      await newOrder.save();
      await user.save();
      res
        .status(201)
        .json({ message: 'Order added successfully', order: newOrder });
    }
  } catch (error) {
    console.error(error);
    res.status(500).json({ error: 'Error adding order' });
  }
})

```

## Implementation of the queries in the DB.

```
_id: ObjectId('66000fc3d01588d2c1c53f35')
firstname: "a"
lastname: "a"
email: "a"
password: "$2b$10$Spx3ja2d2jl97Ur9pyGXV.DpKLYuDElAwI976Fje7ExbnrGIeLuy"
street: "a"
money: 666.1
isAdmin: false
__v: 0
```

# Thank You For The Attention