FastAPI

- Week 1
- Week 2
- Week 3
- Week 4
- Week 5?

Week 1

Week 1

Join the github classroom assignment with the following link:
https://classroom.github.com/a/KOc7f0j5

Once you join your repository will be created https://github.com/blueprint-learn/week1-techteam-sp26-{github-username}

You will get an email with access to your repo

Clone your repo

```
git clone https://github.com/blueprint-learn/week1-techteam-sp26-{github-username}.git
```

Goal for today

> By the end of this class, you will understand what an API is **and** you will have written and run a real backend service.

Set expectations

- This is *not* web dev with HTML
- This is backend infrastructure
- What we build today looks simple, but it's the same pattern used by Netflix, Stripe, AWS, etc.

1. What Is an API?

Core idea

- API = *contract* between a client and a server
- You ask for data → server responds with data

```
Client (browser / app / script)
        |
        |  HTTP request
```

```
        v
      API Server
        |
        |  JSON response
        v
      Client
```

Key clarification

- APIs don't care *who* the client is
- Browser, mobile app, another server — all the same

Examples

- Instagram feed
- Spotify playlists
- Google Maps directions

2. REST & HTTP Basics

REST (high-level)

- Everything is a *resource*
- Each resource has a URL

HTTP Methods

Method	Meaning
GET	Read data
POST	Create data
PUT	Update data
DELETE	Remove data

HTTP Response

- Status code (200, 404, etc.)
- Body (usually JSON)

```
{
  "id": 3,
  "name": "Notebook",
  "price": 5.99
}
```

Important

- JSON ≠ Python dictionary (but very similar)

3. Live Demo Setup

```
pip3 install -r requirements.txt
```

Minimal app

```
from fastapi import FastAPI

app = FastAPI()

@app.get("/")
def root():
    return {"message": "Hello World"}
```

Run the app

```
uvicorn app.main:app --reload
```

4. Endpoints & Path Parameters

```
@app.get("/items/{item_id}")
def read_item(item_id: int):
    return {"item_id": item_id}
```

- Automatic validation
- Type validation

Week 2

Getting Started

Join the github classroom assignment with the following link: https://classroom.github.com/a/Cg-MTRGE

Once you join your repository will be created https://github.com/blueprint-learn/week2-techteam-sp26-{github-username}

Once you join, GitHub Classroom will automatically create your personal repository. You will receive an email confirming access.

After receiving access:

1. Clone your repository locally
2. Install dependencies
3. Run the starter FastAPI app

This repository contains the scaffolding you will extend during this week.

```
git clone https://github.com/blueprint-learn/week1-techteam-sp26-{github-username}.git
```

Recap last week

In Week 1, we established the core foundation of APIs using FastAPI. By the end of the session, you had:

- Built a running FastAPI server
- Created basic endpoints
- Used path parameters
- Verified correctness through automated CI tests

You also saw how an API defines a contract between a client and a server, and how tests enforce that contract.

Goal for today

By the end of this week, you will understand how APIs receive structured data and how FastAPI automatically validates incoming data.

This is a major step forward: your API will no longer only return data — it will safely accept data from clients.

Big idea

Last week, the interaction pattern was:

Client asks for data → Server returns data

This week, we expand that model:

Client sends data → Server validates and accepts data

This shift introduces one of the most important responsibilities of a backend API: validating inputs at the system boundary.

1. How Data Enters an API

There are three primary ways data reaches an API endpoint. Understanding these channels is essential for designing correct APIs.

Type	Where used	Example
Path	Identify resource	/users/3
Query	Filter/search	?limit=10
Body	Send data	POST JSON

Each serves a different purpose:

- Path parameters identify **which** resource
- Query parameters refine **how** to retrieve
- Request bodies provide **new data**

Path parameters

Path parameters help define a resource. They usually appear after a / . IDs are usually passed as path parameters to identify a resource.

```
/items/5
```

Query parameters

Query parameters appear after a ? in the URL and are commonly used for filtering, pagination, or search controls.

Examples to test:

```
/items?limit=10
```

```
@app.get("/items")
def list_items(limit: int = 10):
    return {"limit": limit}
```

FastAPI automatically:

- Applies default values when missing
- Converts types based on type hints
- Validates incorrect input

For example, if limit is defined as an integer and a string is provided, FastAPI will reject the request before your function executes.

Key insight:

FastAPI parses query strings into typed Python values.

Request body (JSON)

Real-world APIs must accept structured data. Common examples include:

- User signup information
- Orders or transactions
- Messages or posts
- Payment details

Clients send this data as JSON in the request body.

```
{
  "name": "Notebook",
  "price": 5.99
}
```

```
@app.post("/items")
def create_item(item: dict):
    return item
```

A naive implementation might accept this as a generic dictionary. However, this approach has critical problems:

- No validation
- No defined structure
- No guarantees about data shape

This leads to unstable and unsafe APIs

We need a schema.

4. Pydantic Models

FastAPI uses Pydantic models to define structured request bodies.

A Pydantic model specifies:

- Required fields
- Types
- Optional fields
- Validation rules

When used in an endpoint, FastAPI:

- Parses incoming JSON
- Validates types and structure
- Converts into a Python object
- Rejects invalid requests automatically

This provides:

- Schema definition
- Type enforcement
- Automatic parsing
- Validation

The model becomes the contract for accepted data.

Why Validation Matters

Validation at the API boundary protects the system from invalid or malicious input.

Without validation, systems risk:

- Database corruption
- Runtime crashes
- Security vulnerabilities
- Inconsistent data

With validation, APIs ensure:

- Safe inputs
- Predictable structure
- Stable behavior

Core principle:

Validation is defensive programming at the API boundary.

Define

```
from pydantic import BaseModel


class Item(BaseModel):
    name: str
    price: float
```

Use

```
@app.post("/items")
def create_item(item: Item):
    return item
```

Explain

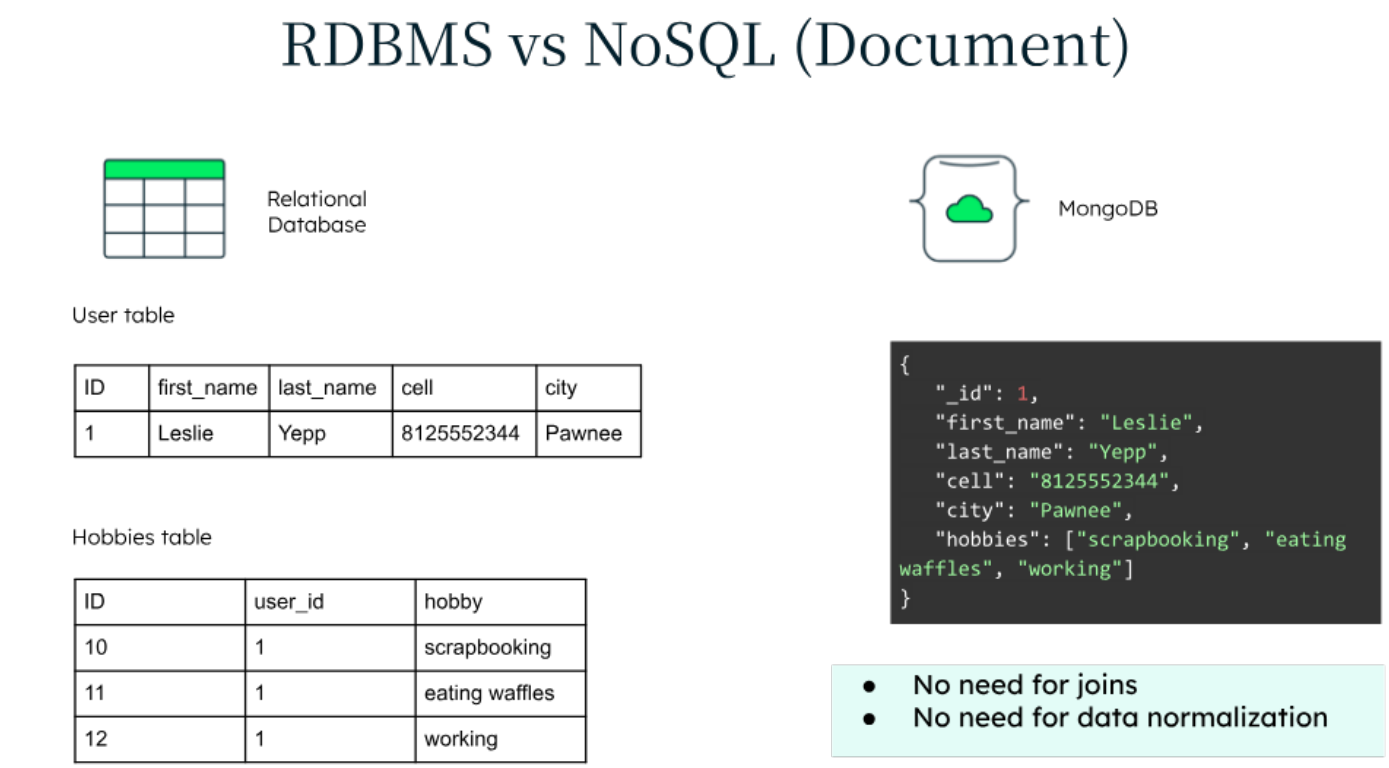
- Schema definition
- Type enforcement
- Automatic parsing
- Validation

Week 3

Before we get into PostgreSQL, we need to understand why we use databases in the first place. Right now, if our apps use dictionaries to store our data. The problem is that the data disappears when the server restarts, creating a problem in the future. A database allows us to store our data permanently so it survives restarts, crashes, and can support concurrent users.

There are two categories of databases

	Relational (SQL)	Non-Relational (NoSQL)
Structure	Tables with rows and columns	Similar to JSON
Schema	Fixed, predefined	Flexible, schema-less
Relationships	Tables link with keys	Handled in the app's code
Query Language	SQL	Varies but usually none
Best for	Structured data that relates to each other	Unstructured or rapidly changing data
Examples	PostgreSQL, MySQL	MongoDB, Redis



What is a relational database?

A relational database organizes data into tables. Each table represents one thing (a resource, a user, an order, etc). Each row is one record and each column is an attribute of that record, with a defined data type.

id	first_name	last_name	email
1	Miguel	Merlin	mmerlin@stevens.edu
2	Nishit	Sharma	nsharma11@stevens.edu

Because these tables are relational, that means they can be linked to each other. An orders table can reference a user table, so every order knows which user placed it.

What is PostgreSQL?

PostgreSQL is an open source Relational Database Management System (RDBMS). The RDBMS is a software layer that sits on top of your data. It enforces rules, handles queries, and manages connections.

Tables and Schema

When you create a table in PostgreSQL, you define its schema, the column names and what type of data each column holds. You do this with `CREATE TABLE`

```
CREATE TABLE users (
    id    SERIAL PRIMARY KEY,
    name  VARCHAR(100) NOT NULL,
    email VARCHAR(255)
);
```

You have to define this schema otherwise, PostgreSQL will reject any data that doesn't fit the schema

Data Types

Every column has a data type that constrains what values it can store:

Type	What it stores	Example
INTEGER	Whole numbers	42

SERIAL	Auto-incrementing integers (use it for IDs)	1, 2, 3, ...
VARCHAR(n)	Text up to n characters	"Nishit Sharma"
TEXT	Unlimited length text	Long descriptions
BOOLEAN	True/False	true
DATE	Calendar date	2026-3-25
NUMERIC	Decimal numbers	19.99

Primary Keys and Foreign Keys

A primary key is a column that uniquely identifies every row in a table. No two rows can share the same primary key. `SERIAL PRIMARY KEY` makes PostgreSQL auto assign a new integer ID to each row you insert.

A foreign Key is a column in one table that references the primary key of another table. This creates the relationship between tables

```
CREATE TABLE orders (
    id          SERIAL PRIMARY KEY,
    user_id     INTEGER REFERENCES users(id),
    amount      NUMERIC,
    order_date  DATE
);
```

In this example, user_id is a foreign key, and it must match an existing id in the users table. If you try to insert an order with a user_id that doesn't exist, PostgreSQL will reject it automatically.

Basic SQL

SQL (Structured Query Language) is the language we will use. Every operation you will perform (creating tables, inserting data, and reading data) is written in SQL.

CREATE TABLE

Defines a new table and its schema:

```
CREATE TABLE resources (
    id            SERIAL PRIMARY KEY,
    name          VARCHAR(255) NOT NULL,
    category      VARCHAR(50),
    description   TEXT
);
```

`NOT NULL` means that the column is required, PostgreSQL won't let you insert a row without a value there.

INSERT (Create)

Adds a new row to a table:

```
INSERT INTO resources (name, category, description)
VALUES ('City Food Bank', 'Food', 'Free weekly groceries for families.');
```

SELECT (Read)

Retrieves rows from a table:

```
-- Get everything
SELECT * FROM resources;

-- Get specific columns
SELECT name, category FROM resources;

-- Filter with WHERE
SELECT * FROM resources WHERE category = 'Food';
```

UPDATE

Changes an existing row:

```
UPDATE resources
SET description = 'Updated description here.'
WHERE id = 1;
```

Make sure to include a `WHERE` clause on `UPDATE` otherwise you will update every row in the table

DELETE

Removes a row:

```
DELETE FROM resources WHERE id = 1;
```

Relationships between tables

The most common relationship in web apps is one-to-many. This means that one resource can have many referrals, one user can have many orders.

```
-- One user
CREATE TABLE users (
    id   SERIAL PRIMARY KEY,
    name VARCHAR(100)
);

-- Many orders belonging to one user
CREATE TABLE orders (
    id       SERIAL PRIMARY KEY,
    user_id  INTEGER REFERENCES users(id),
    amount   NUMERIC
);
```

To retrieve related data across two tables, you can use a `JOIN`

```
SELECT users.name, orders.amount
FROM orders
JOIN users ON orders.user_id = users.id;
```

This pulls the user's name from the `users` table and matches it to their orders using the foreign key relationship

Getting Started

Join the GitHub Classroom assignment with the following link:

https://classroom.github.com/a/yW_GReTh

Once you join, your repository will be created https://github.com/blueprint-learn/week3-techteam-sp26-{github-username}

Once you join, GitHub Classroom will automatically create your personal repository. You will receive an email confirming access.

After receiving access:

1. Clone your repository locally
2. Install dependencies
3. Run the starter FastAPI app

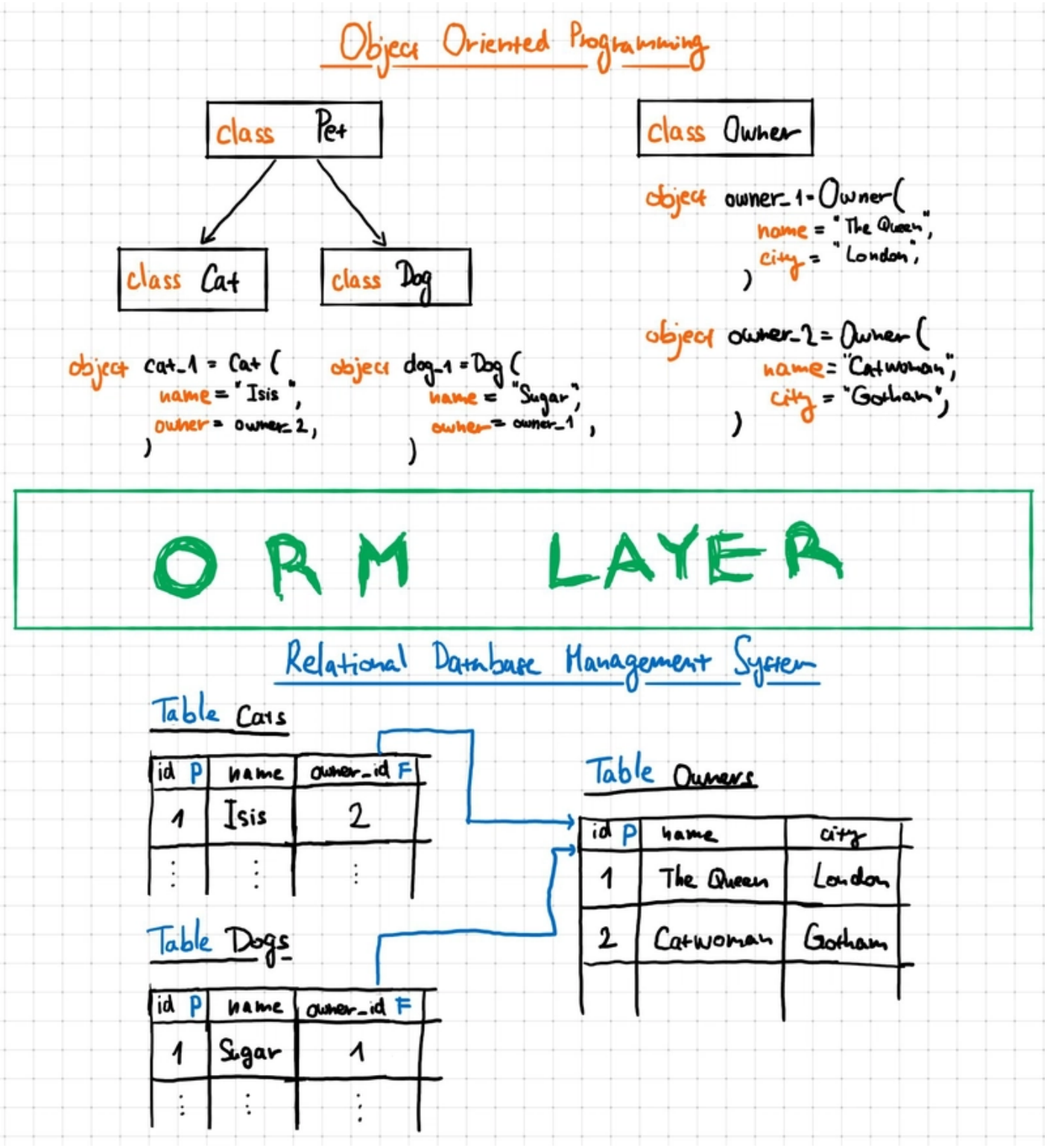
This repository contains the scaffolding you will extend during this week.

```
git clone https://github.com/blueprint-learn/week3-techteam-sp26-{github-username}.git
```

Week 4

What is an ORM?

Last week, we covered SQL and PostgreSQL. However, in many applications, developers don't want to write raw SQL for any usage of the database. An ORM, Object Relation Mapper, helps bridge objects and database tables by representing tables as classes and rows as objects.



Why an ORM?

An ORM gives us a code friendly way of working with data in tables with rows, columns, primary keys, and foreign keys. This is helpful because most backend applications aren't written in SQL, but it allows us to use whatever language we want, Python or Typescript, for example, while being able to use Postgres as the database.

What is SQLAlchemy?

SQLAlchemy is a Python ORM that we can import as a library. This lets us describe our database tables as a Python class and its columns as class attributes.

Models

In SQLAlchemy, a model is a Python class that represents a database table. Each model class has a `__tablename__` value and attributes that define columns, such as `id`, `name`, or `email`

```
from sqlalchemy import Column, Integer, String
from sqlalchemy.orm import declarative_base

Base = declarative_base()

class Resource(Base):
    __tablename__ = "resources"

    id = Column(Integer, primary_key=True)
    name = Column(String, nullable=False)
    category = Column(String, nullable=False)
    address = Column(String)
    contact_email = Column(String)
```

In this example, the `Resource` class maps to a table named `resources`, and each class attribute maps to one column in that table.

Columns and Data Types

Just like in PostgreSQL, each column in a SQLAlchemy model has a type (Integer, String, etc). You can also add constraints in the model definition. For example, `primary_key=True` marks a column as the primary key, and `nullable=False` means that the column is required and should not be left empty.

Primary Keys and Foreign Keys

A primary key identifies a row in a table, and a foreign key references the primary key of another table, which we learned last week. Here is what that looks like in SQLAlchemy

```
from sqlalchemy import Column, Integer, String, ForeignKey, Date
from sqlalchemy.orm import declarative_base

Base = declarative_base()

class Referral(Base):
```

```
    __tablename__ = "referrals"

    id = Column(Integer, primary_key=True)
    family_name = Column(String, nullable=False)
    resource_id = Column(Integer, ForeignKey("resources.id"))
    referral_date = Column(Date)
    notes = Column(String)
```

Here, `resource_id` is a foreign key that points to `resources.id`. That means that each referral can be linked to an existing resource, like we setup last week.

Relationships

Foreign keys connect tables at the database level, but SQLAlchemy can also define relationships at the Python level. Relationships make it easier to move between related objects in code, such as going from a resource to all of its referrals

```
from sqlalchemy.orm import relationship

class Resource(Base):
    __tablename__ = "resources"

    id = Column(Integer, primary_key=True)
    name = Column(String, nullable=False)

    referrals = relationship("Referral", back_populates="resource")

class Referral(Base):
    __tablename__ = "referrals"

    id = Column(Integer, primary_key=True)
    resource_id = Column(Integer, ForeignKey("resources.id"))

    resource = relationship("Resource", back_populates="referrals")
```

Connecting to PostgreSQL

Before SQLAlchemy can do anything, we need to connect it to our database.

```
from sqlalchemy import create_engine

# If you are using a hosting provider, they will give you this
```

```
engine = create_engine("postgresql://user:password@localhost/dbname")
```

Sessions

After SQLAlchemy knows how to connect to PostgreSQL, it needs a way to actually interact with the database. That is the role of a session.

A session is the object that manages database operations like adding rows, querying data, and saving changes. You can think of it as your temporary conversation with the database while your Python code is running.

```
from datetime import date
from sqlalchemy.orm import Session

with Session(engine) as session:
    food_bank = Resource(
        name="City Food Bank",
        category="Food",
        address="123 Main St",
        contact_email="help@cityfoodbank.org",
    )

    tutoring_program = Resource(
        name="Bright Futures Tutoring",
        category="Education",
        address="45 College Ave",
        contact_email="info@brightfutures.org",
    )

    session.add_all([food_bank, tutoring_program])
    session.commit()

    referral_1 = Referral(
        family_name="name1",
        resource_id=food_bank.id,
        referral_date=date(2026, 4, 1),
        notes="note1",
    )

    referral_2 = Referral(
        family_name="name2",
```

```
        resource_id=tutoring_program.id,
        referral_date=date(2026, 4, 2),
        notes="note2",
    )

    session.add_all([referral_1, referral_2])
    session.commit()
```

In this example, we create the session using `with Session(engine) as session:`. Inside that, we create `Resource` and `Referral` objects in Python, add them to the session, and save them to PostgreSQL with `Session.commit()`

Creating Tables

After you define your models, SQLAlchemy can use the models you defined to create the tables in the database, using `Base.metadata.create_all(engine)`. This is an example of why using an ORM is better. Instead of writing raw SQL, SQLAlchemy uses your Python models to create the matching tables

```
from sqlalchemy import Column, Integer, String, ForeignKey, Date, create_engine
from sqlalchemy.orm import declarative_base, relationship

Base = declarative_base()

class Resource(Base):
    __tablename__ = "resources"

    id = Column(Integer, primary_key=True)
    name = Column(String, nullable=False)
    category = Column(String, nullable=False)
    address = Column(String)
    contact_email = Column(String)

    referrals = relationship("Referral", back_populates="resource")

class Referral(Base):
    __tablename__ = "referrals"

    id = Column(Integer, primary_key=True)
    family_name = Column(String, nullable=False)
    resource_id = Column(Integer, ForeignKey("resources.id"))
    referral_date = Column(Date)
```

```
    notes = Column(String)

    resource = relationship("Resource", back_populates="referrals")

engine = create_engine("postgresql://postgres:postgres@localhost:5432/communitybridge")

Base.metadata.create_all(engine)
```

Querying Data

One of the main reasons to use an ORM is to query data using Python. SQLAlchemy sessions can retrieve all rows, filter rows, and follow relationships between models. For example

```
with Session(engine) as session:
    all_resources = session.query(Resource).all()
    food_resources = session.query(Resource).filter(Resource.category == "Food").all()
```

The first line gets every row in the `resources` table, and the second line gets only rows where the category is `"Food"`. SQLAlchemy turns those Python expressions into SQL queries behind the scenes.

You can also create and save new rows:

```
new_resource = Resource(
    name="City Food Bank",
    category="Food",
    address="123 Main St",
    contact_email="help@cityfoodbank.org"
)

session.add(new_resource)
session.commit()
```

Here, we create a Python object, add it to the session, and commit the change so it is saved to PostgreSQL

Getting Started

Join the GitHub Classroom assignment with the following link:

https://classroom.github.com/a/XIyvm9h_

Once you join, your repository will be created https://github.com/blueprint-learn/week4-techteam-sp26-{github-username}

Once you join, GitHub Classroom will automatically create your personal repository. You will receive an email confirming access.

After receiving access:

1. Clone your repository locally
2. Install dependencies
3. Run the starter FastAPI app

This repository contains the scaffolding you will extend during this week.

```
git clone https://github.com/blueprint-learn/week4-techteam-sp26-{github-username}.git
```

Week 5?

need to see if we can make a project that they can work on

So far, we talked a lot about frontend and backend as seperate pieces. The frontend is what the user interacts with, and the backend is what handles logic, data, and communication behind the scenes.

This week, we are taking a step back from focusing on just one thing. Instead, we are going to look at the whole system and how all the pieces connect.

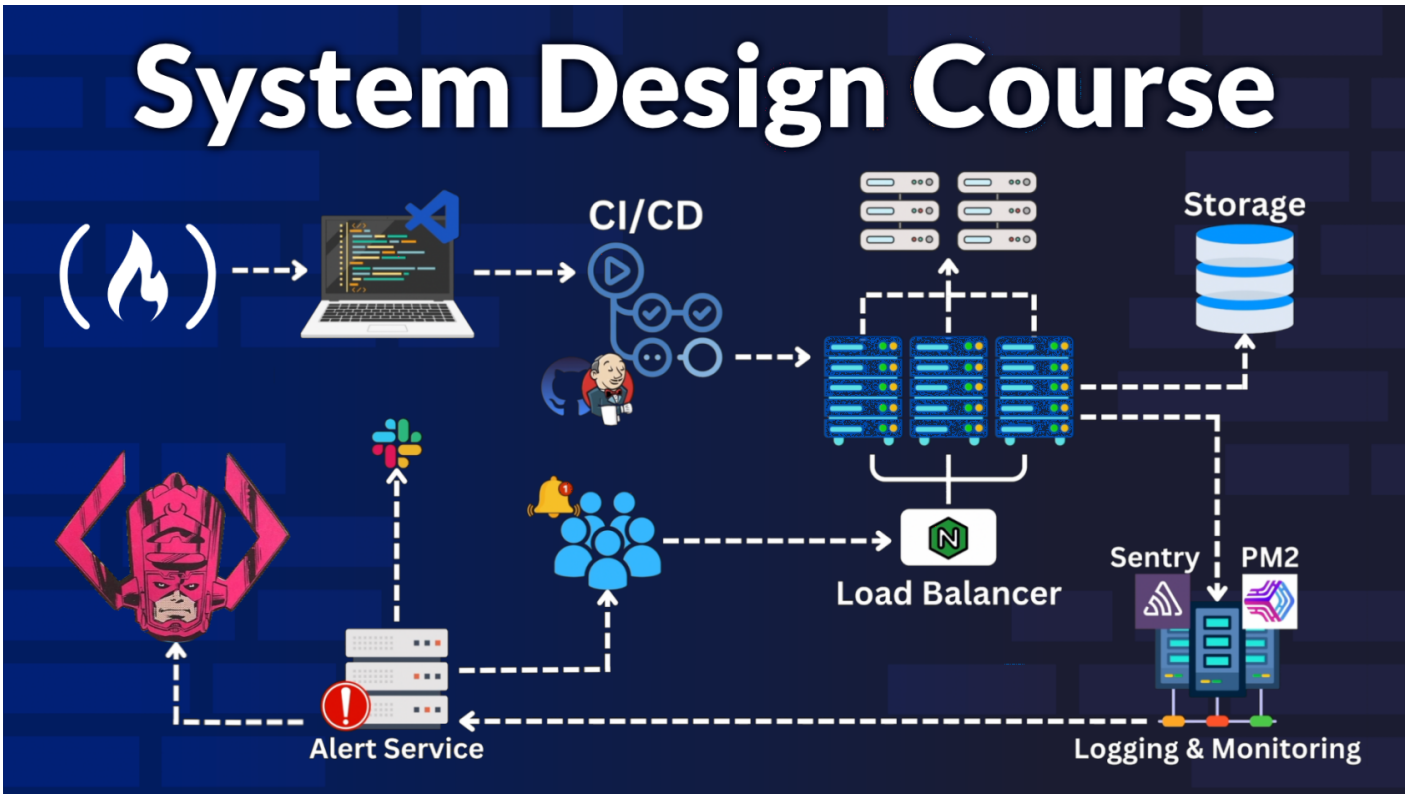
What is system design?

System design is just the process of thinking about how an application works as a whole. When we build an app, there are a bunch of questions beyond "How do I code this?"

You have to answer

1. What parts does this app need?
2. What does the data look like, and where will it go
3. How will the frontend interact with the backend
4. How do we deploy the app
5. What happens when tons of people use it

Answering these questions is pretty much what system design is. It isn't just for big companies or complicated apps. You have probably "done" system design if you've ever worked on a project, because anytime you have a frontend, backend, database, auth, file storage, or deployment, you implemented a system.



Thinking in System Design

The best way to think about system design is to break an app into pieces (and break each piece into smaller pieces if you can)

For most web apps, you usually need

1. Frontend
2. Backend
3. Database
4. Storage
5. Authentication
6. Hosting / Deployment

Not every project needs all of these, but these are the most common things most projects have.

For example, with DuckLink, they have

- A frontend where students browse events
- A backend that handles the requests
- A database that stores the event info
- Authentication so students can log in
- Hosting so that the app exists somewhere people can access

Once you think about apps like this, it becomes much easier to understand what you need to build

Usually, projects go through three primary stages during development: the local development stage, the deployment stage, and the scaling stage.

Local Development

This is where everything runs on your computer. You typically have

1. A frontend running locally (npm run dev)
2. A backend running locally (npm run dev)
3. A database running locally (Docker containers)

At this stage, you just want to make sure the app works properly

Deployment

Once your app runs properly locally, it needs to be hosted somewhere so people can access it (Either on the internet or an App Store). This means you need to think about

1. Where does the frontend live
2. Where does the backend live
3. Where does the database live
4. How can they all connect

Once a project gets deployed, system design matters a lot more because the app isn't just running in one place, its running like a system (system design).

Scaling

Let's say your app had tons of users showing up. Tons of problems will also start showing up. Things that were fine can become bottlenecks. Let's say

1. The backend might get too many requests
    1. Your site gets so much slower, especially if you rely on the backend to get data.
2. The database might slow down under load
    1. Similar to the backend getting too many requests, anything that requires retrieving information will take a while because your database is overloaded
3. File uploads might become expensive
    1. This can be applicable to the frontend and backend too, but let's say you hit the limit on your free plan for your file server. You will no longer be able to upload anything without paying, which, if scaled up to tons of users, can get expensive.
4. Users far away from the server have a slower experience
    1. Let's say you deployed everything to an American server. Users anywhere else in the world will have a slower time on your app because they have to interact with American servers to get everything. This takes a while, and while not as impactful as the other issues, it is still important to consider

This is where ideas like caching, CDNs, load balancing, and horizontal scaling matter. To further show this, let's take a look at an example

The YouTube Example

We asked this question during the Fall Semester recruitment cycle, and very few people could somewhat answer it, and only people who've had internships or tons of projects were able to somewhat answer it, so I figured this is one of the best examples we can do.

"How would you design YouTube?"

More specifically, though, we asked them, "What does your API look like?" How would you deploy this? How would you scale this?

However, for this, I'll walk you through the full thing so you guys can use this as an example if you build anything similar.

First, we need to define the APIs. You generally want to figure out your backend first, as it's harder to make changes to a backend if your frontend is already written. Let's say we are passing parameters from the frontend to the backend, and we need to change the backend; we also need to make changes to the frontend, which is annoying. Now, YouTube has a lot more than these four, but for the video streaming itself, we need a create video, delete video, update video, and get video. Create video takes the account that made the video, any metadata like title, description, thumbnail, and a UUID. A UUID stands for Universally Unique Identifier, and it's a 128-bit number that allows us to create the number before it touches the backend (reducing load) and makes it easier to merge databases as it's always unique (not stored as 1, 2, 3 in the db). Delete video takes the video UUID and the account to make sure whoever is trying to delete a video can actually delete the video. Update video would take similar parameters to create, but instead of creating a new video, you would just update the fields that need to be changed. The get video endpoint retrieves the video metadata that we stored before using the UUID that we provide. We can get the UUID from the frontend, which would request a list of videos from the backend and then map those results into React components. Get video would also return the video, either with a file or a playback URL, which I will explain later.

The frontend can look like whatever, so we don't need to focus on that in terms of system design, for now our working model in our heads can be three buttons, one for each api route, and a bunch of videos using the get video route. In a deployed app, we have our frontend connected to the backend, normally called a full-stack application. However, in this case, it isn't smart to store our video in our backend because the video needs to be streamed. Streaming a video from a database will slow down the database by a lot, and the streaming itself will be slow, so we need something else. For this, we need a file server to store our videos and our thumbnails. So in our mental model, the front end is connected to the backend, and the backend is connected to the file server.

When we store videos with the create video route, we store them in the file server and rename the video to the UUID we generated in the route. This allows us to identify which video corresponds to which entry in our database and retrieve it accordingly. One thing we can also do here to make uploads and downloads better is to have the backend create the UUID + db entry and return a presigned upload or download URL. Then the frontend can upload or download the video to/from your file storage instead of pushing the whole file through the backend.

Right now, the typical flow of this app is that the user goes to the frontend and interacts with the create video button. They upload the video and fill in whatever they need to, and submit the

request. The frontend then gives all the information to the backend, and the backend creates the UUID and stores everything except the video. The backend will return the presigned upload URL so the frontend can upload directly. Whenever the file needs to be streamed, the frontend sends a request to the backend, and the backend gives a presigned download URL to the frontend so it can stream properly, and the backend isn't involved in the actual streaming portion.

Now this is the first development stage done, and everything theoretically works locally, but how would we actually get to deploying this project? For this, we need hosting providers for each thing: the frontend, backend, and file server. For our frontend, we can use Vercel, Cloudflare Pages, AWS Amplify, or an AWS S3 bucket with CloudFront (I think our dev team uses this). For our backend, we would typically use PostgreSQL to store the information, so we can use Supabase, Neon, AWS RDS, or, if you don't want to use PostgreSQL, you can use whatever you can store stuff with (I suggest Convex if you don't want to mess with SQL). For our file server, you can use Vercel Blob, AWS S3, Cloudflare Stream, or Cloudflare R2.

Technically, this config can work. However, a question you may be asked is how would you scale this to a million users in a hypothetical scenario? In our model right now, it is in a straight line. One frontend instance, one backend instance, and one file server instance. We need to scale these horizontally, essentially adding more instances and creating a distributed network. This is where things can get tricky, so I'll try to explain it as simply as possible.

We essentially need multiple instances of each thing, however it is important that the instances for the backend and file server are connected. The frontend is easier to scale because it is mostly static and can be pushed through a CDN, but the backend, database, and file storage need a shared consistent state, otherwise some people will have videos that other people don't have access to.

First, let's address the frontend. We typically want to use a CDN for our multiple frontend instances. A Content Delivery Network allows us to store our frontend on tons of servers instead of just one server. The users will access the server closest to them, making sure they always have the fastest connection.

For our backend and file server, we will use something called a load balancer to distribute the requests effectively. The load balancer is pretty self-explanatory. If a cluster is running slower, the load balancer will reroute the request to a different cluster that has less load. To properly explain everything with examples, we will use Vercel for our frontend, Supabase for our backend, and an S3 bucket + CloudFront for our file server. Vercel has a CDN already, and it is pretty decent, so we don't have to worry too much about that.

For our backend, there is one thing we need to specify. Our backend must be stateless. Pretty much, no user session data (like which account they are on, what video they're watching, etc) can be stored on the database and instead, stored in a session using Redis and Tanstack Query (used to be called React Query). More specifically, Redis would be our shared cache layer on the backend, and React Query would help cache and manage requests on the frontend side. Setting up

our backend this way allows our load balancer to switch clusters without the user noticing, and using Redis + React Query allows us to use our cache (locally and from the db) instead of calling to the db all the time, making our performance as fast as possible.

Supabase has a load balancer built into it called the Supavisor that manages connections to different instances (called the Pool Size) so we can use that alongside the load balancer for our clusters. One trick you can do is have a couple of instances of the database, just for writing data, and a lot more for reading data. Supabase calls this Read Replicas. Most people on an app like YouTube don't really post and instead watch, so that's a way you can lessen the load on your system. Supavisor manages the database connections while the Read Replicas distribute read-heavy traffic.

We use S3 and CloudFront for the file server because it takes the load of streaming away from our backend and frontend. If we didn't use the presigned URLs, the frontend would still play a role in streaming. Also, CloudFront is an example of a CDN, so you could also use this method (S3 + CloudFront) to deploy the frontend.

One very minor thing, but usually on a bigger scale, we would also transcode the uploaded video into multiple resolutions so users with different internet speeds can stream smoothly. Other than that, with this setup, this is how you would design YouTube

Tradeoffs

Ok, I know that was a lot, but this is only one answer to this question. With system design, there are different answers to the same question, but as long as you have an answer, that is fine. Because there are multiple answers, every answer has certain tradeoffs. For instance

- Database Type (SQL vs NoSQL)
- Local file storage or cloud storage
- Monolith vs Seperate services
- Simpler setup now vs more scalable setup later

In our version of the YouTube answer. We are

- Locked into the Vercel Edge Network (Their CDN)
- Locked into Supabase and Supabase utils for our database
  - If Supabase goes down, so does our app
- Setting up our file storage is more difficult to set up compared to our frontend and backend, but we have more control over it

A lot of system design is making reasonable choices based on the size and needs of a project. Maybe there is a service we can use to store our files (uploadThing), so for an early prototype, we can use that instead of using an S3 bucket + CloudFront. Similarly, if we need control over our CDNs and database, we would move away from Vercel and Supabase.

??? Example

Walk me through how you would build ???