**E-commerce Application: Detailed Specifications**

**Hyunjun Kim**

**Abstract**

This document provides a comprehensive description of an e-commerce application, detailing the specifications of its primary components and their methods. The application includes essential classes such as User, Product, ShoppingCart, Order, and EcommerceApp, each with specific functionalities and constraints. The User class handles user initialization and email updates, enforcing strict validation rules. The Product class manages product details with constraints on name, price, and description. The ShoppingCart class allows adding products and viewing cart contents, addressing edge cases like boundary values and invalid quantities. The Order class processes orders with user information, items, address, and payment methods, ensuring all inputs are valid. Finally, the EcommerceApp class coordinates user registration, product listing, shopping cart management, checkout processing, and order tracking, with methods designed to handle edge cases and provide appropriate responses. The document outlines methods, parameters, return values, edge cases, and example inputs/outputs for each class, ensuring a robust and well-defined system.

**Class: User**

**Description:** Represents a user in the e-commerce application. Includes methods for initializing a user and updating their email address.

**Methods:**

1. \_\_init\_\_(username: str, password: str, email: str)
   * **Initializes** a user with a username, password, and email.
   * **Parameters:**
     + username (str): The username of the user. Must be a string of alphanumeric characters (English letters and numbers only), 3-20 characters long. No leading or trailing spaces.
     + password (str): The password of the user. Must be a string, minimum 8 characters long, and include at least one special character and one number. No restriction on the position of special characters.
     + email (str): The email address of the user. Must be in a valid email format (example@domain.com) and can be up to 254 characters long. Leading or trailing spaces will be removed.
   * **Returns:** None
   * **Raises:** ValueError if the username, password, or email is invalid.
   * **Requirements:**
     + For the **Username**, the length must be exactly 3 or 20 characters. Special characters are invalid, leading or trailing spaces are invalid, and mixed case (e.g., "User" vs "user") is distinguished.
     + For the **Password**, the length must be exactly 8 characters. Passwords containing only spaces are invalid, but passwords with spaces are valid. A password without special characters or numbers is invalid, while including special characters in various positions is valid.
     + For the **Email**, valid formats include subdomains and different TLDs (e.g., "user@sub.domain.com", "user@domain.co.uk"). Invalid formats include missing '@', missing domains, or invalid characters. Very long email addresses (up to 254 characters) are allowed. Mixed case (e.g., "User@domain.com" vs "user@domain.com") is distinguished.
   * **Example Inputs/Outputs:**

# Valid inputs

user = User('johndoe', 'password123!', 'johndoe@example.com')

print(user.username) # Output: 'johndoe'

print(user.email) # Output: 'johndoe@example.com'

# Invalid inputs

try:

user = User('jd', 'pass123!', 'invalid\_email')

except ValueError as e:

print(e) # Output: 'Invalid username, password, or email'

1. update\_email(new\_email: str)
   * **Updates** the user's email address.
   * **Parameters:**
     + new\_email (str): The new email address. Must be in a valid email format and can be up to 254 characters long. Leading or trailing spaces will be removed.
   * **Returns:** None
   * **Raises:** ValueError if the new email is invalid.
   * **Requirements: Requirements** include valid email formats as described above, as well as invalid email formats. It also covers very long email addresses (up to 254 characters) and mixed case usage (e.g., "User@domain.com" vs "user@domain.com").
   * **Example Inputs/Outputs:**

user = User('johndoe', 'password123!', 'johndoe@example.com')

user.update\_email(' john.doe@newmail.com ')

print(user.email) # Output: 'john.doe@newmail.com'

try:

user.update\_email('invalid\_email')

except ValueError as e:

print(e) # Output: 'Invalid email format'

**Class: Product**

**Description:** Represents a product in the e-commerce application. Includes methods for initializing a product with a name, price, and description.

**Methods:**

1. \_\_init\_\_(name: str, price: float, description: str)
   * **Initializes** a product with a name, price, and description.
   * **Parameters:**
     + name (str): The name of the product. Must be a string, 1-50 characters long.
     + price (float): The price of the product. Must be a float, between 0.01 and 10000.00.
     + description (str): A brief description of the product. Must be a string, 0-200 characters long.
   * **Returns:** None
   * **Raises:** ValueError if the name, price, or description is invalid.
   * **Requirements:** For the Name field, the length must be exactly 1 or 50 characters. A length of 0 characters is invalid, and a length of 51 characters is also invalid. Names containing special characters or only spaces are invalid. Leading or trailing spaces are not allowed, and names can include Unicode characters. Non-string values (e.g., numeric or None) are also not valid. For the Price field, the minimum allowed price is 0.01.
   * **Example Inputs/Outputs:**

# Valid inputs

product = Product('Laptop', 999.99, 'A high-performance laptop')

print(product.name) # Output: 'Laptop'

print(product.price) # Output: 999.99

# Invalid inputs

try:

product = Product('', 999.99, 'A high-performance laptop')

except ValueError as e:

print(e) # Output: 'Invalid product name'

try:

product = Product('Laptop', -999.99, 'A high-performance laptop')

except ValueError as e:

print(e) # Output: 'Invalid product price'

**Class: ShoppingCart**

**Description:** Represents a shopping cart in the e-commerce application. Includes methods for adding products to the cart and viewing the cart's contents.

**Methods:**

1. \_\_init\_\_()
   * **Initializes** an empty shopping cart.
   * **Parameters:** None
   * **Returns:** None
   * **Example Inputs/Outputs:**

cart = ShoppingCart()

print(cart.view\_cart()) # Output: []

1. add\_to\_cart(product: Product, quantity: int)
   * **Adds** a product to the shopping cart.
   * **Parameters:**
     + product (Product): The product to add.
     + quantity (int): The quantity to add. Must be an integer between 1 and 100.
   * **Returns:** None
   * **Raises:** ValueError if the quantity is invalid.
   * **Requirements:** For Boundary Values, the quantity must be exactly 1 for the lower boundary and exactly 100 for the upper boundary. For Invalid Quantity, adding a product with a quantity less than 1 or more than 100 is not allowed. Adding a product with a quantity of 0, a negative quantity (e.g., -1), or a non-integer quantity (e.g., 1.5 or "two") is also invalid. For Adding the Same Product Multiple Times, adding the same product repeatedly should either combine the quantities or update the existing quantity.
   * **Example Inputs/Outputs:**

cart = ShoppingCart()

product = Product('Laptop', 999.99, 'A high-performance laptop')

cart.add\_to\_cart(product, 2)

print(cart.view\_cart()) # Output: [{'product': <Product instance>, 'quantity': 2}]

try:

cart.add\_to\_cart(product, 101)

except ValueError as e:

print(e) # Output: 'Invalid quantity'

1. view\_cart()
   * **Returns** the contents of the shopping cart.
   * **Parameters:** None
   * **Returns:** List of items in the cart (List[Dict[str, Union[Product, int]]). Each item in the list is a dictionary with keys product (Product) and quantity (int).
   * **Requirements:** Viewing an empty cart should return an empty list. Additionally, the list returned by view\_cart() should not allow any modifications that affect the internal state of the cart.
   * **Example Inputs/Outputs:**

cart = ShoppingCart()

print(cart.view\_cart()) # Output: []

product = Product('Laptop', 999.99, 'A high-performance laptop')

cart.add\_to\_cart(product, 1)

print(cart.view\_cart()) # Output: [{'product': <Product instance>, 'quantity': 1}]

**Class: Order**

**Description:** Represents an order in the e-commerce application. Includes methods for initializing an order and updating the order's status.

**Methods:**

1. \_\_init\_\_(user: User, items: List[Dict[str, Union[Product, int]]], address: str, payment\_method: str)
   * **Initializes** an order with user, items, address, and payment method.
   * **Parameters:**
     + user (User): The user who placed the order. Must not be None.
     + items (List[Dict[str, Union[Product, int]]]): The list of items in the order. Each item is a dictionary with 'product' (Product) and 'quantity' (int) keys. The list must not be empty, and must not contain None values or products with quantity 0.
     + address (str): The shipping address for the order. Must be a string, 1-100 characters long. Must not contain only spaces or non-printable characters.
     + payment\_method (str): The payment method used for the order. Must be one of ['credit\_card', 'debit\_card', 'paypal']. Must not be an empty string and must match the exact case.
   * **Returns:** None
   * **Raises:**
     + ValueError if any of the parameters are invalid.
   * **Requirements:** For the User Parameter, if the user is None, it is considered invalid. For the Items List, it is invalid if the list is empty, contains a None value, or includes a product with a quantity of 0. For the Address, an address with a length of 0 characters is invalid. The length can be exactly 1 or 100 characters, but exceeding 100 characters is invalid. Addresses with special characters, non-printable characters, or only spaces are also considered invalid. For the Payment Method, it must be exactly one of the allowed values. A payment method is invalid if it is an empty string, contains spaces, or has case variations.
   * **Example Inputs/Outputs:**

user = User('johndoe', 'password123!', 'johndoe@example.com')

product = Product('Laptop', 999.99, 'A high-performance laptop')

items = [{'product': product, 'quantity': 2}]

order = Order(user, items, '123 Main St', 'credit\_card')

print(order.address) # Output: '123 Main St'

try:

order = Order(None, items, '123 Main St', 'credit\_card')

except ValueError as e:

print(e) # Output: 'Invalid user'

1. update\_status(new\_status: str)
   * **Updates** the status of the order.
   * **Parameters:**
     + new\_status (str): The new status of the order. Must be one of ['Processing', 'Shipped', 'Delivered', 'Cancelled']. Must not be an empty string and must match the exact case.
   * **Returns:** None
   * **Raises:**
     + ValueError if the new status is invalid.
   * **Requirements:** For the New Status, an empty string is considered invalid. A valid status in different cases (e.g., 'processing', 'SHIPPED') is allowed, but status changes must follow a logical progression, meaning you cannot skip steps like going from 'Processing' to 'Delivered'. Invalid status transitions (e.g., 'Delivered' to 'Processing') are not allowed. Once a status is 'Cancelled', it cannot transition to any other status. Lastly, changing to the same status (e.g., 'Processing' to 'Processing') is also not allowed.
   * **Example Inputs/Outputs:**

user = User('johndoe', 'password123!', 'johndoe@example.com')

product = Product('Laptop', 999.99, 'A high-performance laptop')

items = [{'product': product, 'quantity': 2}]

order = Order(user, items, '123 Main St', 'credit\_card')

order.update\_status('Shipped')

print(order.status) # Output: 'Shipped'

try:

order.update\_status('InvalidStatus')

except ValueError as e:

print(e) # Output: 'Invalid status'

**Class: EcommerceApp**

**Description:** Manages the e-commerce application, coordinating user registration, product listing, and orders. Includes methods for registering users, adding products, managing the shopping cart, processing checkouts, and tracking orders.

**Methods:**

1. \_\_init\_\_()
   * **Initializes** the e-commerce application with empty databases for users, products, and orders.
   * **Parameters:** None
   * **Returns:** None
   * **Example Inputs/Outputs:**

app = EcommerceApp()

print(app.users) # Output: []

1. register\_user(username: str, password: str, email: str) -> bool
   * **Registers** a new user in the e-commerce application.
   * **Parameters:**
     + username (str): The username for the new user. Must be a string of alphanumeric characters, 3-20 characters long.
     + password (str): The password for the new user. Must be a string, 8-20 characters long, and cannot include spaces.
     + email (str): The email address for the new user. Must be in a valid email format, including a local part and a domain with a TLD up to 10 characters long.
   * **Returns:** bool, True if registration is successful, False otherwise.
   * **Raises:** ValueError if the username, password, or email is invalid.
   * **Requirements:** The Username and Email must not be already taken. The Username length should be exactly 3 or 20 characters, and usernames with spaces or special characters are not allowed. For the Password, the length must be exactly 8 or 20 characters, and it should include special characters without spaces. Valid Email formats include those with subdomains and different TLDs, with a total length of up to 10 characters. Invalid Email formats include those missing '@', missing domain, containing invalid characters or spaces, or lacking a local part. Emails with mixed case are also allowed, but duplicate usernames and emails should be checked in a case-insensitive manner.
   * **Example Inputs/Outputs:**

app = EcommerceApp()

success = app.register\_user('johndoe', 'password123!', 'johndoe@example.com')

print(success) # Output: True

try:

app.register\_user('jd', 'pass123!', 'invalid\_email')

except ValueError as e:

print(e) # Output: 'Invalid username, password, or email'

1. add\_product(name: str, price: float, description: str) -> bool
   * **Adds** a new product to the product listing.
   * **Parameters:**
     + name (str): The name of the product. Must be a string, 1-50 characters long, not empty, and not just spaces.
     + price (float): The price of the product. Must be a float, between 0.01 and 10000.00.
     + description (str): A brief description of the product. Must be a string, 0-200 characters long.
   * **Returns:** bool, True if the product is added successfully, False otherwise.
   * **Raises:** ValueError if the name, price, or description is invalid.
   * **Requirements:** The Name length must be exactly 1 or 50 characters, and names can include special characters. However, an Empty Name or a name with just spaces is not valid. For Price, it must be at the minimum value of 0.01 or the maximum value of 10000.00. P rices with very small decimals are allowed, but Zero or Negative Prices are invalid. The Description length should be exactly 0 or 200 characters. Descriptions can include special characters and emojis, but excessively Long Descriptions are not permitted.
   * **Example Inputs/Outputs:**

app = EcommerceApp()

success = app.add\_product('Laptop', 999.99, 'A high-performance laptop')

print(success) # Output: True

try:

app.add\_product('', 999.99, 'A high-performance laptop')

except ValueError as e:

print(e) # Output: 'Invalid product name'

1. add\_to\_cart(username: str, product\_id: int, quantity: int) -> bool
   * **Adds** a specified quantity of a product to the user's shopping cart.
   * **Parameters:**
     + username (str): The username of the user adding the product to the cart. Must be a string of alphanumeric characters, 3-20 characters long.
     + product\_id (int): The ID of the product to be added. Must correspond to a valid product ID.
     + quantity (int): The quantity of the product to be added. Must be an integer between 1 and 100.
   * **Returns:** bool, True if the product is added successfully, False otherwise.
   * **Requirements:** The Quantity must be exactly 1 or 100. When adding the same product multiple times, it should either combine the quantities or update the existing quantity. Adding a product with a quantity less than 1 or more than 100 is considered invalid, and adding a product with a quantity of 0 is also invalid. Additionally, an Invalid Product ID is not acceptable, and adding to the cart without being logged in is not allowed.
   * **Example Inputs/Outputs:**

app = EcommerceApp()

app.register\_user('johndoe', 'password123!', 'johndoe@example.com')

app.add\_product('Laptop', 999.99, 'A high-performance laptop')

success = app.add\_to\_cart('johndoe', 0, 2)

print(success) # Output: True

try:

app.add\_to\_cart('johndoe', 0, 101)

except ValueError as e:

print(e) # Output: 'Invalid quantity'

1. checkout(username: str, address: str, payment\_method: str) -> int
   * **Processes** the checkout for the user's current cart and generates an order.
   * **Parameters:**
     + username (str): The username of the user checking out. Must be a string of alphanumeric characters, 3-20 characters long.
     + address (str): The shipping address for the order. Must be a string, 1-100 characters long.
     + payment\_method (str): The payment method for the order. Must be one of ['credit\_card', 'debit\_card', 'paypal'].
   * **Returns:** int, The order ID if the checkout is successful, -1 otherwise.
   * **Requirements:** The Address length must be exactly 1 or 100 characters, and addresses can include special characters. The Payment Method must be exactly one of the allowed values, and a payment method with spaces is considered invalid. Additionally, Checkout with an empty cart is not allowed.
   * **Example Inputs/Outputs:**

app = EcommerceApp()

app.register\_user('johndoe', 'password123!', 'johndoe@example.com')

app.add\_product('Laptop', 999.99, 'A high-performance laptop')

app.add\_to\_cart('johndoe', 0, 2)

order\_id = app.checkout('johndoe', '123 Main St', 'credit\_card')

print(order\_id) # Output: Some integer representing the order ID

try:

order\_id = app.checkout('johndoe', '', 'credit\_card')

except ValueError as e:

print(e) # Output: 'Invalid address'

1. track\_order(order\_id: int) -> Dict[str, Union[str, List[Dict[str, Union[Product, int]]]]]
   * **Retrieves** the status of a specific order.
   * **Parameters:**
     + order\_id (int): The ID of the order to be tracked. Must correspond to a valid order ID.
   * **Returns:** dict, A dictionary containing order details and status, or None if the order ID is invalid.
   * **Requirements:** Invalid order IDs, including negative values and excessively large values.
   * **Example Inputs/Outputs:**

app = EcommerceApp()

app.register\_user('johndoe', 'password123!', 'johndoe@example.com')

app.add\_product('Laptop', 999.99, 'A high-performance laptop')

app.add\_to\_cart('johndoe', 0, 2)

order\_id = app.checkout('johndoe', '123 Main St', 'credit\_card')

order\_status = app.track\_order(order\_id)

print(order\_status) # Output: {'username': 'johndoe', 'items': [{'product': <Product instance>, 'quantity': 2}], 'address': '123 Main St', 'payment\_method': 'credit\_card', 'status': 'Processing'}

invalid\_status = app.track\_order(-1)

print(invalid\_status) # Output: None