Assignment 1

Full Name : Hoang Minh Toan

ID: B21DCCN713

Class: E21CNPM03

Nội dung

[1]

1. Why Study Software Architecture and Design?

Why Study Software Architecture and Design?

Introduction

Software architecture and design are fundamental disciplines in software engineering that influence the quality, maintainability, scalability, and overall success of a software system. Understanding these concepts is crucial for software developers, engineers, and architects to create efficient, robust, and adaptable software applications. This paper explores the importance of studying software architecture and design by examining its role in ensuring system quality, facilitating scalability and performance, enhancing maintainability, supporting team collaboration, and improving security and risk management.

a. Ensuring System Quality

One of the primary reasons for studying software architecture and design is to ensure high-quality software systems. A well-designed architecture provides a structured approach to software development, minimizing technical debt and ensuring that the system meets functional and non-functional requirements. Quality attributes such as performance, security, reliability, and usability are heavily influenced by architectural decisions.

For example, choosing the right architectural patterns—such as microservices for scalability or layered architecture for modularity—ensures that the system is robust and meets user expectations. The selection of appropriate design principles such as Domain-Driven Design (DDD) and Clean Architecture helps to structure applications in a way that makes them adaptable to business changes over time.

Poor architectural decisions can lead to systems that are difficult to maintain, scale, or extend. By studying software architecture, developers can learn how to design systems that are resilient to change and can adapt to evolving business needs. Architectural principles such as separation of concerns, modularity, and abstraction play a vital role in reducing complexity and improving system quality. Additionally, techniques such as Test-Driven Development (TDD) and Behavior-

[1]

Driven Development (BDD) contribute to higher quality assurance throughout the development lifecycle.

b. Facilitating Scalability and Performance

Software systems often need to handle increasing workloads, user demands, and data processing requirements. A well-architected system can efficiently scale up or down based on these demands. Studying software architecture helps engineers understand how to design scalable systems using techniques like load balancing, caching, distributed computing, and database sharding.

For instance, cloud-based applications require a scalable architecture to support millions of users simultaneously. Knowledge of architecture patterns like microservices and event-driven architectures enables developers to build systems that can scale horizontally and efficiently utilize computing resources. Without proper architectural planning, systems may experience performance bottlenecks, leading to poor user experiences and potential business losses. Additionally, adopting distributed systems methodologies, such as the CAP theorem (Consistency, Availability, Partition Tolerance), helps developers understand trade-offs in distributed computing environments.

Performance optimization is another key aspect of software architecture. Efficient memory management, optimized database queries, asynchronous processing, and proper resource allocation can significantly improve the performance of a system. Developers and architects must also consider network latency, parallel processing, and real-time data processing when designing high-performance applications.

c. Enhancing Maintainability and Flexibility

Software applications require continuous updates, bug fixes, and feature enhancements. A poorly designed system can make maintenance challenging and time-consuming. Software architecture and design principles help create modular and loosely coupled systems that facilitate easier modifications and extensions.

By adhering to design principles such as SOLID (Single Responsibility, Open/Closed, Liskov Substitution, Interface Segregation, and Dependency Inversion), developers can build maintainable software that reduces the likelihood of introducing errors during updates. Additionally, architecture styles such as microservices allow teams to independently update and deploy components without disrupting the entire system.

Understanding software design patterns—such as Factory, Observer, and Strategy patterns—also aids in creating reusable and flexible code, making it easier to adapt to new requirements and technological advancements. Software architects must also consider versioning strategies, backward compatibility, and documentation standards to ensure seamless evolution of the software over time.

d. Supporting Team Collaboration and Communication

Software development is a collaborative effort that involves multiple stakeholders, including developers, project managers, business analysts, and quality assurance engineers. A well-defined software architecture serves as a blueprint that facilitates clear communication among team members.

When software teams follow architectural guidelines, it ensures consistency in coding practices, documentation, and design decisions. This is especially important for large projects with distributed teams, where clear architectural documentation helps onboard new developers and maintain coherence across different components of the system.

Moreover, software architecture plays a critical role in agile and DevOps methodologies. Continuous integration and continuous deployment (CI/CD) pipelines depend on well-architected software that supports automated testing, monitoring, and version control. Studying software architecture empowers developers to implement best practices that improve team collaboration and overall project success.

Collaboration tools such as architectural decision records (ADRs), UML diagrams, and API documentation further streamline communication and ensure that all team members have a shared understanding of the system's structure and evolution.

e. Improving Security and Risk Management

Security is a critical concern in software development, and poor architectural decisions can introduce vulnerabilities that attackers can exploit. By studying software architecture, developers learn how to implement security best practices, such as authentication, authorization, data encryption, and secure API design.

A well-architected system incorporates security measures at every level, from the database to the user interface. Architecture patterns like Zero Trust Security, microservices security models, and defense-in-depth strategies help protect sensitive data and prevent unauthorized access. Secure software development practices such as threat modeling, security testing, and compliance with industry standards (e.g., OWASP, GDPR, ISO 27001) ensure that security is integrated into the architecture from the outset.

Risk management is another essential aspect of software architecture. By analyzing potential risks early in the development lifecycle, architects can design systems that are resilient to failures and cyber threats. Techniques such as fault tolerance, redundancy, disaster recovery planning, and high availability ensure that the system remains operational even in the face of unexpected failures. The implementation of observability and monitoring solutions, such as logging frameworks and alerting mechanisms, further enhances system security and reliability.

f. Advantages and Disadvantages of Studying Software Architecture and Design

Advantages:

- Better System Quality – Leads to more reliable, scalable, and maintainable software.
- Improved Performance – Helps optimize systems for high efficiency and fast response times.
- Enhanced Collaboration – Provides a common framework for teams to follow, improving communication.
- Greater Flexibility – Allows for easier modifications, reducing long-term costs.
- Security Strength – Ensures secure design principles are integrated from the beginning.
- Future-Proofing – Enables systems to evolve with technological advancements and business requirements.

Disadvantages:

- Steep Learning Curve – Requires understanding of various patterns, methodologies, and frameworks.

- Time-Consuming – Designing architecture takes time and effort before actual development starts.
- Resource Intensive – Needs skilled professionals, tools, and thorough documentation.
- Over-Engineering Risk – Can lead to overly complex systems if not balanced properly.
- High Initial Costs – Investment in architecture design and planning may seem costly in the short term.

Conclusion

Studying software architecture and design is essential for building high-quality, scalable, maintainable, and secure software systems. While there are some challenges, the long-term benefits outweigh the disadvantages. By mastering these principles, developers and engineers can create resilient systems that meet modern demands and technological advancements. Whether working on enterprise applications, mobile apps, or cloud-native solutions, a deep understanding of software architecture remains a cornerstone of successful software engineering.

2. Comparison of Monolithic and Microservices Architectures, and the Relationship Between Microservices and Agile Development

a. Introduction

As software development evolves, different architectural paradigms have emerged to address scalability, maintainability, and deployment concerns. Two prominent approaches are **Monolithic Architecture** and **Microservices Architecture**. Each has its advantages and drawbacks, depending on the nature of the software project. Additionally, the relationship between **Microservices and Agile Development** has become a crucial topic, as both emphasize flexibility, iterative improvements, and faster delivery. This paper explores these architectures in detail and examines how microservices align with Agile methodologies.

b. Monolithic vs. Microservices Architecture

Monolithic Architecture

Monolithic architecture is a traditional software development approach where an entire application is built as a single, unified system. This means that all components (user interface, business logic, and database access) are tightly coupled and run as a single codebase.

Characteristics:

1. **Single Codebase**: All functionalities reside in one repository.
2. **Unified Deployment**: The entire application is deployed as a single unit.
3. **Centralized Data Management**: A single database is used across all functionalities.

Advantages:

1. **Simpler Development and Deployment**: A single codebase makes it easier for developers to build, test, and deploy the application.
2. **Better Performance**: Since components are tightly integrated, communication between them is faster.
3. **Easier Debugging and Testing**: With everything in one place, developers can test the entire system without worrying about network-related issues.
4. **Faster Initial Development**: Since all parts of the system are centralized, early development can be quicker.

Disadvantages:

1. **Scalability Challenges**: Scaling specific parts of an application independently is difficult, often requiring the entire system to scale.
2. **Limited Flexibility**: Changes to one part of the system may require a full redeployment, increasing the risk of breaking functionality.
3. **Slower Development and Deployment Cycles**: Larger applications take longer to build, test, and deploy, slowing down iteration speed.

4. **Technology Lock-in**: A monolithic approach makes it difficult to adopt new technologies without reworking significant portions of the codebase.

Microservices Architecture

Microservices architecture decomposes an application into small, independent services that communicate via APIs. Each service has its own database, allowing for independent deployment and scaling.

Characteristics:

1. **Decentralized Services**: Each functionality is an independent service.
2. **Independent Deployment**: Services can be updated without affecting the entire system.
3. **Polyglot Persistence**: Different services can use different databases and technologies.

Advantages:

1. **Scalability**: Each microservice can be scaled independently based on demand.
2. **Flexibility in Development**: Different teams can work on separate services using different technologies.
3. **Faster Deployment and Iteration**: Microservices enable continuous integration and deployment, allowing for quicker feature releases and updates.
4. **Improved Fault Isolation**: If one service fails, it does not necessarily bring down the entire system.
5. **Better Adaptability to Agile**: Microservices allow for frequent updates and faster responses to changing business needs.

Disadvantages:

1. **Increased Complexity**: Managing multiple services introduces challenges in deployment, communication, and debugging.

2. **Latency Overhead**: Inter-service communication (e.g., via REST or gRPC) may introduce performance bottlenecks.
3. **Operational Overhead**: Requires robust DevOps practices, including monitoring, service discovery, and fault tolerance mechanisms.
4. **Difficult Data Consistency Management**: Since services have independent databases, ensuring consistency across services can be complex.

c. Microservices and Agile Development

The Agile Methodology

Agile is a software development methodology focused on iterative progress, collaboration, and customer feedback. Agile principles emphasize:

- **Incremental Development**: Delivering software in small, manageable parts.
- **Customer Collaboration**: Continuous feedback from users.
- **Flexibility and Adaptability**: Responding to changing requirements efficiently.

How Microservices Support Agile Development

1. **Independent Development and Deployment**: Agile teams can build and release microservices independently, aligning with Agile's goal of delivering value incrementally.
2. **Faster Iterations**: With microservices, teams can release updates frequently without affecting the entire system.
3. **Improved Collaboration**: Microservices allow teams to work on separate services simultaneously, improving efficiency and parallel development.
4. **Resilience and Fault Isolation**: Unlike monolithic applications, failures in one microservice do not bring down the entire system, supporting continuous delivery.
5. **Scalability of Teams**: Microservices enable organizations to scale development teams efficiently by assigning different teams to different services.

6. **Technology Diversity**: Different teams can choose the best technologies for their specific microservices without being constrained by a single technology stack.

Challenges of Combining Microservices with Agile

While microservices align well with Agile principles, challenges exist:

1. **Complexity in Management**: Requires robust DevOps, CI/CD pipelines, and monitoring tools.
2. **Testing Difficulties**: Ensuring seamless integration among multiple services can be complex.
3. **Increased Communication Overhead**: Teams must coordinate effectively to avoid inconsistencies in APIs and service dependencies.
4. **Data Synchronization Issues**: Managing transactions and data consistency across distributed services can be challenging.
5. **Security Considerations**: A distributed system requires careful authentication and authorization mechanisms to prevent vulnerabilities.

Conclusion

Both monolithic and microservices architectures have their advantages and trade-offs, depending on the project requirements. While monolithic applications offer simplicity and easier debugging, microservices provide scalability and flexibility, making them well-suited for Agile development. The combination of microservices and Agile methodologies enables faster development cycles, independent deployments, and improved team collaboration. However, adopting microservices requires careful planning, effective DevOps strategies, and a robust testing framework to manage the complexity. As businesses continue to demand rapid innovation, microservices and Agile will remain crucial approaches for modern software development.

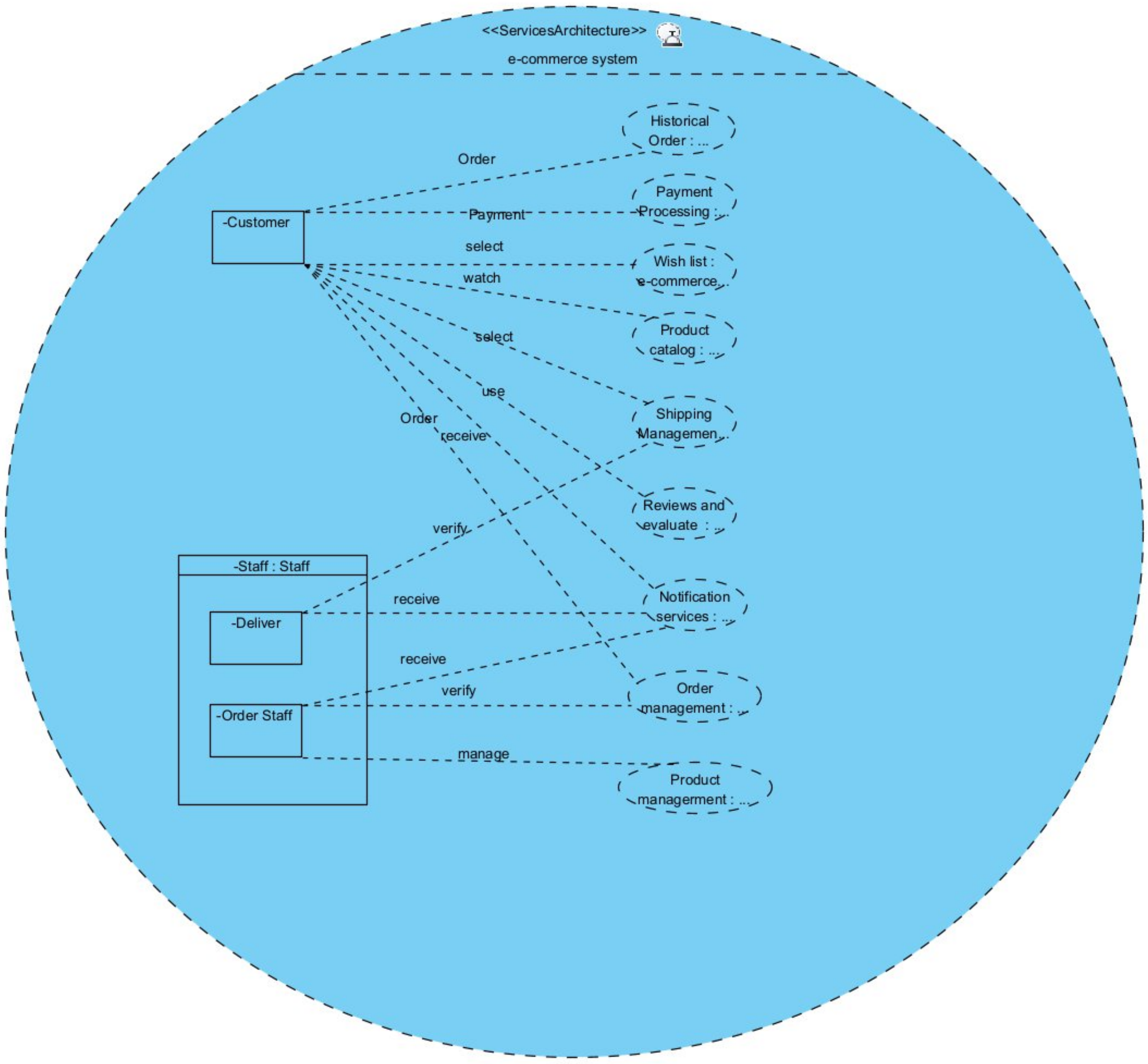
Organizations must evaluate their specific needs before choosing an architectural approach. For smaller projects or teams with limited resources, monolithic architecture may be the better option. However, for large-scale applications requiring rapid iterations, microservices and Agile development provide a powerful combination. The key to success is proper planning, automation, and a strong emphasis on collaboration across development teams.

3. Decompose a software system in microservices and using tool in VP

a. Decompose e-commerce system

- Users : Customers, Deliver , Order Staff

- Services:
+ Product Catalog
+ Historical Order
+ Wish List
+ Payment Process
+ Reviews and Evaluation
+ Notification Services
+ Order Management
+ Product Management
+ Shipping Management

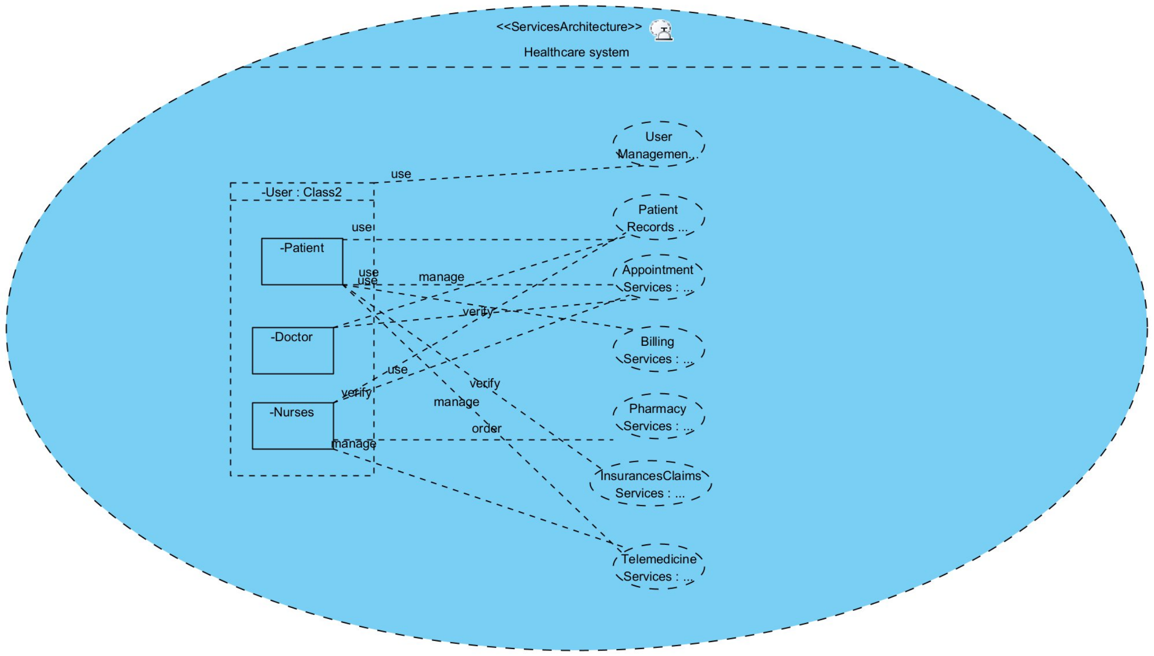


b. Decompose medicine system

- Users: Patients , Doctor, Nurse

- Services :

+ User Management
+ Patient Records
+ Appointment Services
+ Billing Services
+ Pharmacy Services
+ Insurances Management
+ Telemedicine Services

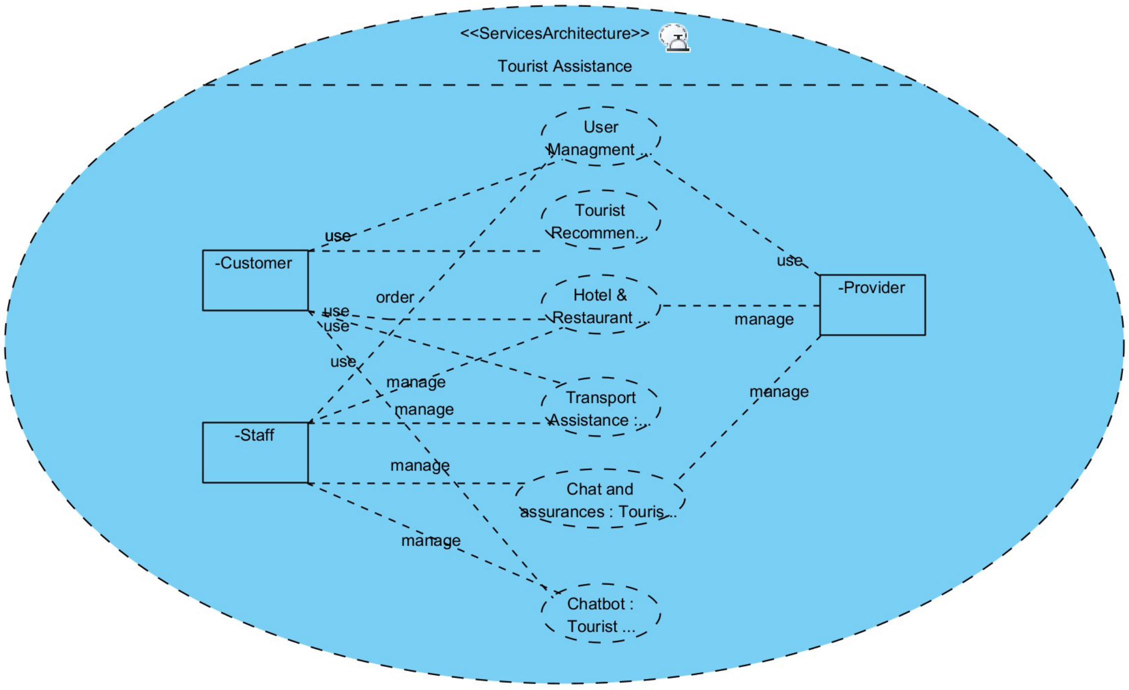


c. Decompose Tourist Assistant System

- User : Customer , Provider, Staff

- Services :

+ User Management
+ Tourist Recommendation
+ Hotel and Restaurant System
+ Transport Assistance
+ Chat and Assurance
+ Tourist :

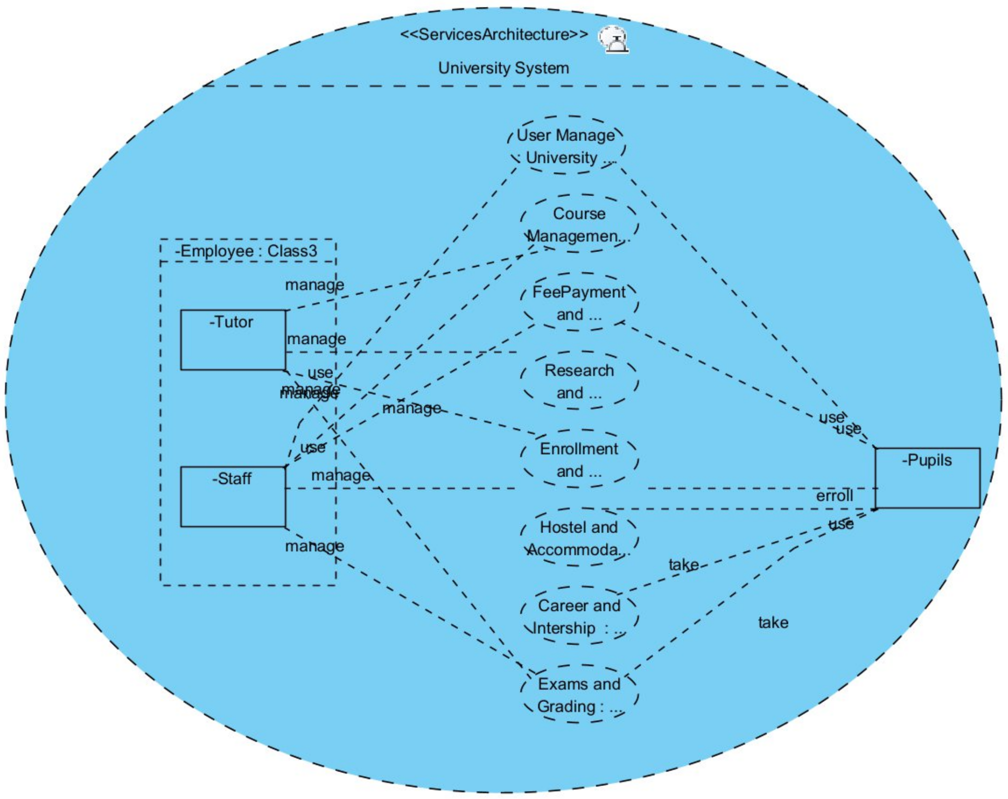


d. Decompose University Management System

- Users: Tutor, Staff, Pupils

- Services :

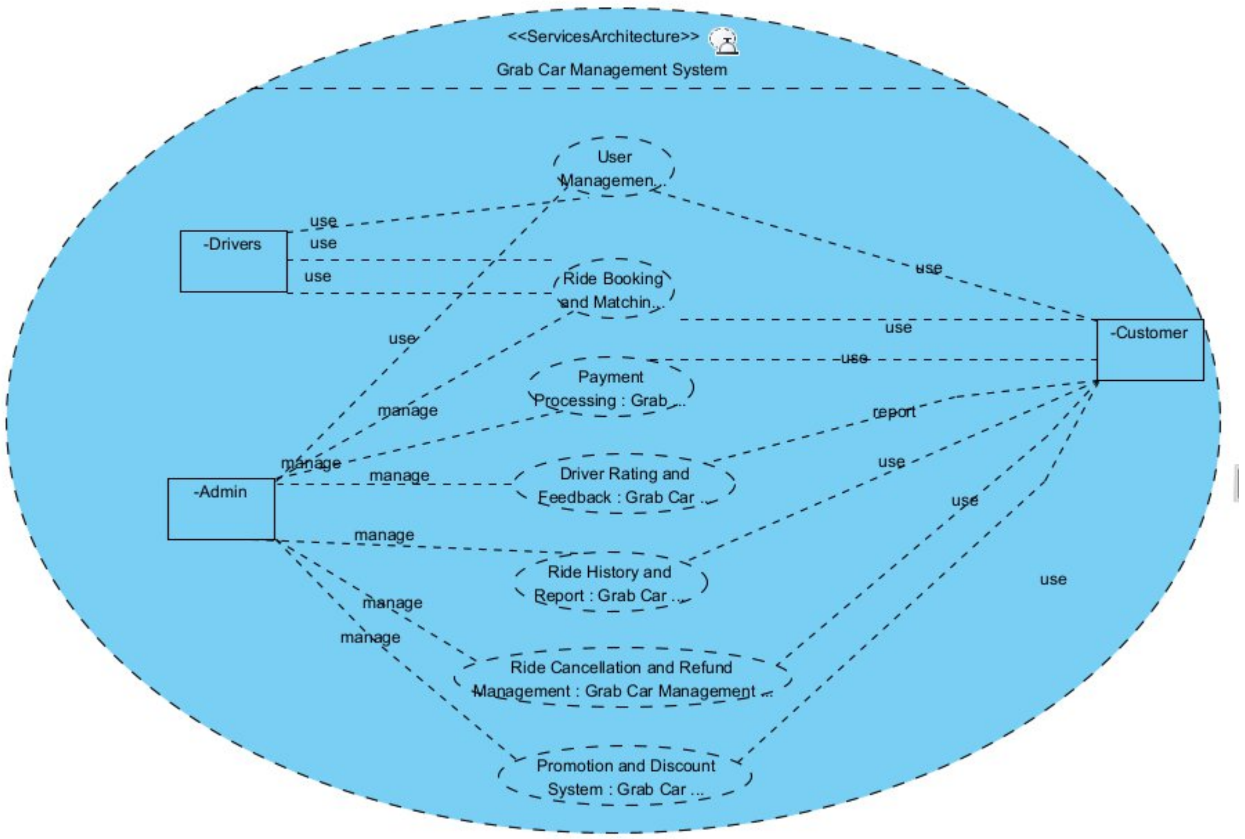
+ User management
+ Course Management
+ Fee Payment and Scholarships
+ Research and publication
+ Enrollment and Register
+ Hostel and Accommodation
+ Career and Internship
+ Exams and Grading



e. Decompose Grab Car Management System

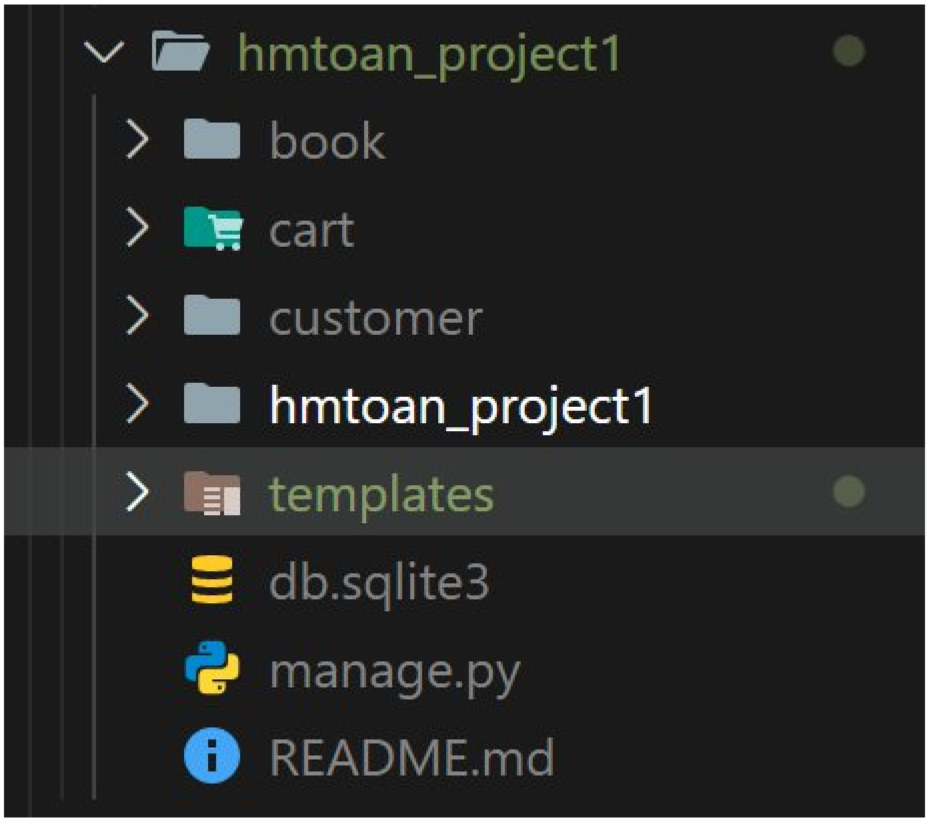
- Users: Drivers, Admin , Customers

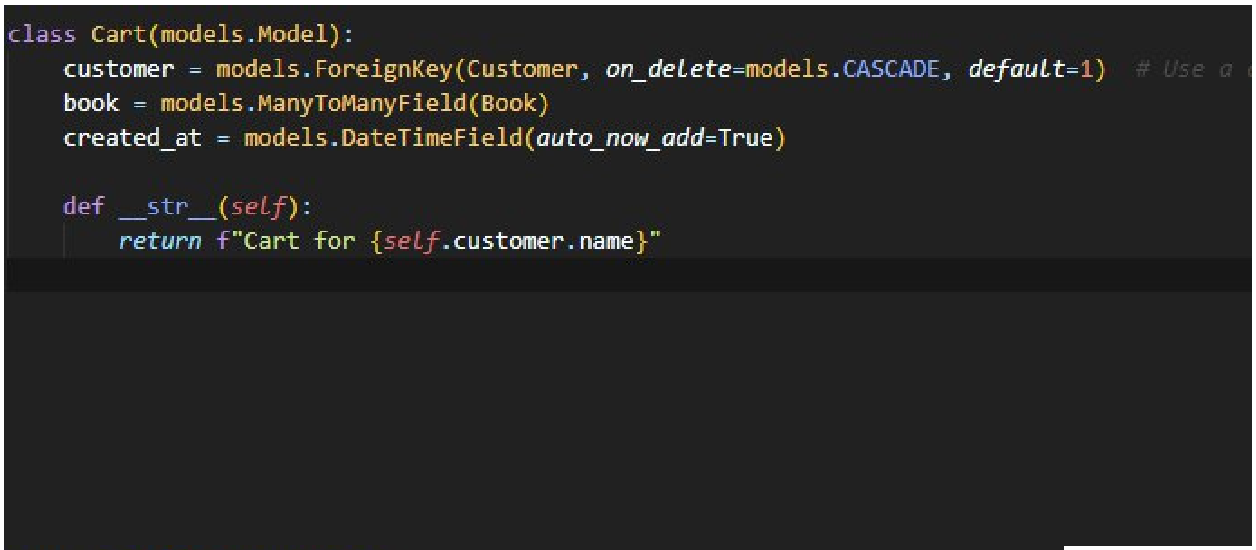
- Services:
+ User Management
+ Ride Booking and Matching
+ Payment Processing
+ Driver and Feedback
+ Ride History and Reports
+ Ride Cancellation and Refund
+ Promotions and Discount



<<ServicesArchitecture>>
Grab Car Management System
User Managemen...
Ride Booking and Matchin...
Payment Processing : Grab ...
Driver Rating and Feedback : Grab Car ...
Ride History and Report : Grab Car ...
Ride Cancellation and Refund Management : Grab Car Management ...
Promotion and Discount System : Grab Car ...
-Drivers
-Admin
-Customer
use
use
use
use
manage
manage
manage
manage
manage
manage
use
use
use
report
use
use
use

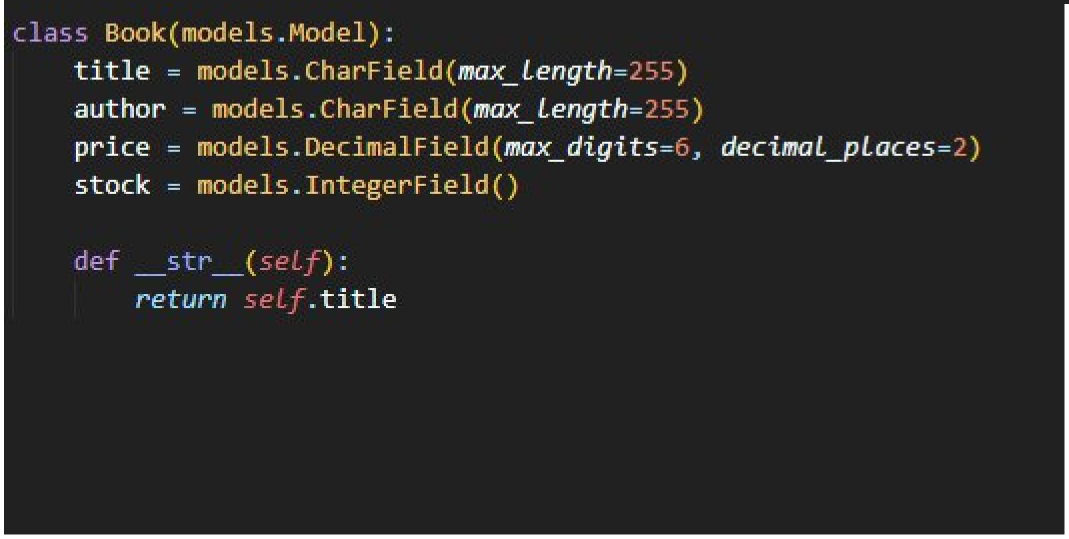
4. Project


hmtoan_project1
book
cart
customer
hmtoan_project1
templates
db.sqlite3
manage.py
README.md



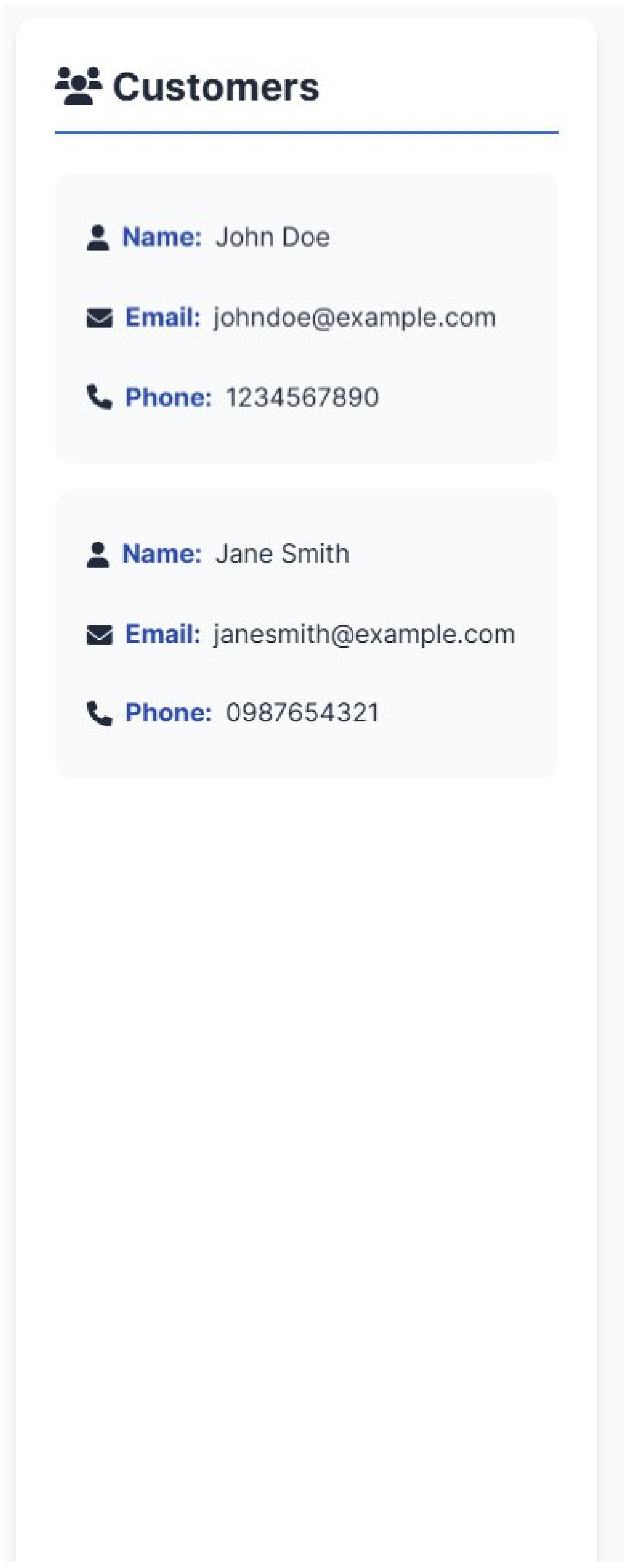
```python
class Cart(models.Model):
    customer = models.ForeignKey(Customer, on_delete=models.CASCADE, default=1)  # Use a
    book = models.ManyToManyField(Book)
    created_at = models.DateTimeField(auto_now_add=True)

    def __str__(self):
        return f"Cart for {self.customer.name}"
```



```python
class Book(models.Model):
    title = models.CharField(max_length=255)
    author = models.CharField(max_length=255)
    price = models.DecimalField(max_digits=6, decimal_places=2)
    stock = models.IntegerField()

    def __str__(self):
        return self.title
```



Customers

Name: John Doe

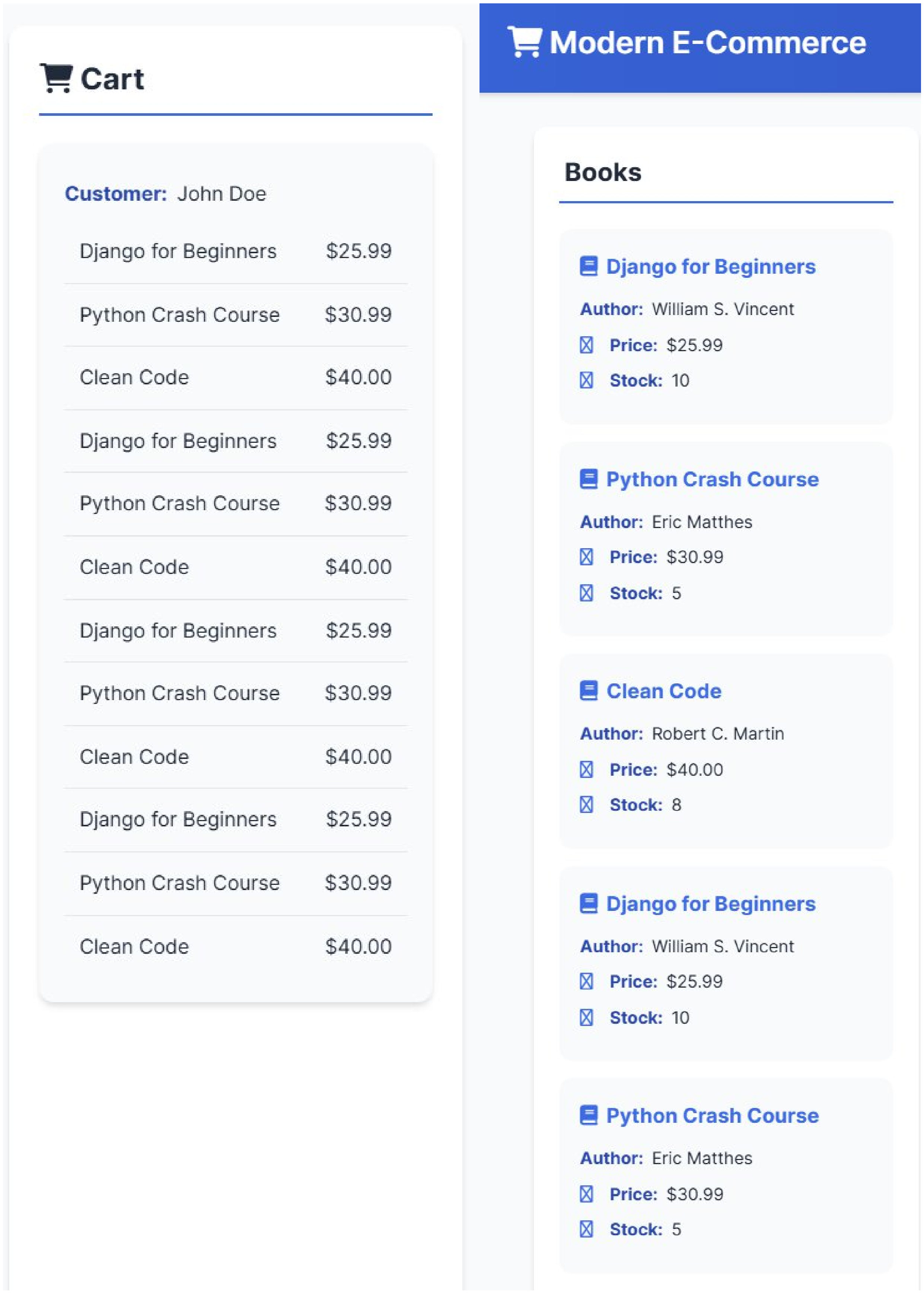
Email: johndoe@example.com

Phone: 1234567890

Name: Jane Smith

Email: janesmith@example.com

Phone: 0987654321



Cart

Customer: John Doe

Django for Beginners	$25.99
Python Crash Course	$30.99
Clean Code	$40.00
Django for Beginners	$25.99
Python Crash Course	$30.99
Clean Code	$40.00
Django for Beginners	$25.99
Python Crash Course	$30.99
Clean Code	$40.00
Django for Beginners	$25.99
Python Crash Course	$30.99
Clean Code	$40.00

Modern E-Commerce

Books

Django for Beginners

Author: William S. Vincent

Price: $25.99

Stock: 10

Python Crash Course

Author: Eric Matthes

Price: $30.99

Stock: 5

Clean Code

Author: Robert C. Martin

Price: $40.00

Stock: 8

Django for Beginners

Author: William S. Vincent

Price: $25.99

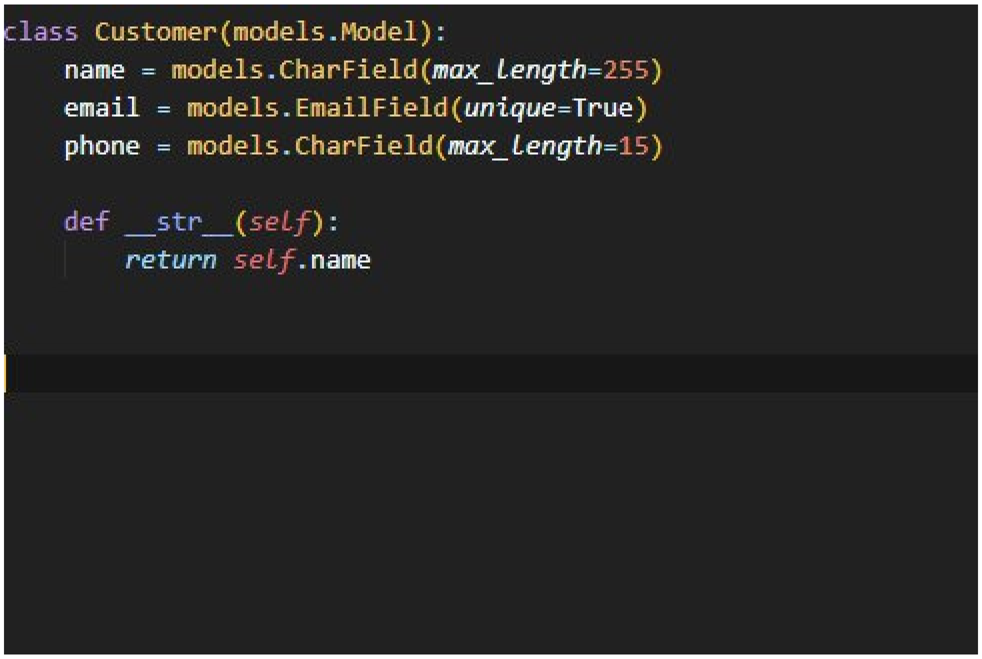
Stock: 10

Python Crash Course

Author: Eric Matthes

Price: $30.99

Stock: 5



```
class Customer(models.Model):
    name = models.CharField(max_length=255)
    email = models.EmailField(unique=True)
    phone = models.CharField(max_length=15)

    def __str__(self):
        return self.name
```