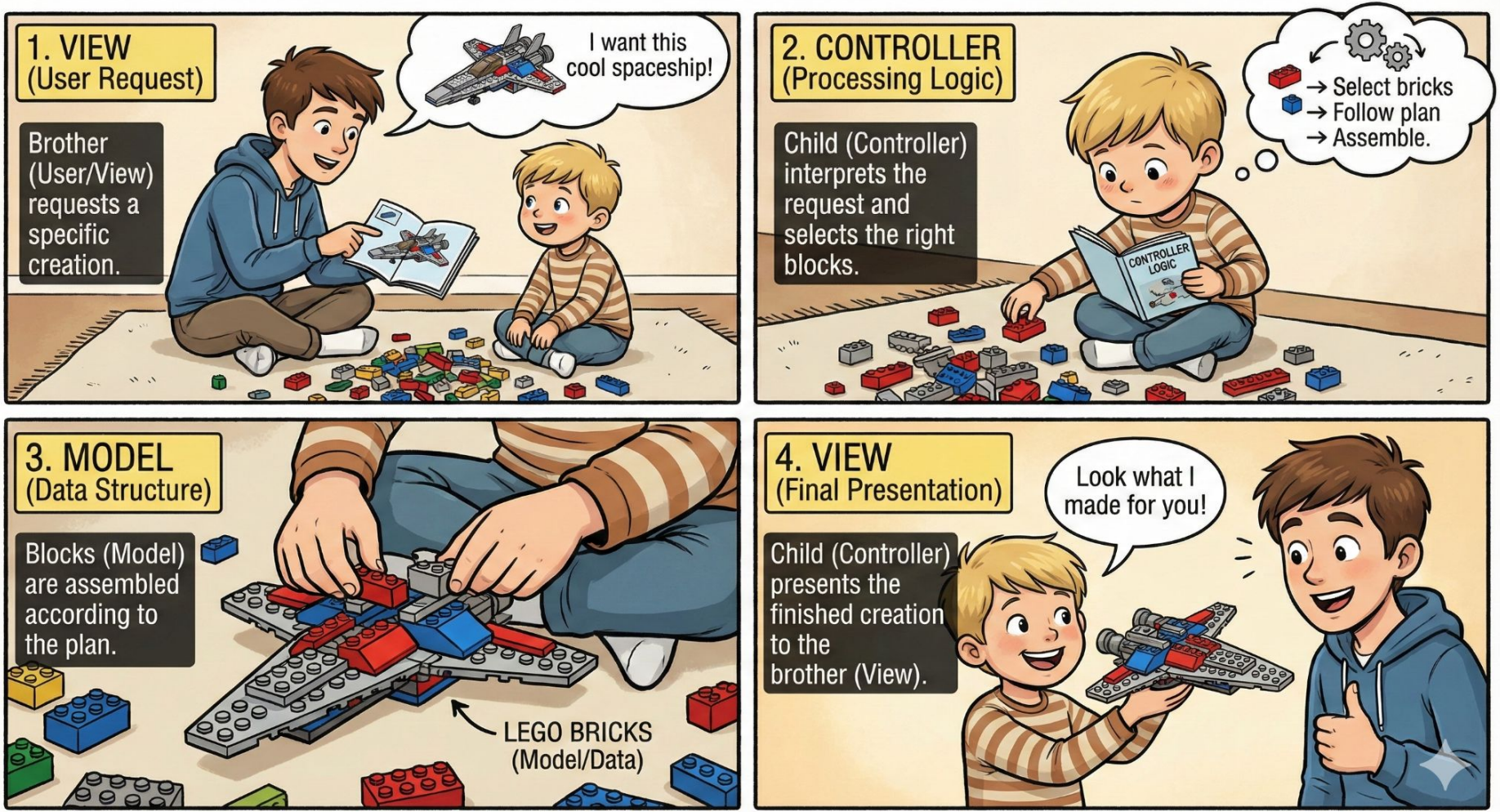
1. VIEW
(User Request)
Brother (User/View) requests a specific creation.
I want this cool spaceship!
2. CONTROLLER
(Processing Logic)
Child (Controller) interprets the request and selects the right blocks.
→ Select bricks
→ Follow plan
→ Assemble.
CONTROLLER LOGIC
3. MODEL
(Data Structure)
Blocks (Model) are assembled according to the plan.
LEGO BRICKS
(Model/Data)
4. VIEW
(Final Presentation)
Child (Controller) presents the finished creation to the brother (View).
Look what I made for you!

TypeScript Download Docs Handbook Community Playground Tools 한국어 Search Do

Playground TS Config Examples Help

v5.9.3 Run Export Share

.JS .D.TS Errors Logs Plugi

```
interface IProduct {
  name: string;
  unitPrice: number;
}

function calculateTotalPrice(product: IProduct, quantity: number, discount: number): number {
  // [수정된 부분] unitPrice.price가 아니라 product.unitPrice 입니다.
  var priceWithoutDiscount: number = product.unitPrice * quantity;
  var discountAmount: number = priceWithoutDiscount * discount;
  return priceWithoutDiscount - discountAmount;
}

// [추가된 부분] 실제로 함수를 실행하고 결과를 출력하는 코드
const myProduct: IProduct = { name: "MacBook", unitPrice: 2000000 };
const result = calculateTotalPrice(myProduct, 1, 0.1); // 1개 구매, 10% 할인

console.log("최종 가격:", result);
```

```
[LOG]: "최종 가격:",  1800000
```

TypeScript

```typescript
class Product {
  // 1. 속성 (데이터) 정의
  name: string;
  unitPrice: number;

  // 2. 생성자 (Constructor): 처음 물건을 만들 때 실행되는 함수
  constructor(name: string, unitPrice: number) {
    this.name = name;
    this.unitPrice = unitPrice;
  }

  // 3. 메서드 (기능): 함수가 클래스 안으로 들어왔습니다.
  // 더 이상 product를 매개변수로 받을 필요가 없습니다. (내 정보를 쓰면 되니까요)
  calculateTotalPrice(quantity: number, discount: number): number {
    // 'this'는 '내 자신'을 가리킵니다. (내 가격 * 수량)
    var priceWithoutDiscount: number = this.unitPrice * quantity;
    var discountAmount: number = priceWithoutDiscount * discount;
    return priceWithoutDiscount - discountAmount;
  }
}

// --- 실행 코드 ---

// 1. 'new' 키워드로 실제 상품(객체)을 만듭니다.
const macbook = new Product("MacBook", 2000000);

// 2. 상품이 스스로 자신의 가격을 계산합니다.
const result = macbook.calculateTotalPrice(1, 0.1);

console.log(`${macbook.name}의 최종 가격:`, result);
```

Python

```python
class Product:
    # 1. 생성자 (__init__): TypeScript의 constructor와 같습니다.
    # self는 '나 자신(this)'을 의미하며, 함수 첫 번째 인자로 꼭 써줘야 합니다.
    def __init__(self, name: str, unit_price: int):
        self.name = name
        self.unit_price = unit_price

    # 2. 메서드 (기능)
    # Python은 변수 타입(int, float)을 적는 것이 선택 사항(Type Hinting)입니다.
    def calculate_total_price(self, quantity: int, discount: float) -> float:
        # this.unitPrice 대신 self.unit_price를 사용합니다.
        price_without_discount = self.unit_price * quantity
        discount_amount = price_without_discount * discount

        return price_without_discount - discount_amount

# --- 실행 코드 ---

# 1. 객체 생성 ('new' 키워드가 필요 없습니다!)
macbook = Product("MacBook", 2000000)

# 2. 메서드 실행
result = macbook.calculate_total_price(1, 0.1)

# f-string을 사용해 문자열 안에 변수를 넣습니다. (TypeScript의 `${}`와 비슷)
print(f"{macbook.name}의 최종 가격: {result}")
```

Python

```python
# 1. BaseModel을 상속받습니다. (Pydantic의 핵심)
class Product(BaseModel):
    # 2. __init__ 없이 변수와 타입을 바로 선언합니다.
    # 이제 이 클래스는 데이터가 들어올 때 'name'이 문자인지, 'unit_price'가 숫자인지 "자동
    name: str
    unit_price: float

    # 3. 기능(메서드)은 그대로 작성하면 됩니다.
    def calculate_total_price(self, quantity: int, discount: float) -> float:
        price_without_discount = self.unit_price * quantity
        discount_amount = price_without_discount * discount
        return price_without_discount - discount_amount

# --- 실행 코드 ---

try:
    # 1. 객체 생성 (키워드 인자를 사용하는 것을 권장합니다)
    macbook = Product(name="MacBook", unit_price=2000000)

    # 2. 메서드 실행
    result = macbook.calculate_total_price(quantity=1, discount=0.1)

    print(f"{macbook.name}의 최종 가격: {result}")

    # [Pydantic의 장점 테스트]
    # 만약 숫자가 들어가야 할 가격에 '문자'를 넣으면 어떻게 될까요?
    # error_product = Product(name="ErrorBook", unit_price="이백만원") # <- 여기서 바

except Exception as e:
    print(f"에러 발생: {e}")
```

1. 📁 프로젝트 구조

Plaintext

```
my_project/
├── models.py      # [Model] 데이터 구조 (Pydantic)
├── services.py    # [Logic] 실제 계산 로직
├── controllers.py # [Controller] URL 요청 처리
└── main.py        # [App] 실행 파일
```

(1) `models.py` (Model)

데이터의 모양(Type)을 정의합니다. `IProduct` 인터페이스와 같은 역할입니다.

Python

```python
from pydantic import BaseModel

# 사용자가 보낼 데이터 형태
class ProductDto(BaseModel):
    name: string
    unit_price: float
```

(2) `services.py` (Business Logic)

실제 계산(기능)을 담당합니다. 컨트롤러가 "이거 계산해줘"라고 시키면 일을 수행하는 곳입니다.

Python

```python
from models import ProductDto

def calculate_logic(product: ProductDto, quantity: int, discount: float) -> float:
    # 비즈니스 로직 (이전 클래스의 메서드 내용)
    price_without_discount = product.unit_price * quantity
    discount_amount = price_without_discount * discount
    return price_without_discount - discount_amount
```

(3) `controllers.py` (Controller)

사용자의 접속(URL)을 받아들이는 창구입니다.

Python

```python
from fastapi import APIRouter
from models import ProductDto
import services # 로직을 가져옴

router = APIRouter()

# POST 요청을 받음: http://localhost:8000/calculate
@router.post("/calculate")
def get_total_price(product: ProductDto, quantity: int, discount: float):

    # 1. 서비스(로직)에게 계산을 시킵니다.
    final_price = services.calculate_logic(product, quantity, discount)

    # 2. 결과(View)를 JSON으로 돌려줍니다.
    return {
        "product_name": product.name,
        "quantity": quantity,
        "final_price": final_price,
        "message": "계산이 성공적으로 완료되었습니다."
    }
```

(4) `main.py` (Entry Point)

앱을 실행하고 컨트롤러를 등록하는 메인 파일입니다.

Python

```python
from fastapi import FastAPI
from controllers import router # 컨트롤러 가져오기

app = FastAPI()

# 컨트롤러(라우터) 등록
app.include_router(router)

# 실행: uvicorn main:app --reload
```

FastAPI MVC Architecture Flow