# **Django**

* MVT(Model View Template):  
  Diagram of a diagram of a computer

  Description automatically generated  
  1. MVC (Model-View-Controller):
* Model: Manages the data, business logic, and rules related to the data. It interacts with the database and provides the data required by the controller.
* View: Responsible for presenting data to the user. It is the part of the application that generates the user interface (UI).
* Controller: Acts as the intermediary between the model and the view. It handles user input, updates the model, and passes the data to the view for display.
* 2. MVT (Model-View-Template):
* Model: Similar to MVC, it manages the data and business logic. In Django, this usually involves defining models that map to database tables and handle data interactions.
* View: In Django’s MVT, the "view" is actually what we call the "controller" in MVC. It handles the logic of processing user requests and returning responses. It pulls data from the model and passes it to the template for rendering.
* Template: This is Django’s replacement for the "view" in MVC. The template is responsible for rendering the UI and displaying data dynamically using the Django template language.

**Key Difference:**

* In **MVC**, the **Controller** is separate, and the **View** is responsible for rendering the UI.
* In **MVT**, Django's **View** acts like the controller, and the **Template** is responsible for rendering the UI.

**Summary:**

* **MVT**: The Django **view** handles the logic (controller-like), and the **template** renders the UI.
* **MVC**: The **controller** manages input and the **view** renders the UI directly.

DRF:  
 **DRF** simplifies API development by providing tools like serializers and viewsets.

 **Serializers** handle the conversion between Django models and JSON data.

 **Viewsets** manage CRUD operations with minimal code.

 You can make **GET, POST, PUT, DELETE** requests to interact with your API.

# **Database**

A **ForeignKey** in Django is a field used to define a many-to-one relationship between two models. It connects one model (the "child" model) to another model (the "parent" model). The ForeignKey field allows rows in one table (the child) to reference a row in another table (the parent).

**Key Points:**

* It creates a relationship where multiple records in the child model can reference a single record in the parent model.
* Useful for relationships like "a book belongs to one author" or "a post has one category."

**ACID** stands for **Atomicity, Consistency, Isolation, and Durability**—principles ensuring reliable database transactions.

1. **Atomicity**: A transaction is all-or-nothing. If any part fails, the whole transaction is rolled back.
   * Example: In a bank transfer, either both accounts are updated, or neither is.
2. **Consistency**: A transaction brings the database from one valid state to another, maintaining all rules.
   * Example: After a transfer, total money in both accounts stays the same.
3. **Isolation**: Transactions don't interfere with each other; results are as if executed one by one.
   * Example: Multiple transfers can happen simultaneously without affecting each other.
4. **Durability**: Once a transaction is committed, changes are permanent, even in case of failure.
   * Example: After completing a transfer, it remains, even if the system crashes.

# **OOP**

1. **Class and Object**: A **class** is like a bank’s account template, and each customer’s account (e.g., John’s account) is an **object** created from that template.
2. **Encapsulation**: Customer details like the balance are hidden and only accessible through specific methods like depositing and withdrawing.
3. **Inheritance**: A **SavingsAccount** or **CurrentAccount** inherits common behaviors (like deposit/withdraw) from a **BankAccount**, but they can also have unique features.
4. **Polymorphism**: Different account types can share the same methods (like account\_type), but they behave differently depending on the account.
5. **Abstraction**: Customers can withdraw money without knowing how the bank handles the underlying transactions.

OOP helps in structuring complex real-world scenarios into manageable, reusable, and scalable code!

Method overloading vs overriding   
OOP তে দুইটা কনফিউশন তৈরি করা টপিক হচ্ছে overloading আর overriding। দুইটাই আসলে পলিমরফিজম এর দুইটা টাইপ মাত্র।   
overloading:  
১. compile time পলিমোরফিজম   
২. এটা স্ট্যাটিক বাইন্ডিং করে  
৩. দুই বা ততোধিক ফাংশন এর নাম সেম কিন্তু প্যারামিটার, রিটার্ন টাইপ আলাদা হবে।  
  
overriding:  
১. run time পলিমরফিজম, ইনহেরিটেন্স এ ইউজ হয়  
২. ডাইনামিক বাইন্ডিং করে  
৩. base বা প্যারেন্ট ক্লাসে মেথড টা যেভাবে থাকবে child ক্লাসে মেথড এর প্যারামিটার, রিটার্ন টাইপ সেম থাকতে হবে।

# **Basic Python**

* **Lists** are used for ordered collections that may change.
* **Tuples** are used for ordered collections that do not change.
* **Dictionaries** are used for key-value pairs where the order does not matter.
* **Sets** are used for unordered collections of unique elements.
* **Strings** are used for immutable sequences of characters.

These data structures are fundamental in Python programming and help manage and manipulate data efficiently.