[CAP Theorem (Consistency, Availability, Partition Tolerance) 1](#_Toc192530177)

## CAP Theorem (Consistency, Availability, Partition Tolerance)

The **CAP theorem**, proposed by **Eric Brewer**, states that in a **distributed system**, you can only achieve **two out of three** guarantees at any given time:

1️⃣ **Consistency (C)** → Every read receives the most recent write (or an error).  
2️⃣ **Availability (A)** → Every request gets a response (even if outdated or partial).  
3️⃣ **Partition Tolerance (P)** → The system continues to function despite network failures.

**Trade-offs in CAP Theorem**

| **CAP Property** | **Impact** |
| --- | --- |
| **CA (Consistency + Availability, no Partition Tolerance)** | Works only in a **single-node system** or when the network is **fully reliable** (e.g., traditional relational databases like MySQL in a local setup). |
| **CP (Consistency + Partition Tolerance, no Availability)** | Ensures **data consistency** but might reject requests during network failures (e.g., MongoDB with strong consistency settings). |
| **AP (Availability + Partition Tolerance, no Consistency)** | Ensures **high availability** but may serve **stale data** during partitions (e.g., DynamoDB, Cassandra). |

**Real-World Examples**

* **CA:** Relational Databases (MySQL, PostgreSQL) without replication.
* **CP:** Zookeeper, MongoDB (with strong consistency).
* **AP:** DynamoDB, Cassandra, Riak (focus on availability and partition tolerance).

⚡ **Key takeaway:** In a **distributed system**, network failures **will happen** → so **Partition Tolerance (P) is always required**, meaning we must choose between **Consistency (CP) or Availability (AP).**

## Patterns

**1️⃣ Factory Pattern**

✅ **Purpose:** Used for **creating objects** without specifying their exact class.  
✅ **How it Works:** A **factory method** returns an instance of a class based on input parameters.  
✅ **Example:**

* **Scenario:** Creating different shapes (Circle, Square) without directly instantiating them.
* **Used In:** Dependency Injection, Object Pooling.

📌 **Example in Python:**

class Shape:

def draw(self):

pass

class Circle(Shape):

def draw(self):

return "Drawing a Circle"

class Square(Shape):

def draw(self):

return "Drawing a Square"

class ShapeFactory:

@staticmethod

def get\_shape(shape\_type):

if shape\_type == "circle":

return Circle()

elif shape\_type == "square":

return Square()

shape = ShapeFactory.get\_shape("circle")

print(shape.draw()) # Output: Drawing a Circle

**2️⃣ Singleton Pattern**

✅ **Purpose:** Ensures that **only one instance** of a class is created and provides a **global access point** to it.  
✅ **How it Works:** Uses a **private constructor** and a **static method** to return a single instance.  
✅ **Example:**

* **Scenario:** Database connections, logging frameworks.
* **Used In:** Connection pools, Thread management.

📌 **Example in Python:**

class Singleton:

\_instance = None # Private instance variable

def \_\_new\_\_(cls):

if cls.\_instance is None:

cls.\_instance = super().\_\_new\_\_(cls)

return cls.\_instance

obj1 = Singleton()

obj2 = Singleton()

print(obj1 == obj2) # Output: True (Both refer to the same instance)

**3️⃣ Observer Pattern**

✅ **Purpose:** Allows **one-to-many dependencies**, meaning **when one object changes, all its dependents get notified automatically**.  
✅ **How it Works:** The **Subject** keeps a list of **Observers** and notifies them of any changes.  
✅ **Example:**

* **Scenario:** A stock market app where multiple users get updates when stock prices change.
* **Used In:** Event handling, Pub-Sub systems.

📌 **Example in Python:**

class Observer:

def update(self, message):

pass

class User(Observer):

def \_\_init\_\_(self, name):

self.name = name

def update(self, message):

print(f"{self.name} received update: {message}")

class StockMarket:

def \_\_init\_\_(self):

self.observers = []

def register(self, observer):

self.observers.append(observer)

def notify(self, message):

for observer in self.observers:

observer.update(message)

market = StockMarket()

user1 = User("Alice")

user2 = User("Bob")

market.register(user1)

market.register(user2)

market.notify("Stock price increased!")

# Output:

# Alice received update: Stock price increased!

# Bob received update: Stock price increased!

**4️⃣ Model-View-Controller (MVC) Pattern**

✅ **Purpose:** Separates **business logic, UI, and user input** for better maintainability.  
✅ **How it Works:**

* **Model (M)** → Handles data and logic.
* **View (V)** → Displays the UI.
* **Controller (C)** → Connects user input with the model and updates the view.  
  ✅ **Example:**
* **Scenario:** A web application where the **View** updates whenever the **Model** changes.
* **Used In:** Django, Flask, React-Redux apps.

📌 **Example in Python (Simplified MVC)**

class Model:

def get\_data(self):

return "Hello, MVC!"

class View:

def display(self, data):

print(f"View: {data}")

class Controller:

def \_\_init\_\_(self, model, view):

self.model = model

self.view = view

def update\_view(self):

data = self.model.get\_data()

self.view.display(data)

model = Model()

view = View()

controller = Controller(model, view)

controller.update\_view() # Output: View: Hello, MVC!

**📌 Summary Table**

| **Pattern** | **Purpose** | **Example Use Case** |
| --- | --- | --- |
| **Factory Pattern** | Create objects dynamically without specifying exact class | Shape creation, Dependency Injection |
| **Singleton Pattern** | Ensure only one instance of a class exists | Database connections, Logging |
| **Observer Pattern** | Notify multiple objects when a subject changes | Stock market alerts, Event listeners |
| **MVC Pattern** | Separate logic, UI, and user input for maintainability | Web applications, GUI frameworks |

Let me know if you need further explanations! 🚀