System Design Preparation

1. High-Level Design (HLD)

Key Areas to Focus On

1. **Scalability and Performance**:
    - Horizontal vs. Vertical Scaling.
        i. https://medium.com/@ayush_mittal/horizontal-vs-vertical-scaling-scalability-system-design-d10658b7f94e
        ii. https://www.youtube.com/watch?v=krgzOa7Hp_o
        iii. https://www.digitalocean.com/resources/articles/horizontal-scaling-vs-vertical-scaling
    - Asynchronous communication (message queues like RabbitMQ, Kafka), Publisher Subscriber Model.
        i. https://www.geeksforgeeks.org/what-is-pub-sub/
        ii. https://learn.microsoft.com/en-us/azure/architecture/patterns/publisher-subscriber
        iii. https://www.youtube.com/watch?v=FMhbR_kQeHw
    - Content Delivery Networks (CDNs).
        i. https://www.cloudflare.com/en-gb/learning/cdn/what-is-a-cdn/
        ii. https://www.youtube.com/watch?v=RI9np1LWzqw

2. **System Reliability and Fault Tolerance**:
    - Data replication and backups.
        i. https://www.unitrends.com/blog/backup-vs-replication
        ii. https://www.geeksforgeeks.org/replication-in-system-design/
        iii. https://www.geeksforgeeks.org/database-replication-and-their-types-in-system-design/
    - Handling failures with retries and fallbacks.
        i. https://www.codecentric.de/wissens-hub/blog/resilience-design-patterns-retry-fallback-timeout-circuit-breaker
        ii. https://harish-bhattbhatt.medium.com/best-practices-for-retry-pattern-f29d47cd5117

3. **APIs and Communication**:
    - REST APIs vs. gRPC.
        i. https://blog.postman.com/grpc-vs-rest/
        ii. https://thecodemood.com/grpc-vs-rest/

- Event-driven communication for real-time systems.
   i. https://www.confluent.io/learn/event-driven-architecture
   ii. https://www.youtube.com/watch?v=Tu1GEIhkIqU

4. **Database Design**:
   - Choose SQL or NoSQL based on requirements.
      i. https://www.geeksforgeeks.org/difference-between-sql-and-nosql/
      ii. https://www.geeksforgeeks.org/which-database-to-choose-while-designing-a-system-sql-or-nosql/
      iii. https://medium.com/geekculture/choosing-the-right-database-for-system-design-sql-vs-nosql-and-beyond-d58fde5a6fe3
   - CAP Theorem, eventual consistency, BASE
      i. https://www.geeksforgeeks.org/the-cap-theorem-in-dbms/
      ii. https://www.scylladb.com/glossary/eventual-consistency
      iii. https://www.geeksforgeeks.org/acid-model-vs-base-model-for-database/
   - Design for partitioning, indexing, and sharding.
      i. https://www.cockroachlabs.com/blog/what-is-data-partitioning-and-how-to-do-it-right/
      ii. https://www.techtarget.com/searchoracle/definition/sharding
      iii. https://learn.microsoft.com/en-us/azure/architecture/patterns/sharding

5. **Load Management**:
   - Load balancing techniques.
      i. https://www.cloudflare.com/en-gb/learning/performance/what-is-load-balancing/
      ii. https://www.f5.com/glossary/load-balancer
   - Rate limiting and throttling.
      i. https://systemsdesign.cloud/SystemDesign/RateLimiter
      ii. https://www.geeksforgeeks.org/rate-limiting-in-system-design/

Preparation Steps

1. **Practice Use Cases**:
   - Design solutions for common problems from practise interviews etc. (in resources given below)
2. **Understand Real-World Systems**:
   - Read about architectures of popular systems (Netflix, Instagram, YouTube).
3. **Visualize Solutions**:
   - Practice drawing architecture diagrams using tools like Lucidchart or Draw.io.

Resources

- **Grokking** (This repo has a very comprehensive guide on how to approach HLD, and also different resources for every key concept)
  - https://github.com/Jeevan-kumar-Raj/Grokking-System-Design
- **Key concepts** (Excellent playlist covering everything in HLD)
  - https://www.youtube.com/watch?v=SqcXvc3ZmRU&list=PLMCXHnjXnTnvo6alSjVkgxV-VH6EPyvoX
- **Books**:
  - *System Design Interview* by Alex Xu.
  - *Designing Data-Intensive Applications* by Martin Kleppmann.
- **Practise interview playlists**:
  - https://www.youtube.com/watch?v=NtMvNh0WFVM&list=PLrtCHHeadkHp92TyPt1Fj452_VGLipJnL
  - https://www.youtube.com/watch?v=wL-Gx5XE9XE
- **Blogs**:
  - https://slack.engineering/
  - https://aws.amazon.com/blogs/architecture/
  - https://stripe.com/blog/engineering
  - https://engineering.linkedin.com/blog
  - https://engineering.fb.com/
  - https://blog.cloudflare.com/
  - http://eng.uber.com/
  - https://medium.com/netflix-techblog

NOTE:

For last minute practise go through Gaurav Sens youtube playlist for all of the key concepts, then a system design practise interview from above playlists for each type of problems in grokking repository.

---

2. Low-Level Design (LLD)

Key Areas to Focus On

1. **Object-Oriented Design (OOD)**:
        - https://www.geeksforgeeks.org/oops-object-oriented-design/
    - Understand SOLID principles for maintainable code.
        - https://www.digitalocean.com/community/conceptual-articles/s-o-l-i-d-the-first-five-principles-of-object-oriented-design

        - https://www.geeksforgeeks.org/solid-principle-in-programming-understand-with-real-life-examples/
        - https://www.youtube.com/watch?v=kF7rQmSRlq0&pp=ygULI2JpdHdpc2Vub3Q%3D
    - Practice design patterns (Factory, Singleton, Builder, Strategy)
        - https://www.youtube.com/watch?v=OuNOyFg942M
        - https://medium.com/@theautobot/design-patterns-in-java-singleton-factory-and-builder-317cb407c2e7
        - https://www.geeksforgeeks.org/difference-between-singleton-and-factory-design-pattern-in-java/
        - https://www.geeksforgeeks.org/modern-c-design-patterns-tutorial/
        - https://www.geeksforgeeks.org/software-design-patterns/

2. **Class Design**:
    - Break a problem into entities and relationships.
        - https://www.geeksforgeeks.org/introduction-of-er-model/
        - https://www.geeksforgeeks.org/how-to-draw-entity-relationship-diagrams
        - https://www.javatpoint.com/software-engineering-entity-relationship-diagrams
    - Ensure SRP (Single Responsibility Principle) is adhered to.
        - https://www.geeksforgeeks.org/single-responsibility-in-solid-design-principle/
3. **Code Modularity**:
    - Write extensible and reusable code.
        - https://best-practice-and-impact.github.io/qa-of-code-guidance/modular_code.html
    - Use interfaces and abstract classes effectively.
        - https://www.infoworld.com/article/2171958/when-to-use-abstract-classes-vs-interfaces-in-java.html
4. **Design for Concurrency**:
    - Understand thread safety, synchronization, and race conditions.
        - https://web.mit.edu/6.005/www/fa14/classes/17-concurrency/#:~:text=Concurrency%20means%20multiple%20computations%20are,cores%20on%20a%20single%20chip)
        - https://www.geeksforgeeks.org/concurrency-in-operating-system/
        - https://www.geeksforgeeks.org/multithreading-in-cpp/
        - https://www.digitalocean.com/community/tutorials/thread-safety-in-java
        - https://www.geeksforgeeks.org/race-condition-vulnerability/
        - https://www.geeksforgeeks.org/mutex-lock-for-linux-thread-synchronization/

Resources

- **Concepts**:
    - https://dev.to/srishtikprasad/low-level-design-and-solid-principles-4am9
    - https://www.geeksforgeeks.org/what-is-low-level-design-or-lld-learn-system-design/
    - https://www.youtube.com/playlist?list=PL6W8uoQQ2c61X_9e6Net0WdYZidm7zooW
- **Books**:
    - *Head First Design Patterns* by Eric Freeman.
    - *Design Patterns: Elements of Reusable Object-Oriented Software* by GoF.
- **Practice**:
    - *Exercism.io* for practical LLD exercises.
    - Mock interviews on Pramp for LLD scenarios.
    - https://www.youtube.com/playlist?list=PL6W8uoQQ2c61X_9e6Net0WdYZidm7zooW

---

3. Interview Strategy

For High-Level Design (HLD):

1. **Focus on the 4 S's**:
    - Scalability: How does your design handle millions of users?
    - Simplicity: Keep the design straightforward and understandable.
    - Security: Incorporate authentication, authorization, and encryption.
    - Stability: Ensure fault tolerance and backup mechanisms.
2. **Explain Trade-Offs**:
    - Discuss why you chose a particular database, caching strategy, or architecture.
3. **Think Big and Scale Later**:
    - Start small and explain how the system can evolve with increased traffic.

For Low-Level Design (LLD):

1. **Master Class and Relationship Design**:
    - Think of entities and how they interact (e.g., inheritance or composition).
2. **Follow a Methodical Approach**:
    - Analyze the problem -> Identify classes -> Define methods and relationships.
3. **Write Clean, Testable Code**:
    - Stick to principles like DRY (Don’t Repeat Yourself) and KISS (Keep It Simple, Stupid).