Problem 1
Det saknas Dependency Injection, controllers instansierar services direkt i själva controllern. Detta bryter mot “Dependency Inversion Principle” där controllers är beroende av instansierade objekt. Lösningen är att implementera Dependency Injection (DI) genom att registrera alla objekt i “Program.cs” och injicera de via konstruktorn.

Problem 2
Det saknas Interfaces för services. Detta bryter mot “Interface Segregation Principle”, då det inte finns någon separation. Detta gör det även omöjligt att skapa mocks och stubs. Lösningen är att skapa interfaces för services, repositories och andra business logik, för att sedan registrera de med DI.

Problem 3
Repository pattern saknas, services hanterar både data hantering och business logik. Detta bryter mot “Single Responsibility Principle" då services ansvarar för både business logik och data hantering. Lösningen är att implementera Repository Pattern med separata repository klasser för att hantera data åtkomst.

Problem 4
Entities är definierade i service klasserna. Detta bryter mot SRP för att services har hand om både business logik och entity modeller. Det blir också svårare att återanvända entities i andra lager. Lösningen är att skapa entitiy klasser där alla entities går att återanvända i projektet.

Problem 5
DTO:s är definierade i controllers. Liknande problem som “Problem 4”, där det definieras datastrukturer i controllers som bryter mot SRP. Lösningen är att separera DTO:sen till DTO klasser.

Problem 6
Det finns duplicerad Auth logik på flera rader i controllers. Det bryter mot “Dont Repeat Yourself” (DRY) där det finns flera rader duplicerad kod på många olika ställen i projektet. Lösningen är att implementera ett custom attribute som sköter auth automatisk, när en kontroller har attributet.

Problem 7
Lösenord sparas i plain text och har ingen kryptering. Detta kan leda till att ifall data läcks så finns lösenord exponerat till de som har data, samt så bryter det mot GDPR. Lösningen är att använda “BCryptPasswordHasher” för att kryptera alla lösenord innan de sparas i datalagringen.

Problem 8
Det finns business logik i controllers, då controllers ska bara hantera HTTP request och ingen business logik så bryter detta mot SRP. Lösningen är att flytta all business logik till Command Handlers med hjälp av CQRS och mediator pattern.

Problem 9

Det finns några “Magic strings” (Hårdkodade värden) som till exempel “X-Auth-Token” i koden. Ifall man ska byta ut dessa strängar så är det många platser där det behövs ändras på strängar och detta bryter mot DRY. Lösningen är att skapa konstant klasser där till exempel “X-Auth-Token” sätts i en konstant och sedan använder man konstanten, och ifall man vill byta ut strängen, kan man bara ändra i konstant klassen.

Problem 10
Det finns listor som är av typen “Object” vilket gör att det inte är type safe. Istället ska det vara entities som används istället för “Object” till exempel läggs alla order produkter och information om odern i en lista av typen “Object”. Istället ska det vara en entity som tar hand om produkter och information av ordern som sedan listan har typen av.

Problem 11
Det finns ingen möjlighet att ha transaktioner, om en order läggs kommer den att minska produkt stock, skapa order och sedan rensa cart. Ifall ett steg misslyckas kan det bli data fel, till exempel att stock minskas men det skapas ingen order. Detta bryter mot ACID principerna. Lösningen är att implementera Unit Of Work pattern som stöder transaktioner, som säkerställer att data sparas för än alla steg är genomförda eller inte spara data ifall något steg går fel.

Problem 12
Det finns ingen strukturerad error hantering, fel hanteras på olika ställen i koden och har ingen centraliserad punkt där den hanterar errors. Lösningen är att implementera Result Pattern som strukturerar upp felhanteringen och hantera alla fel i business logik. Detta hjälper med prestanda, då inga throws används och det görs så att business logiken följer samma struktur med error hantering i hela projektet.

Problem 13
Det finns en del otydliga variabel namn som inte visar vad de variablerna lagrar. Detta bryter mot att koden ska vara självdokumenterande i clean code. Lösningen är att ändra variabel namnen så att det är mer självförklarande.

Implementerade Design Patterns

Repository Pattern
Repository Pattern separerar business logiken från datahanterings logiken, genom att abstrahera dataåtkomsten till en plats i projektet. Det gör så att all dataåtkomst logik är lätt att ändra ifall man vill byta ut datalagringen från in-memory till SQL. Repository Pattern följer SRP i SOLID, där den bara kan hantera dataåtkomst.

Unit Of Work Pattern
Unit Of Work säkerställer att alla ändringar som sker i business logiken antingen går igenom eller inte alls, ifall det är en transaktion. Detta följer Atomicity i ACID, då alla ändringar ska lyckas eller så händer inga ändringar alls. Detta skapar så att man kan rulla tillbaka data ifall något går fel för att behålla data integritet. Det skapar också en instans som alla repositories använder.

CQRS (Command Query Responsibility Segregation)

CQRS separerar på läs operationer från skrivoperationer i olika objekt. Detta skapar en tydlig separation mellan operationer som läser data och operationer som skriver data vilket följer SRP i SOLID. På grund utav allt separeras till små bitar, gör det lätt att isolera tester. Det skapar också en tydlighet på vad de olika operationerna gör, ifall det är ett command vet man att ett state kommer att ändras och om det är en query var man att endast data hämtas.

Mediator Pattern
Mediator Pattern minskar direkt koppling mellan olika komponenter genom att låta de kommunicera med en mediator. Mediator följer “Open/Closed Principle” och skapar en loose coupling då controller inte behöver känna till command och query handlers, de behöver bara känna till mediatorn. Detta gör så att controllers ända syfte är att hantera HTTP requests och översätta de till commands och queries, för att sedan skicka de till mediatorn, vilket följer SRP.

Result Pattern
Result Pattern returnerar ett resultat objekt som innehåller ett lyckat värde eller ett misslyckat värde. Detta används istället för att kasta exceptions. Det gör så man måste hantera felen istället för att anta att allt går bra. Det leder också till bättre prestanda då det inte kastas eller fångas exceptions som är en dyr sak i .NET.

Adapter Pattern
Adapter Pattern används för lösenords hashning. Där business logiken är helt oberoende av vilket hashings bibliotek som används, då interfacet är samma på de olika klasserna.

Test strategi
Testerna kommer använda xUnit för att skriva och köra tester, Moq för att skapa mocks och stubs av interfaces och Fakes för att testa repositories. All business logik ska testat genomförligt, inkluderat alla command och query handlers, samt repositories.

Jag kommer använda mockning för att testa command handlers och den interaktionen den har med repositories och Unit Of Work.

Jag kommer använda stubs för att simulera olika situationer och hur det hanteras.

Jag kommer använda Fakes för att skapa en fake databas för att kunna testa hur databasen fungerar med de olika repositories:en.

Testerna kommer att följa Arrange, Act och Assert och kommer att täcka både lyckade fall och misslyckade fall.

Reflektion
Kod kvaliteten har fått ett tydligt struktur som följer de olika clean code principerna, med små klasser med löss koppling och beskrivande namn som förmedlar vad det är. Kod designen gör det även lätt att testa alla funktioner oberoende med en testtäckning på 63 tester. Då koden är löst kopplad, har DI och följer SOLID principerna gör det att projektet har en underhållbarhet som är lätt att utöka funktionalitet på. Med hjälp av CQRS ökar skalbarheten med separata read och writes som gör att de kan skalas och optimeras separat. Repository

Pattern gör det enkelt att byta datahantering. Ytterligare förbättringar skulle vara att ha loggning för att få en bättre inblick i projektet när det körs, för tillfället är den ända informationen som kommer fram är från API:et. Sen skulle projektet blivit bättre ifall det fanns en riktig databas som är kopplad till projektet med hjälp av till exempel EF Core och en SQL server, för att öka prestanda. Containerisering med hjälp av Docker för att enklare kunna deploy:a och skala i molnmiljö. Refaktoreringen har gjort det så att det inte kommer vara en stor/jobbig uppgift att utöka denna funktionalitet och annan funktionalitet som kan före komma.