Design Patterns

Design patterns are reusable solutions to commonly occurring software design challenges. They are best practices for solving problems in software design and development. Here are some design patterns:

Singleton: This pattern ensures that a class has only one instance and provides access to that single instance.

Factory pattern: This pattern creates objects without specifying the exact class to create.

Adapter pattern: This pattern allows objects to cooperate and each perform their own unique task by converting the interface of one class into another.

Decorator pattern: This pattern dynamically adds functionality to an object by wrapping it with a different class at runtime.

Observer pattern: This pattern defines a one-to-many dependency between objects, allowing updates to be broadcast to multiple objects.

Command pattern: This pattern encapsulates a request as an object, allowing it to be treated as a standalone unit and providing the ability to undo or redo commands.

Template pattern: This pattern defines the skeleton of an algorithm, allowing subclasses to redefine/override certain steps.

Iterator pattern: This pattern allows sequential access to elements of an object without exposing its underlying implementation.

Proxy pattern: This pattern provides a surrogate or placeholder object to control access to another object, such as to provide security checks.

Facade pattern: This pattern provides a simple interface for a complex subsystem, making it easier to use and understand.

Data Structures

Data structures are ways of organizing and storing data in a computer. Here are some common data structures:

Arrays: A collection of elements of the same type stored in contiguous memory locations.

Linked Lists: A collection of nodes that contain data elements and links to the next node in the sequence.

Stacks: A collection of elements that supports two operations, push and pop, in which elements can be added or removed only from one end.

Queues: A collection of elements that supports two operations, enqueue and dequeue, in which elements can be added at the back and removed from the front.

Trees: A hierarchical data structure that consists of nodes connected by edges, where each node has a parent and zero or more children.

Graphs: A collection of vertices connected by edges.

Hash Tables: A data structure that maps keys to values using a hash function to compute an index into an array of buckets.

Heaps: A complete binary tree data structure that satisfies the heap property, which is that every parent node is either greater than or equal to its children.

Tries: A tree-like data structure used to store and retrieve associative data, such as words in a dictionary.

AVL Trees: A self-balancing binary search tree in which the heights of the left and right subtrees of any node differ by at most one.

Types of Trees

There are several types of trees, each with unique properties and uses. Here are some of the most common types of trees:

Binary Tree: A tree in which each node has at most two children (left and right).

Binary Search Tree (BST): A binary tree in which the left child of each node has a smaller value than the parent node, and the right child has a larger value.

AVL Tree: A self-balancing binary search tree in which the heights of the left and right subtrees of any node differ by at most one.

Red-Black Tree: A self-balancing binary search tree in which the nodes are colored either red or black, and the height of every subtree from a given node has at most a fixed ratio to the height of every other subtree.

B-Tree: A self-balancing tree data structure that can store large amounts of data in each node and allows efficient access to elements.

Trie: A prefix tree that stores strings or sequences by encoding each character as a key in a tree, where the path to a leaf node represents a word or sequence.

Heap: A complete binary tree data structure in which the parent of each node has a higher priority than its children.

Spanning Tree: A subset of edges in a graph that forms a tree and includes all the vertices of the graph.

M-Tree: A metric tree used for similarity search, where the distance between two points can be calculated using a metric function.

Radix Tree: A data structure used for string searches or IP routing, where each node represents a common prefix in a set of strings.

Types of Graphs

There are several types of graphs, each with unique properties and uses. Here are some of the most common types of graphs:

Directed graph: A graph in which the edges have a direction between nodes.

Undirected graph: A graph in which the edges have no direction between nodes.

Weighted graph: A graph in which each edge has a weight or cost assigned to it.

Complete graph: A graph in which every node is directly connected to every other node.

Cycle graph: A graph in which every node is connected to the next and the last node is connected to the first node, forming a cycle.

Bipartite graph: A graph in which the nodes can be divided into two sets such that each edge connects a node from one set to a node from the other set.

Tree: A connected acyclic graph, i.e., a graph with no cycles.

Spanning tree: A subgraph of a graph that connects all the nodes and is also a tree.

Planar graph: A graph that can be drawn in a two-dimensional plane such that no edges cross each other.

Complete bipartite graph: A graph in which the nodes can be divided into two sets and every node in one set is connected to every node in the other set.

what is internet and how it works?

The internet is a global network of connected computers and devices that communicate through a standardized set of protocols. It enables information and data to be sent from one place to another quickly and efficiently.

The internet works by utilizing a system of interconnected networks that use a protocol called TCP/IP (Transmission Control Protocol/Internet Protocol) to communicate with each other. This protocol provides the rules by which data is transmitted and received on the internet.

When a device connects to the internet, it is assigned a unique IP (Internet Protocol) address, which is used to identify and communicate with it on the network. Information is transmitted through a series of routers and switches, which help guide the data to its final destination.

Web browsers, such as Google Chrome or Mozilla Firefox, connect to websites using the HTTP (Hypertext Transfer Protocol) protocol. This protocol allows for webpages to be loaded onto a user's device and displayed in their browser.

Overall, the internet is a complex system of interconnected networks, protocols, and devices that work together to enable fast and efficient communication and data transfer across the globe.

Network Concepts

Network concepts refer to the fundamental principles and components that are used in the design, development, and operation of communication networks. Here are some important network concepts:

Bandwidth: The maximum amount of data that can be transmitted over a communication network in a given time period.

Latency: The amount of time it takes for data to travel from one point to another on a communication network.

Routing: The process of determining the best path for data to travel from one network device to another.

Protocols: A set of rules that govern communication between devices on a network.

Switches: Hardware devices that connect multiple network devices together and direct traffic between them.

Routers: Hardware devices that connect networks together and direct traffic between them.

IP address: A unique numeric identifier assigned to each device on a network that allows it to be addressed and communicated with.

DNS: The domain name system is responsible for converting human-readable domain names to numerical IP addresses so that devices can be accessed on the internet.

Firewall: A network security device that monitors and controls incoming and outgoing network traffic.

Wireless networking: A wireless network uses radio waves to communicate data between devices without the need for physical connections.

Understanding these network concepts can help you design, build, and operate communication networks that are efficient, reliable, and secure.

Programming Concepts

Programming concepts are fundamental ideas that underpin the practice of software development. Understanding these concepts is essential to effectively design and write code. Here are some of the most important programming concepts:

Variables: A variable is a container that stores a value, such as a number or a string.

Data Types: The category or type of data that a variable can store, such as integers, floating-point numbers, strings, booleans, and more.

Control Structures: Control structures change the flow of a program based on certain conditions, such as loops, conditionals, and branches.

Functions: A function is a block of code that performs a specific task, and may return a value.

Object-Oriented Programming: Object-oriented programming is a programming paradigm in which code is organized around objects that have properties (states) and methods (behaviors).

Libraries and Frameworks: These are collections of code that are designed to make common tasks easier, and help developers build software faster.

Algorithms: An algorithm is a set of instructions to perform a specific task or solve a particular problem.

Debugging: Debugging is the process of finding and fixing errors or bugs in code.

Testing: Testing is the process of systematically evaluating code to ensure it meets requirements and functions as expected.

Version Control: Version control is the practice of tracking and managing changes to code over time, usually using Git or another version control system.

Understanding these programming concepts is essential for developing quality software that is efficient, maintainable, and scales well. While there are many more programming concepts to learn, these concepts provide a solid foundation for those beginning their programming journey.

OOP Principles

■■■■■■■■■■■■■

Object-Oriented Programming (OOP) is a programming paradigm that is based on the concept of objects that have properties and methods. There are four key principles of OOP that help developers build software that is scalable, maintainable, and organized. These principles are:

Encapsulation: Encapsulation is the process of hiding the internal state and behavior of an object, and only exposing a certain set of interfaces or public methods that can be used outside of the object. This helps to organize and modularize code, improve security (as it hides sensitive data), and prevent direct modification of data.

Inheritance: Inheritance is the process of creating new classes based on existing ones, called superclasses. This allows code to be reused and provides a way of organizing and sharing functionality between different classes.

Polymorphism: Polymorphism is the ability of objects of different types to be used interchangeably, often through the use of inheritance and interfaces. Polymorphism allows for code to be written that works with a variety of different

objects, without requiring the specific type of the object to be explicitly known.

Abstraction: Abstraction is the practice of simplifying complex systems by breaking them down into smaller pieces, and only exposing the essential features to users. This helps developers to manage code complexity by focusing on high-level design and preventing unnecessary details from cluttering the code.

In summary, the principles of OOP are designed to make code more structured, easier to manage, and more reusable. By following these principles, developers can build high-quality software that is easy to modify, maintain, and scale over time.

Backend Concepts

■■■■■■■■■■■■■■■■■

Backend concepts refer to the fundamental principles of server-side programming for building web applications. Some important backend concepts are:

Server-side programming languages: The programming languages used to build web applications includes PHP, Python, Ruby, Java, and Node.js.

Relational and NoSQL databases: Databases are used to store, retrieve, and manipulate data. Common types of databases include relational databases like MySQL, PostgreSQL, and SQLite, and NoSQL databases like MongoDB, Cassandra, and CouchDB.

Application Programming Interfaces (APIs): APIs are software interface that allow different applications to communicate with each other. They enable backend systems to provide access to data, functionality, or services.

Web servers: It is the infrastructure that handles requests and responses from client/s. Some of the popular web servers are Apache and Nginx.

Cloud computing services: Cloud computing services like Amazon Web Services, Microsoft Azure, and Google Cloud Platform provide infrastructure, platforms, and software as a service that enables companies to access computing services on demand.

Server monitoring and logging: Monitoring server's performance, tracking server logs provide insights to optimize application performance and predict performance glitches.

Security: It is important to have secure backend code which is free of vulnerabilities and protection against exploitation in the form of SQL injection or cross-site scripting (XSS)

In summary, backend concepts include server-side programming languages, databases, APIs, web servers, cloud computing, server monitoring, logging and security. Understanding these concepts is critical for building reliable and scalable backends for web applications.

Django Components

Django is a high-level Python web framework for creating web applications. Here are the key components of Django:

Model-View-Controller (MVC) Architecture: Django uses a variation of the MVC architecture called Model-View-Template (MTV). It separates the application into three layers: models (where data is defined and queried), views (where data is manipulated and rendered), and templates (where the user interface is defined).

URL Routing: Django uses a URL routing system to handle incoming requests and map them to specific views in the application. This provides a clean and organized way to handle different URLs and requests.

Database Access: Django offers a high-level database API that makes it easy to define and interact with databases using Python objects rather than writing SQL.

Admin Panel: The Django admin panel is a built-in application that provides an easy-to-use interface for managing data models and content.

Template Engine: Django provides a template engine that allows developers to define the structure and layout of HTML pages with placeholders for dynamic content.

Authentication and Authorization: Django has a built-in authentication and authorization system that makes it easy to securely authenticate users and control access to certain parts of the application.

Middleware: Middleware is a mechanism provided by Django for processing requests and responses. It allows developers to define reusable components that can add functionality to the application, such as handling caching or logging.

In summary, Django is a powerful web framework that offers many components to simplify and speed up the building of web applications. By using these components effectively, developers can create powerful and scalable web applications rapidly.

What should i know as a Django Backend Developer?

As a Django Backend Developer, you should have a strong understanding of the following:

Python: Python is the programming language used in Django. You should have a solid understanding of Python data structures, functions, and classes.

Django Framework: You should be familiar with Django’s main components and tools, including the ORM, views, templates, URL routing, and middleware.

Database Management: You should be proficient in database management including creating, updating, reading and deleting records using Django’s built-in ORM.

Django REST Framework: You should be familiar with the Django REST Framework to handle RESTful APIs to manage models and data.

Server Management: You should have knowledge of managing web servers such as Apache or Nginx, setting up and managing virtual environments and configuring production environments deployment.

Security: You should have knowledge of cybersecurity techniques, methods, and best practices for handling user input, authenticate and store user data securely.

Testing: You should have experience with testing frameworks such as Pytest or Django Test and understand how to write unit, functional and integration tests.

Git & Version Control: You should have knowledge of version control via Git or other related tools, know best practices for managing source code using Git.

Debugging: You should have an understanding of Django Debug Toolbar, and basic knowledge of debugging strategies and practices.

Industry Standards & Practices: You should have an understanding of industry standards and practices in software development, project management, and agile methodologies.

In summary, as a Django Backend Developer, you need to have a solid understanding of Python, Django framework, database management, Django REST Framework, server management, security, testing, version control, debugging, and industry standards and practices. Also, staying updated with the latest Django version would help you enhance your skills and knowledge.

What should i know as a Django Backend Developer in interview?

As a Django backend developer, there are several important things you should know for a job interview. Here are some key areas:

Django Framework: You should have a strong understanding of the Django framework including models, views, templates, URL routing, middleware, and forms. You should be familiar with the Django ORM and know how to use it to interact with databases.

Python Programming: You should have strong skills in Python programming, including object-oriented programming concepts and advanced features such as decorators, context managers and generators.

Web Development: You should be familiar with web development concepts like HTTP request-response cycle, web servers, server-side programming, API development, AJAX requests, and client-side web development.

Databases: Understanding of SQL Databases, Database Normalization, and optimization is important. Django supports multiple database back ends and its own ORM framework.

Security: You should be knowledgeable about security concepts and best practices in web development, including secure user authentication, password hashing, encryption, authorization, and validation.

Deployment: Ideally you should understand deployment concepts like server architecture, scaling strategies, deployment automation, and version control.

Project Experience: Having experience in developing and deploying a project in Django framework would be a great advantage in the interview.

In general, interviewers may ask you ways on how you have applied these skills in your previous job or project-specific to Django, how you solved technical challenges, and best practices you follow in your coding approach.

<

Git: A distributed version control system that allows multiple developers to collaborate on a codebase and keep track of changes over time.

HTTP: HyperText Transfer Protocol that enables communication between clients and servers on the internet.

API: An Application Programming Interface that defines how different software programs can interact and exchange data with each other.

RESTful API: An API that follows the principles of Representational State Transfer architecture for data exchange and resource state management.

SSL: Secure Sockets Layer is a security protocol that facilitates encrypted communication between client and web server.

Cookie: A small data file that a web server sends to a web browser and stores on a user's device to remember user information.

Session: A mechanism used to maintain a user's state between HTTP requests.

Token: An encrypted string that a client uses to prove its identity to a server.

OAuth: An authentication protocol that allows third-party applications limited access to a user's data on a website without exposing the user's credentials.

Cloud: A network of remote servers that provide storage, processing, and data management services.

CDN: Content Delivery Network is a distributed network of servers used to deliver content faster and more efficiently.

Load Balancing: A technique for distributing traffic across multiple servers to improve performance and reliability.

Cache: A mechanism for storing frequently accessed data in memory to improve performance.

Microservices: An architectural approach to software development where applications are designed as a collection of independent services that communicate with each other via APIs.

Webservices: A software system that provides a standardized way of communication between different software applications over the internet.

Cloud Computing: The delivery of computing services—including servers, storage, databases, networking, software, analytics, and intelligence—over the internet to offer faster innovation, flexible resources, and economies of scale.

Lazy Loading: A technique for loading objects only when they are needed, improving performance and reducing load times.

Eager Loading: A technique for loading all objects at once, improving performance and reducing database access.

Database Transactions: A group of SQL statements treated as a single unit of work, ensuring data integrity and consistency.

Design Patterns: Reusable solutions to common software development problems.

FTP: File Transfer Protocol used to transfer files between client and server.

HTTPS: Hypertext Transfer Protocol Secure is an extension of HTTP, used for secure communication over a computer network.

SSH: Secure Shell is a network protocol that allows secure data communication over insecure networks.

Asynchronous and Synchronous: Synchronous programming has a wait time to get the result, while asynchronous programming does not wait for a response.

How does the internet work?: The internet is a global network of computers interconnected through standard protocols, allowing communication and exchange of information.

DNS: Domain Name System is a protocol that translates domain names into IP addresses.

TLS: Transport Layer Security is a security protocol for secure communication over a network.

CORS: Cross-Origin Resource Sharing is a mechanism to allow accessing resources from different domains.

Encrypting Algorithms: Algorithms used to scramble data into a code that cannot be deciphered without a key.

Hashing Algorithms: An algorithm that generates a unique fixed-size string for the input data.

SHA family, MD5: Examples of hashing algorithms where SHA has different variants named SHA-1, SHA-256, SHA-512 while MD5 generates 128-bit hashes.

Integration Testing: Testing technique used to test the integration between different modules of software.

Unit Testing: Testing technique used to test individual units or components of software.

TDD: Test-Driven Development is a software development methodology where the test cases are written before writing the code.

Functional Testing: Testing technique used to verify the functionality of software components.

CI/CD: Continuous Integration / Continuous Deployment is a software development approach where code is continuously integrated and tested before being deployed to production environments.

Serverless: An architectural approach to computing where developers do not have to manage server infrastructure and can focus on building and deploying their applications.

DRY: Don't Repeat Yourself is a design principle that encourages software developers to avoid duplicating code.

Reusability: The ability to use components of software systems in multiple applications or contexts.

Pluggability: The ability to add or remove components to a software system without changing the existing functionality.

Agile: A software development methodology that emphasizes iterative development and customer collaboration.

UML: Unified Modeling Language is a visual modeling language used for software design.

Scrum: An Agile methodology that emphasizes collaboration, adaptive planning, and iterative delivery.

GPG: GNU Privacy Guard is a free software used for secure communication and data encryption.

CMS: Content Management System is software used to create, manage, and distribute digital content.

SOAP: Simple Object Access Protocol is a messaging protocol used for exchanging structured information between applications.

JWT: JSON Web Token is a type of token used for authentication and authorization between two parties.

TCP/IP: Transmission Control Protocol / Internet Protocol is a set of communication protocols used for sending and receiving data on the internet.

PWA: Progressive Web Apps are web applications that use modern web technologies to offer a fast, engaging user experience.

SPA: Single Page Application is a web application that loads a single HTML page and dynamically updates the page in response to user interactions.

WSGI: Web Server Gateway Interface is a specification for communication between web servers and web applications.

ASGI: Asynchronous Server Gateway Interface is a specification for communication between web servers and asynchronous web applications.

Asynchronous and Synchronous

■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■

• asynchronous and synchronous are two modes of communication or execution. Asynchronous means that the sender and receiver do not have to wait for each other, while synchronous means that they do.
• For example, when you send an email to someone, you don’t have to wait for them to reply before you can do something else. This is asynchronous communication. But when you make a phone call to someone, you have to wait for them to answer and talk to them in real time. This is synchronous communication.
• Similarly, when you write a program that performs some tasks, you can choose whether to make them asynchronous or synchronous. Asynchronous tasks can run in the background without blocking the main thread of execution, while synchronous tasks have to finish before the next task can start.
• Asynchronous programming can improve performance and responsiveness of your application, but it can also introduce complexity and challenges such as handling errors, callbacks, promises, async/await syntax etc.
• Synchronous programming can be simpler and easier to understand and debug, but it can also cause delays and bottlenecks if some tasks take too long or depend on external factors such as network latency etc.
>
<
How does internet work?

■■■■■■■■■■■■■■■■■■■■■■■

• The internet is a global network of computers that can

communicate with each other by sending and receiving data1. The data is divided into small segments called packets, which contain both information and instructions for routing them to the correct destination23. The internet uses a set of rules called protocols to ensure that packets are delivered correctly and efficiently. One of the most important protocols is Internet Protocol (IP), which assigns a unique address to each computer on the internet45.

• There are many other protocols besides IP that enable different types of communication and functionality on the internet. Some of them are:

• TCP: Transmission Control Protocol ensures reliable and ordered delivery of data across IP connections1 .

• UDP: User Datagram Protocol enables low-latency and non-reliable data transmission for time-sensitive applications like voice or video12.

• FTP: File Transfer Protocol allows users to transfer files between computers on a network.

• HTTP: Hypertext Transfer Protocol defines how web browsers and servers exchange data over the web.

>

Software Engineering Principles

Software engineering principles are a set of best practices and techniques used to develop high-quality software products that meet the requirements of the customer or end-user. They include:

Modularity: Modularity is the practice of designing software as a set of independent, interchangeable modules, which can be developed and tested separately.

Abstraction: Abstraction is the practice of hiding complex implementation details and providing simple, high-level interfaces to the functionality of the software.

Encapsulation: Encapsulation is the practice of grouping related data and procedures into classes or objects, which encapsulate the behavior of the software.

Separation of Concerns: Separation of concerns is the practice of dividing software into separate components that each address a single concern or responsibility.

Reusability: Reusability is the practice of developing software components that can be reused in other projects or programs.

Testability: Testability is the practice of designing software for ease of testing, which is essential in ensuring the correctness and quality of the software.

Maintainability: Maintainability is the practice of designing software that is easy to maintain and modify, which is essential in ensuring the longevity and evolution of the software.

Scalability: Scalability is the practice of designing software that can handle increasing numbers of users, data, and transactions without requiring major changes to the architecture.

Security: Security is the practice of designing software that is secure against malicious attacks or unauthorized access.

Usability: Usability is the practice of designing software that is easy to use and understand by the end-users.

These principles help software engineers to develop software that is reliable, efficient, maintainable, and easy to extend and modify over time.

The book "Design Patterns"

■■■■■■■■■■■■■■■■■■■■■■■■

The book "Design Patterns: Elements of Reusable Object-Oriented Software" by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides is a classic in the field of software engineering. It consists of 23 design patterns, which are commonly used solutions to recurring design problems. Each pattern in the book is presented in the same format, using a common vocabulary and presentation style.

In summary, the book can be broken down into the following sections:

Section 1: Introduction

The introduction provides an overview of the concept of design patterns and their importance in software engineering. It introduces the four authors and their backgrounds in software development. The authors then present the Gang of Four (GoF) book, which is a collection of 23 design patterns that they have identified as being particularly useful for object-oriented software development.

Section 2: Design Patterns

This section provides a comprehensive overview of each of the 23 design patterns in the book. Each pattern is presented in the same format, including a brief description, a list of the problems it solves, a list of the solutions it provides, implementation details, and examples.

The patterns are divided into three categories:

Creational Patterns

These patterns deal with creating objects in a way that is flexible and efficient, allowing for the creation of complex objects from simpler ones.

Structural Patterns

These patterns deal with the composition of classes and objects, allowing the creation of more complex structures from simpler ones.

Behavioral Patterns

These patterns deal with the communication between objects, allowing for more flexible and dynamic interactions between different parts of a system.

Section 3: Conclusion

The final section of the book provides a conclusion and some final thoughts on design patterns. It discusses how design patterns can be adapted and modified to suit different programming languages and environments. It also emphasizes the importance of using design patterns in software development and how they can help developers create more flexible and efficient systems.

In conclusion, the book "Design Patterns: Elements of Reusable Object-Oriented Software" is an important resource for software developers who want to learn about common design patterns and their practical applications. Each pattern is presented in a clear and easily understood format, with examples and implementation details provided. While the book is relatively long, it is a valuable resource for anyone who is serious about software development.

The book was originally published in 1994 and has become a classic in the world of software engineering. It is often cited as a foundational text in the field and has been translated into dozens of languages.

The authors of the book, Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, are often referred to as the "Gang of Four" (GoF). They are prominent figures in the software development community and have made significant contributions to the field over the years.

The book presents 23 design patterns that are commonly used in object-oriented software development. Each pattern is presented in a clear and concise format, with examples and implementation details provided. The patterns are divided into categories based on their type (creational, structural, or behavioral).

The book teaches not only the specific patterns but also the concepts behind them. As such, it is designed to help developers understand how to design software systems that are flexible, reusable, and maintainable.

The patterns presented in the book are not specific to any one programming language or platform. Instead, they are presented in a language-agnostic way, making them applicable to a wide range of software development contexts.

While the book can be dense and technical at times, it is also considered to be quite readable and accessible. It is often used as a textbook in university courses and is recommended reading for anyone looking to become a skilled software developer.

Overall, the book "Design Patterns: Elements of Reusable Object-Oriented Software" is an important resource for anyone interested in software development. It is widely considered to be a classic and is a valuable tool for anyone looking to improve their skills as a developer.

The book emphasizes the importance of using design patterns to promote code reuse and maintainability. By using established patterns, developers can create software that is more modular and easier to maintain, even as it grows in complexity.

The patterns presented in the book are grounded in object-oriented programming concepts such as encapsulation, inheritance, and polymorphism. The authors use these concepts to explain how different patterns can be used to solve different types of design problems.

In addition to the 23 design patterns presented in the book, the authors also cover various other concepts related to software design, such as anti-patterns, design principles, and design trade-offs.

The book has had a significant impact on the software industry, and many of the patterns presented in it have become widely used in real-world software projects. Some of the patterns, such as the Singleton and Observer patterns, have even become part of the common vocabulary of software developers.

The book is not just theoretical; it includes real-world examples of how design patterns have been applied in practice. These examples help to illustrate the practical applications of the patterns and how they can be used to solve specific design problems.

Finally, the book encourages developers to think critically and creatively about software design. The authors stress that design patterns are not a one-size-fits-all solution, and that developers must be willing to adapt and modify patterns to suit their specific needs. By teaching developers how to think about software design, the book provides a lasting legacy that continues to influence the field of software engineering today.

PEP 8

■■■■■

PEP 8 (Python Enhancement Proposal 8) is a style guide for writing Python code. It outlines guidelines and best practices for code formatting, organization, and naming conventions to improve readability and consistency across a codebase. PEP

8 covers topics such as indentation, line length, variable naming, function and class naming, imports, and more. Adhering to PEP 8 standards is important to ensure code quality and make code more maintainable and easier to collaborate on.

Here are some additional details about PEP 8:

Indentation: PEP 8 recommends using 4 spaces for indentation and avoiding the use of tabs.

Line Length: Lines should not exceed 79 characters. In cases where the limit is exceeded, the line should be broken up into logical segments.

Naming Conventions: Variables and functions should be named using lowercase words separated by underscores. Class names should be in TitleCase. Constants should be in ALL_CAPS.

Imports: Imports should be listed at the top of the file and grouped in a specific order (standard library, third-party libraries, and local imports). You should avoid using wildcard imports.

Function and Class definitions: Use empty lines between function and class definitions. The first line of documentation should be a short summary of the function or class.

Comments: Comments should be used sparingly and only to explain complicated sections of code. They should be written in complete sentences and avoid unnecessary fluff.

Overall, adhering to PEP 8 guidelines helps ensure consistency and readability of Python code, which improves its maintainability, reduces errors, and makes it easier to collaborate with other developers.

Data Structures

■■■■■■■■■■■■■■

Array

Linked List

Stack

Queue

Binary Tree

Heap

Hash Table

Set

Map

Graph

Array: An array is a collection of elements of the same data type, stored in contiguous memory locations. Elements in an array can be accessed using an index, and arrays can be used to store and manipulate large amounts of data efficiently.

Linked List: A linked list is a collection of nodes, where each node contains a value and a reference to the next node in the list. Linked lists can be used to implement dynamic structures, such as stacks, queues, and hash tables.

Stack: A stack is a collection of elements that follows the Last-In-First-Out (LIFO) principle. Elements are added and removed from the top of the stack, and the most recently added element is the first to be removed.

Queue: A queue is a collection of elements that follows the First-In-First-Out (FIFO) principle. Elements are added to the back of the queue and removed from the front, and the oldest element is always the first to be removed.

Binary Tree: A binary tree is a tree data structure where each node has at most two children, known as the left and right child. Binary trees are commonly used in search algorithms and data compression.

Heap: A heap is a special type of binary tree that satisfies the heap property, which states that the parent nodes are always greater or less than the child nodes. Heap data structures can be used to efficiently find the minimum or maximum element in a collection.

Hash Table: A hash table is a data structure that uses a hash function to map keys to values in a table. Hash tables provide fast access to elements and are commonly used in database indexing and caching.

Set: A set is a collection of unique elements, where the order of the elements is not significant. Sets can be used to efficiently check for membership and perform operations such as union, intersection, and difference.

Map: A map is a collection of key-value pairs, where each key is unique. Maps can be used to efficiently look up values based on their associated keys.

Graph: A graph is a collection of nodes and edges that can be used to represent complex relationships between objects. Graphs can be used to model social networks, transportation systems, and many other real-world scenarios.

Tree

■■■■

Binary Tree: A binary tree is a tree data structure where each node has at most two children, known as the left and right child.

Binary Search Tree: A binary search tree is a binary tree where the elements are stored in a way that enables fast search, insertion, and deletion operations. In a binary search tree, the left child of a node contains elements that are smaller than the parent node, and the right child contains elements that are greater.

AVL Tree: An AVL tree is a binary search tree that is balanced, meaning that the height of the left and right subtrees of any node differ by at most one. AVL trees are often used in database indexing.

Red-Black Tree: A red-black tree is a binary search tree that is self-balancing, meaning that the height of the tree is always logarithmic in the number of elements stored. Red-black trees are commonly used in database indexing and memory allocation.

B-Tree: A B-tree is a tree data structure that is used to store large amounts of data on disk or in memory. B-trees are optimized for read and write operations and are commonly used in file systems and databases.

Trie: A trie is a tree data structure that is used to store strings or sequences of elements. Tries are optimized for prefix search operations and are commonly used in search engines and spell checkers.

Heap: A heap is a tree data structure that is used to maintain a partially ordered set. Heaps are commonly used in sorting algorithms and priority queues.

Graph

Undirected Graph: An undirected graph is a graph where edges do not have a direction, meaning that they can be traversed in either direction.

Directed Graph: A directed graph is a graph where edges have a direction, meaning that they can only be traversed in one direction.

Weighted Graph: A weighted graph is a graph where edges have a weight or cost associated with them. Weighted graphs are commonly used in optimization and routing problems.

Bipartite Graph: A bipartite graph is a graph where the nodes can be divided into two disjoint sets, with edges only connecting nodes from different sets. Bipartite graphs are commonly used in matching problems.

Complete Graph: A complete graph is a graph where every pair of nodes is connected by an edge. Complete graphs are commonly used in network topology and graph theory.

Cyclic Graph: A cyclic graph is a graph that contains at least one cycle, which is a path that starts and ends at the same node.

Acyclic Graph: An acyclic graph is a graph that does not contain any cycles. Acyclic graphs are commonly used in directed acyclic graph (DAG) algorithms, such as topological sorting and shortest path algorithms.

Planar Graph: A planar graph is a graph that can be drawn in the plane without any edges crossing. Planar graphs are commonly used in graph theory and network topology.

Spanning Tree: A spanning tree is a subgraph of a connected graph that includes all of the nodes of the original graph, but contains no cycles. Spanning trees are commonly used in network design and optimization.

Search Algorithms

[redacted]

of data. Here are some common search algorithms:

Linear Search: Linear search, also known as sequential search, checks each element in a collection of data until it finds the target element or reaches the end of the collection. Linear search has a time complexity of O(n) in the worst case scenario.

Binary Search: Binary search is a more efficient search algorithm for sorted collections. It works by repeatedly dividing the collection in half until the target element is found or determined to be not present. Binary search has a time complexity of O(log n) in the worst case scenario.

Depth-First Search (DFS): DFS is a graph traversal algorithm that explores the deepest branches of a graph or tree first. DFS can be used for finding a path between two nodes, testing if a graph is connected, and more. DFS has a time complexity of O(V+E) where V is the number of vertices and E is the number of edges in the graph.

Breadth-First Search (BFS): BFS is a graph traversal algorithm that explores all the neighboring nodes at the current depth before moving on to the next depth level. BFS can be used for finding the shortest path in an unweighted graph, testing if a graph is bipartite, and more. BFS has a time complexity of O(V+E) where V is the number of vertices and E is the number of edges in the graph.

A* Search: A* search is a heuristic search algorithm that combines the advantages of both BFS and DFS by using a heuristic function to guide the search process. A* search is commonly used in pathfinding problems, such as finding the shortest path in a maze or in a transportation network.

Sort Algorithms

[redacted]

Bubble Sort: Bubble sort is a simple sorting algorithm that repeatedly swaps adjacent elements if they are in the wrong order. Bubble sort has a time complexity of O(n^2) in the worst case scenario.

Insertion Sort: Insertion sort is a simple sorting algorithm that builds a final sorted array one element at a time. It works by inserting each new element in the correct position in the sorted array. Insertion sort has a time complexity of O(n^2) in the worst case scenario.

Selection Sort: Selection sort is a simple sorting algorithm that repeatedly selects the smallest element from the unsorted part of the array and places it at

the beginning of the sorted part of the array. Selection sort has a time complexity of O(n^2) in the worst case scenario.

Merge Sort: Merge sort is a divide-and-conquer sorting algorithm that recursively divides the array into two halves, sorts each half, and then merges the two sorted halves back together. Merge sort has a time complexity of O(n log n) in the worst case scenario.

Quick Sort: Quick sort is a divide-and-conquer sorting algorithm that partitions the array into two parts, with all elements less than a chosen pivot element on one side and all elements greater on the other side. Quick sort then recursively sorts each partition. Quick sort has a time complexity of O(n log n) in the average case scenario, but can have a worst case time complexity of O(n^2).

Heap Sort: Heap sort is a comparison-based sorting algorithm that uses a binary heap data structure to sort elements. Heap sort has a time complexity of O(n log n) in the worst case scenario.

■■■■■■■■■■■

چطور میتوانم خودم را برای مصاحبه شغلی بک اند جنگو آماده کنم؟

برای آماده شدن برای مصاحبه شغلی بک اند جنگو، می‌توانید به موارد زیر توجه کنید:

تحقیق درباره شرکت و پروژه‌های قبلی آن

آشنایی با زبان‌های برنامه‌نویسی مورد استفاده در پروژه

تسلط به فریم‌ورک جنگو

تسلط به نحوه پیاده‌سازی و استفاده از دیتابیس‌ها

تسلط به نحوه استفاده از Git و GitHub

تسلط به نحوه حل مشکلات و رفع باگ‌ها

با توجه به اینکه هر شرکت و هر مصاحبه شغلی خاص خودش را دارد، بهتر است قبل از مصاحبه، با شرکت تماس گرفته و سوالات مورد نظر را پرسید.

Here are some concepts you should know to get a job as a Django developer[2]:

- Python programming language
- Object-Oriented Programming (OOP)
- Model-View-Controller (MVC) architecture
- Django framework
- Django ORM (Object Relational Mapping)
- Django REST framework
- Git and GitHub

In addition to these concepts, you should also be familiar with software development principles, JavaScript, SQL and have experience in any programming language[1].

Software development principles are a set of guidelines that help developers create high-quality software. These principles include[1]:

- Architecture first approach
- Agile methodology
- Model-based evolution
- Continuous integration and delivery
- Test-driven development (TDD)
- Code review
- Refactoring

[redacted]

These principles help developers create software that is maintainable, scalable, and easy to understand. They also help ensure that the software is delivered on time and within budget.

- Architecture first approach: This approach focuses on building a strong architecture for software development. It involves creating a blueprint of the software before writing any code.
- Agile methodology: This methodology emphasizes collaboration and flexibility. It involves breaking down the development process into smaller, more manageable chunks called sprints.
- Model-based evolution: This principle supports the evolution of graphics and textual notions. It involves creating a model-based approach to software development.
- Continuous integration and delivery: This principle involves integrating code changes into a shared repository frequently. It also involves automating the build and testing process to ensure that the software is always ready for deployment.
- Test-driven development (TDD): This principle involves writing tests before writing any code. It helps ensure that the code is working as expected and that any changes made to the code do not break existing functionality.
- Code review: This principle involves reviewing code changes made by other developers. It helps ensure that the code is maintainable, scalable, and easy to understand.
- Refactoring: This principle involves improving the design of existing code without changing its functionality. It helps ensure that the code is maintainable and easy to understand.

Design Patterns:

[redacted]

Singleton,

Factory,

Adapter,

Decorator,

Observer,

Command,

Template,

Iterator,

Proxy,

Façade

difference between Compiler and Interpreter

A **compiler** takes the entire program written in a high-level language and translates it into machine code as a whole. The compiled code runs faster compared to interpreted code[1].

On the other hand, an **interpreter** translates the program one statement at a time. Programs written in an interpreted language are easier to debug and are more flexible than a compiled language. However, interpreted code runs slower compared to compiled code[1].

Cloud

The term "cloud" refers to servers that are accessed over the Internet, and the software and databases that run on those servers. Cloud servers are located in data centers all over the world. By using cloud computing, users and companies do not have to manage physical servers themselves or run software applications on their own machines[1].

The cloud enables users to access the same files and applications from almost any device because the computing and storage take place on servers in a data center, instead of locally on the user device. This is why you can log into your email or social media account on a new device and still find your old account in place with all your messages and contacts[1].

There are three main types of cloud computing: **public cloud**, **private cloud**, and **hybrid cloud**. Within these deployment models, there are four main services: **infrastructure as a service (IaaS)**, **platform as a service (PaaS)**, **software as a service (SaaS)**, and **serverless computing**[3].

- A **public cloud** shares resources and offers services to the public over the Internet.

- A **private cloud** isn't shared and offers services over a private internal network typically hosted on-premises.

- A **hybrid cloud** shares services between public and private clouds depending on their purpose².

<

https://docs.python.org/3/library/

https://docs.djangoproject.com/en/3.2/topics/

https://www.django-rest-framework.org/

Tutorial

API Guide

Topics

>

■ ■ ■ ■ ■ ■ ■ ■ ■ ■

دیزاین پترن ها:::

در کتاب دیزاین پترن، 23 تا آبجکت اورینتید دیزاین پترن معرفی شد در سه دسته (Creational Patterns, Structural Patterns, Behavioral Patterns)

■Creational Pattern, The Factory ■

کاری به چیزی نداری، فقط میخوای که یک کلاس/فکتوری/چیز رو صدا بزنی و اون برات بسازه(مثلا کلاس رو)، مثلا یک کلاس هست که چندین متد داره و هرکدوم یک مدل خاصش رو میسازن(مثلا یک کلاسی رو).

■Creational Pattern, The Builder ■

فکر کن یک کلاس بیلدر داری که کلاس میسازه و متد هاش اون کلاس رو تغییر میدن و در نهایت یک متد بیلد داره که اون کلاس نهایی شده رو برمیگردونه. و اینکه هرکدوم از اون متد ها، کلاس رو میگیره و تغییر رو روش اعمال میکنه و برش میگردونه.

■Creational Pattern, The Singleton ■

Singleton is just a class that can only have a single instance of it that’s instantiated

یک کلاسی که فقط میتونه یکبار ساخته/اینیشیِیت بشه و ساخت کلاس توسط یک استاتیک متد انجام میشه و این متد برای ساخت یک نمونه/اینستنس جدید چک میکنه و شرط/if داره که اگر ساخته نشده، بسازه و در نهایت اینستنس رو برمیگردونه(این اینستنس رو تو خودش ذخیره داره که هر بار که میگی یه نمونه بساز اونو برگردونه).

■Behavioral Pattern, Observer ■

میتونیم مثلا بهش بگیم PubSub پترن، برای درک بهتر.

مثلا یک سابجکت/پابلیشر داریم و چند آبزرور/سابسکرایبر که اون میفرسته و اینا میگیرن.

مثلا یک کلاس داریم که یک لیستی از سابسکرایبر ها داره و یک متد سابسکرایب داره که با اون بهش سابسکرایب میکنی و یم متد نوتیفای داره که به همه اعضای لیست سابسکرایبرها یک چیزی رو میفرسته.

مثلا ما تو این جا اگر در پایتون باشیم، لیست سابسکرایبر ها میشه یک لیست از آبجت های اعضا که میتونیم یک متدشونو فرابخونیم و مقداری رو هم بهش ارسال کنیم.

■Behavioral Pattern, Iterator ■

تعیین میکنه که چجوری میشه بین آبجکت های یک کلاس ایتریت کرد و دونه دونه دیدشون، به چه صورت، به چه ترتیب، با چه ترتیب و الگوریتمی.

متد های __iter__ و __next__ در پایتون به ما این امکان رو میدن که روی یک کلاس ایتریت کنیم و حلقه بزنیم.

■Behavioral Pattern, Strategy ■

If you want to modify or extend the behavior of a class without directly changing it

مثلا چند کلاس(فیلتر کلاس ها) داریم که از کلاس خودساخته-ابسترکت-فیلتر ارث میبرن و هرکدوم هم متدِ آیا-حذف-بشود رو به شیوه خودشون پیاده سازی میکنن، حالا یک کلاس داریم تعدادی مقدار داره و یک متد فیلتر داره که این متد یک لیستی یا یکی از اون (فیلتر کلاس ها)رو میگیره و مقدار رو میده به متد اون (فیلتر کلاس) و اگر پس شد، ادش میکنه.

■Structural Pattern, Adapter ■

فکر کن یک پیچ داری که توی مهره جا نمیشه، یک واشری/چیزی/ادپتری بهش میچسبونی تا فیت/درست/دلخواه بشه.

مثلا یک کلاس پایتونی داری که میخوای مثل آداپتور بین دو کلاس باشه و ارتباط برقرار کنه یا تغییر بده یا اضافه کنه، مثلا از یک کلاسی ارث میبره و یک نمونه/اینستنس از یک کلاس دیگه دریافت میکنه و این دریافتی رو رفتارش رو شبیه اونی میکنه که ازش ارث برد.

■Structural Pattern, Facade ■

The word Facade meaning, an outward appearance that is used to conceal a less pleasant or credible reality

بخش ها و قسمت ها و ساختار و سیستمِ لو لول و سطح پایین رو خیلی انتزاعی میکنه و یجورایی مثل یه اینترفیس/API میشه.

یک ظاهری رو فراهم میکنی که با اون خیلی تمیز تر و ساده تر کارهارو انجام بدی، مثلا لیست که توی پایتون همه چیز انتزاعی شده و کاری به چگونه کارکردش نداری چون انجام شده و یک ظاهر به ما داده شده که با اون کار کنیم و نگران مموری و اینها و پیاده سازی یک لیست توی رم و غیره نباشیم و استفاده کنیم.

■ ■ ■ ■ ■ ■ ■ ■ ■ ■

GIT

===

git clone ADDRESS

```
la = ls -la
# Header
## Sub-Header
git status
git add file1 file2
git add .
git commit -m "Message"
git commit -m "Subject-Message" -m "Description"
{
    ssh-keygen -t rsa -b 4096 -C "EMAIL"
    ...
}
git push origin master
git init
git remote add NAME ADDRESS
git remote
git remote -v
            verbose
{
    git push -u origin master
              -u = --set-upstream
    az in be bad faghat bezan 'git push'
}
git branch
git checkout EXIST-BRANCH-NAME
git checkout -b NEW-BRANCH-NAME
git diff BRANCH-NAME
git merge BRANCH-NAME
git branch -d BRANCH-NAME
            delete
git commit -am "Message"
            add, it's only for 'modified' files not 'created' files.
            git add + git commit
git reset
    soft
```

```
    reset to Un-Staged
git reset FILE-NAME
git reset HEAD~1
          one previous commit
git reset COMMIT-ID
git reset --hard COMMIT-ID
            all changes will be Un-Staged and Removed
{
    changes = Un-Staged
    add = Staged
    commit = commit
}
git log
========
restore
git config --global user.name "NAME"
                        .email
                        .editor "code --wait"
git config --global core.autocrlf input
                                  for linux and mac
                                  true
                                  for windows
git config --global -e
                    open the config file in the default editor
git COMMAND --help
git COMMAND -h
            gives a short summery of this command
git help COMMAND
    opens a browser to the document/COMMAND page
git restore --staged FILE-NAME
ls -a
'git rm' is just like the 'rm' command in linux.
git add PATTERN
echo TEXT >> FILE.EXT
          append
```

```
git rm --cached FILE-NAME

       to unstage

git mv FILE-NAME-1 FILE-NAME-2

git rm FILE-NAME

.gitignore

git ls-files

    same as 'git status' but listed and clean

*{

    vaghti yek file/folder ra add+commit mikoni bad mizarish to .gitignore file ama taghirat
track mishe, che baiad kard:

    git rm --cached FILE

    git rm -r --cached FOLDER

}

git status -s

            short

git diff --staged

         chia gharare commit beshan ke add shodan, che taghirati

git diff

    chia gharare add/stage beshan, che taghirati

setting git diff configs:

    git config --global diff.tool vscode

    git config --global difftool.vscode.cmd = "code --wait --diff $LOCAL $REMOTE"

'git difftool' bejaye 'git diff'

git log --oneline

          on line description

        --reverse

          in reverse order

git show COMMIT-ID

git show HEAD

git show HEAD~1

git show COMMIT-ID:FILE-OR-FOLDER

                  for see the changes on a special file/folder/path

git ls-tree COMMIT-ID

    file and folders at that COMMIT-ID

git reset --soft COMMIT-ID
```

```
          --hard
undo the 'add' operation (Un-Stage):
    git restore --staged FILE
git clean -fd
           f: force
           d: directories
           for discarding local changes; to remove all new un-tracked file/folder
git restore FILE
    get the previous version of that FILE
restore a file to the previous version:
    git restore --source=HEAD~1 FILE
====================================
DOCKER
======
when you run an image, then a container will be created-running.

docker run ubuntu COMMAND INPUT-OPTIONS-FOR-THE-COMMAND
docker run ubuntu echo helllllllllo

execute a command on my docker container:
    docker exec CONTAINER-IMAGE-NAME COMMAND INPUT-OPTIONS-FOR-THE-COMMAND
================================
balaye docker compose bayad version ra moshakhas konid:
"
version: 2
services:
    redis:
        image: redis
"
==============================
vaghti az version 'alpine' estefade mikoni dige 'bash' nadari chon 'alpine' kochike vali
bejash 'sh' ke hamon shell-e default hast ro dari.
    docker run -it IMAGE-NAME bash
    for 'alpine' :
        docker run -it IMAGE-NAME sh
```

hazfe tamame container haye stop shode:

1:

docker container prune

hazfe tamame image haye bedardnakhor va bi nam:

2:

docker image prune

docker run -p bironi:daroni

docker run -p (HOST)3000:(CONTAINER)5000

docker run -p 3000:5000

Execute a command in a running container:

docker exec CONTAINER-ID ANY-LINUX-OR-WINDOWS-COMMANDS

docker exec -it CONTAINER-ID ANY-LINUX-OR-WINDOWS-COMMANDS

docker exec -it CONTAINER-ID sh

docker exec -it CONTAINER-ID bash

'docker start' vs 'docker run' :

docker start :

    start mikone yek container-e stop shode

docker run :

    yek container jadid az 'IMAGE'-e delkhah ejra mikone

VOLUMES:

docker volume create VOLUME-NAME

docker run -v VOLUME-NAME:/app/a-name-for-directory IMAGE-NAME

docker run -v NEW-VOLUME:/app/a-name-for-directory IMAGE-NAME

NOTE: you have to mkdir that directory in Dockerfile with 'RUN mkdir dir-name' to prevent any permission issues.

copy from a container to host(folder):

docker cp CONTAINER-ID:/app/path/file.txt .

in noghte yani poshe ee ke Dockerfile toshe

from host to container:

```
docker cp file.txt CONTAINER-ID:/app/path
docker cp ./file.txt CONTAINER-ID:/app

bind/binding/link-using-volumes:
When we want to link a path/folder in host(where project and/or Dockerfile is) to a container:
    docker run -v $(pwd):/app IMAGE-NAME
    docker run -v .:/app IMAGE-NAME
==================================
docker compose up
    # manande 'run' ast
# docker compose baraye kar ba multi container ast
docker compose up --build
docker compose up -d
                   detach
docker compose ps
docker compose down

going in with a special user:
docker exec -it -u root 243hfg9gfd sh
docker exec -it -u root 243hfg9gfd bash
docker exec -it -u amir 243hfg9gfd sh

docker compose logs
docker compose logs 2j3h44hg23
docker compose logs 2j3h44hg23 -f
                               follow
```

■ ■ ■ ■ ■ ■ ■ ■ ■ ■

مفاهیم agile, scrum, sprint, daily, ...

در حالی که Scrum یک روش دقیق Agile است که برای تسهیل یک پروژه استفاده می شود.