Models

- In Flask-SQLAlchemy, a "model" refers to a Python class that represents a database table.
- In traditional relational databases, tables are used to store data. Each table has columns (fields) that define the structure of the data it holds. In Flask-SQLAlchemy, you define these tables as Python classes, and each attribute of the class represents a column in the table.

Column

- 'primary_key': a boolean property use to define the primary key column of a database table. When primary_key=True, the column is primary_key, unique, not null and auto increase.
- 'nullable': a boolean property use to define column can store null value or not
- 'unique': a boolean property use to define column must store unique value across all rows in the corresponding database table

Common Data Type

Ojbect Name	Description
Integer	an integer
String(size)	a string with a maximum length (optional in some databases, e.g. PostgreSQL)
Text	some longer unicode text
DateTime	date and time expressed as Python datetime object.
Float	stores floating point values
Boolean	stores a boolean value
PickleType	stores a pickled Python object
LargeBinary	stores large arbitrary binary data

Relationship

- `relationship`: Provide a relationship between two mapped classes. This corresponds to a parent-child or associative table relationship.
- Both `back_populates` and `backref` are used to define bidirectional relationships between models.
  - `back_populates`: Requires explicitly defining the relationship in both models
  - `backref`: Automatically sets up the reverse relationship using the backref argument in one of the models. You only need to specify the relationship in one model, and SQLAlchemy handles the creation of the complementary attribute in the other model.
- The `lazy` parameter determines when and how the related data is loaded when querying the database
  - `lazy='select' or True` (the default): This is the default loading strategy. It means that the related data will be loaded lazily when you access the attribute representing the relationship. The related objects will be fetched from the database in a separate SELECT statement only when you use the children attribute on the parent object.

- `lazy='joined'`: This strategy performs a SQL JOIN between the two related tables when querying the parent model. It fetches the related data along with the parent data in a single query. This can be more efficient when you know that you will always need the related data together with the parent data.
- `lazy='dynamic'`: This strategy returns a query object instead of a collection of objects. When you access the attribute representing the relationship, you get a query object that you can further filter, paginate, or perform other database operations on. The related data is loaded from the database when you execute the query.

1. One-to-Many Relationships

```
  class Parent(Base):
    __tablename__ = 'left'__
    id = Column(Integer, primary_key=True)

    children = relationship("Child", back_populates="parent")

  class Child(Base):
    __tablename__ = 'right'
    id = Column(Integer, primary_key=True)
    left_id = Column(Integer, ForeignKey('left.id'), unique=True,
uselist=False)

    parent = relationship("Parent", back_populates="children")
```

2. One-to-One Relationships

- Use `uselist=False` in `relationship` mapping
- `unique=True` for Foreign Key of the relationship

```
  class Parent(Base):
    __tablename__ = 'left'__
    id = Column(Integer, primary_key=True)

    child = relationship("Child", back_populates="parent")

  class Child(Base):
    __tablename__ = 'right'
    id = Column(Integer, primary_key=True)
    left_id = Column(Integer, ForeignKey('left.id'), unique=True,
  uselist=False)

    parent = relationship("Parent", back_populates="child")
```

3. Many-to-Many Replationships

```
class Association(Base):
  __tablename__ = 'association'
  left_id = Column(Integer, ForeignKey('left.id'), primary_key=True)
  right_id = Column(Integer, ForeignKey('right.id'), primary_key=True)
  extra_data = Column(String(50))

  child = relationship("Child", back_populates="parents")
  parent = relationship("Parent", back_populates="children")

class Parent(Base):
  __tablename__ = 'left'
  id = Column(Integer, primary_key=True)

  children = relationship("Association", back_populates="parent")

class Child(Base):
  __tablename__ = 'right'
  id = Column(Integer, primary_key=True)

  parents = relationship("Association", back_populates="child")
```