Week 4: HTTP, APIs & FastAPI

CS 203: Software Tools and Techniques for AI

Prof. Nipun Batra, IIT Gandhinagar

This Week's Journey

Part 1: How the Web Works

- HTTP methods, status codes, headers
- CLI tools: curl, httpie, jq

Part 2: Building APIs with FastAPI

- Setup and Hello World
- Request/response models with Pydantic
- File uploads and different response types

Part 3: Git Fundamentals

- Version control basics
- Collaboration workflows

What is HTTP?

HyperText Transfer Protocol - the foundation of web communication

Key Concepts:

- Client-Server model
- Request-Response cycle
- Stateless protocol
- Text-based protocol

Example Flow:

1. Browser (client) sends HTTP request
2. Server processes request
3. Server sends HTTP response
4. Browser renders content

HTTP Methods (Verbs)

Method	Purpose	Example Use
GET	Retrieve data	Fetch user profile
POST	Create new resource	Register new user
PUT	Update entire resource	Replace user data
PATCH	Partial update	Update email only
DELETE	Remove resource	Delete account

Safe methods: GET (doesn't modify data)

Idempotent methods: GET, PUT, DELETE (same result if repeated)

HTTP Status Codes

Success (2xx)

- 200 OK - Request succeeded
- 201 Created - Resource created
- 204 No Content - Success, no data to return

Client Errors (4xx)

- 400 Bad Request - Invalid syntax
- 401 Unauthorized - Authentication required
- 404 Not Found - Resource doesn't exist

Server Errors (5xx)

- 500 Internal Server Error - Server crashed

Anatomy of HTTP Request

```
POST /api/users HTTP/1.1
Host: example.com
Content-Type: application/json
Authorization: Bearer token123

{
  "name": "Alice",
  "email": "alice@example.com"
}
```

Components:

1. Request line: Method + Path + Protocol
2. Headers: Metadata (content type, auth, etc.)
3. Body: Data payload (for POST/PUT)

Anatomy of HTTP Response

```
HTTP/1.1 201 Created
Content-Type: application/json
Location: /api/users/42

{
  "id": 42,
  "name": "Alice",
  "email": "alice@example.com",
  "created_at": "2025-12-08T10:30:00Z"
}
```

Components:

1. Status line: Protocol + Status code + Message
2. Headers: Metadata about response
3. Body: Actual data returned

HTTP Headers - Request

Common Request Headers:

```
Accept: application/json
Content-Type: application/json
Authorization: Bearer eyJhbG ...
User-Agent: Mozilla/5.0
Cookie: session_id=abc123
```

- **Accept**: What formats client can handle
- **Content-Type**: Format of request body
- **Authorization**: Credentials
- **User-Agent**: Client software info
- **Cookie**: Session/state data

HTTP Headers - Response

Common Response Headers:

```
Content-Type: application/json
Content-Length: 1234
Set-Cookie: session_id=xyz789
Cache-Control: max-age=3600
Access-Control-Allow-Origin: *
```

- **Content-Type**: Format of response body
- **Content-Length**: Size in bytes
- **Set-Cookie**: Store data on client
- **Cache-Control**: Caching instructions
- **CORS headers**: Cross-origin permissions

JSON: The Language of APIs

JavaScript Object Notation - human-readable data format

```
{
  "user": {
    "id": 123,
    "name": "Alice",
    "email": "alice@example.com",
    "roles": ["user", "admin"],
    "active": true,
    "metadata": null
  }
}
```

Supported types:

- Objects: `{}`
- Arrays: `[]`

Meet curl - The HTTP Swiss Army Knife

Basic GET request:

```
curl https://api.github.com/users/octocat
```

GET with headers:

```
curl -H "Authorization: Bearer token123" \
     https://api.example.com/users
```

POST with JSON:

```
curl -X POST https://api.example.com/users \
     -H "Content-Type: application/json" \
     -d '{"name": "Alice", "email": "alice@example.com"}'
```

curl - Advanced Usage

Save response to file:

```
curl -o output.json https://api.github.com/users/octocat
```

Show response headers:

```
curl -i https://api.github.com/users/octocat
```

Follow redirects:

```
curl -L https://short.url/abc123
```

Upload file:

```
curl -F "file=@document.pdf" https://api.example.com/upload
```

Meet jq - JSON Processor

Pretty print JSON:

```
curl https://api.github.com/users/octocat | jq
```

Extract specific field:

```
curl https://api.github.com/users/octocat | jq '.name'
# Output: "The Octocat"
```

Filter array:

```
curl https://api.github.com/users/octocat/repos | jq '.[].name'
```

Select with condition:

jq - Building New Objects

Transform structure:

```
jq '{username: .login, profile: .html_url}' user.json
```

Array of transformed objects:

```
jq '[.[] | {name: .name, stars: .stargazers_count}]' repos.json
```

Calculations:

```
jq '[.[] | .price] | add' products.json  # Sum prices
jq '[.[] | .age] | max' users.json       # Find maximum
```

Meet HTTPie - curl's Friendly Cousin

Simpler syntax than curl:

```
# GET request
http https://api.github.com/users/octocat

# POST JSON (automatic)
http POST https://api.example.com/users \
  name="Alice" email="alice@example.com"

# Custom headers
http https://api.example.com/data \
  Authorization:"Bearer token123"
```

Advantages:

- Automatic JSON formatting
- Syntax highlighting

HTTPie - More Examples

Download file:

```
http --download https://example.com/file.pdf
```

Form submission:

```
http --form POST https://example.com/upload \
  file@document.pdf description="My doc"
```

Session persistence:

```
http --session=user1 POST https://api.example.com/login \
  username="alice" password="secret"

http --session=user1 GET https://api.example.com/profile
```

Testing APIs - Practical Workflow

1. Explore API:

```
http https://jsonplaceholder.typicode.com/posts/1
```

2. Extract data with jq:

```
http https://jsonplaceholder.typicode.com/posts | \
  jq '[.[] | {id, title}]'
```

3. Create new resource:

```
http POST https://jsonplaceholder.typicode.com/posts \
  title="My Post" body="Content here" userId=1
```

4. Update resource:

Why Build APIs?

Real-world scenarios:

- Mobile app backend
- Microservices architecture
- Third-party integrations
- Data pipeline endpoints
- ML model serving

Benefits:

- Separation of concerns
- Multiple clients (web, mobile, CLI)
- Easier testing and scaling
- Language-agnostic

Why FastAPI?

Modern Python web framework for APIs

Key Features:

- Fast performance (matches Node.js, Go)
- Automatic API documentation
- Built-in data validation (Pydantic)
- Type hints throughout
- Async support
- Easy to learn

Popular alternatives: Flask, Django REST, Tornado

FastAPI Setup

Installation:

```
pip install fastapi uvicorn[standard]
```

Hello World (`main.py`):

```
from fastapi import FastAPI

app = FastAPI()

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

Run server:

Your First API Endpoint

Code:

```
from fastapi import FastAPI

app = FastAPI()

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

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

Test:

```
curl http://localhost:8000/
```

Path Parameters

Dynamic URL segments:

```python
@app.get("/users/{user_id}")
def get_user(user_id: int):
    return {"user_id": user_id, "name": f"User {user_id}"}

@app.get("/posts/{post_id}/comments/{comment_id}")
def get_comment(post_id: int, comment_id: int):
    return {
        "post_id": post_id,
        "comment_id": comment_id
    }
```

Type validation automatic:

- `/users/abc` returns 422 error
- `/users/42` works correctly

Query Parameters

URL parameters after `?`:

```
@app.get("/items")
def list_items(skip: int = 0, limit: int = 10):
    return {
        "skip": skip,
        "limit": limit,
        "items": ["item1", "item2"]
    }
```

Usage:

```
curl "http://localhost:8000/items?skip=5&limit=20"
```

Optional parameters:

Request Body with Pydantic

Define data model:

```
from pydantic import BaseModel

class User(BaseModel):
    name: str
    email: str
    age: int

@app.post("/users")
def create_user(user: User):
    return {
        "message": "User created",
        "user": user
    }
```

Test:

Pydantic Validation

Field constraints:

```python
from pydantic import BaseModel, Field, EmailStr

class User(BaseModel):
    name: str = Field( ... , min_length=1, max_length=50)
    email: EmailStr
    age: int = Field( ... , ge=0, le=150)
    bio: Optional[str] = None

@app.post("/users")
def create_user(user: User):
    return user
```

Invalid data returns 422 with detailed errors

Pydantic - Advanced Validation

Custom validators:

```
from pydantic import BaseModel, validator

class Product(BaseModel):
    name: str
    price: float
    discount: float = 0.0

    @validator('discount')
    def discount_valid(cls, v, values):
        if v < 0 or v > 0.5:
            raise ValueError('Discount must be 0-50%')
        return v

    @property
    def final_price(self):
        return self.price * (1 - self.discount)
```

Response Models

Specify return type:

```
class UserIn(BaseModel):
    username: str
    password: str
    email: str

class UserOut(BaseModel):
    username: str
    email: str

@app.post("/users", response_model=UserOut)
def create_user(user: UserIn):
    # Process user (hash password, etc.)
    return user  # Password automatically excluded
```

Benefits:

Status Codes

Explicit status codes:

```
from fastapi import status

@app.post("/users", status_code=status.HTTP_201_CREATED)
def create_user(user: User):
    return user

@app.delete("/users/{user_id}", status_code=status.HTTP_204_NO_CONTENT)
def delete_user(user_id: int):
    # Delete user
    return
```

Common codes:

- `200` - HTTP_200_OK
- `201` - HTTP_201_CREATED

Error Handling

Raise HTTP exceptions:

```
from fastapi import HTTPException

users_db = {1: {"name": "Alice"}, 2: {"name": "Bob"}}

@app.get("/users/{user_id}")
def get_user(user_id: int):
    if user_id not in users_db:
        raise HTTPException(
            status_code=404,
            detail=f"User {user_id} not found"
        )
    return users_db[user_id]
```

Response:

File Uploads

Single file:

```
from fastapi import File, UploadFile

@app.post("/upload")
async def upload_file(file: UploadFile = File( ... )):
    contents = await file.read()
    return {
        "filename": file.filename,
        "content_type": file.content_type,
        "size": len(contents)
    }
```

Test:

```
curl -F "file=@document.pdf" http://localhost:8000/upload
```

Multiple File Uploads

Accept multiple files:

```
from typing import List

@app.post("/uploadfiles")
async def upload_files(files: List[UploadFile] = File( ... )):
    results = []
    for file in files:
        contents = await file.read()
        results.append({
            "filename": file.filename,
            "size": len(contents)
        })
    return {"files": results}
```

Test:

Saving Uploaded Files

Save to disk:

```
import shutil
from pathlib import Path

@app.post("/upload")
async def upload_file(file: UploadFile = File( ... )):
    upload_dir = Path("uploads")
    upload_dir.mkdir(exist_ok=True)

    file_path = upload_dir / file.filename

    with file_path.open("wb") as buffer:
        shutil.copyfileobj(file.file, buffer)

    return {
        "filename": file.filename,
        "location": str(file_path)
```

Returning Files

Send file as response:

```
from fastapi.responses import FileResponse

@app.get("/download/{filename}")
def download_file(filename: str):
    file_path = Path("uploads") / filename

    if not file_path.exists():
        raise HTTPException(status_code=404, detail="File not found")

    return FileResponse(
        path=file_path,
        filename=filename,
        media_type='application/octet-stream'
    )
```

Different Response Types

HTML Response:

```
from fastapi.responses import HTMLResponse

@app.get("/", response_class=HTMLResponse)
def read_root():
    return """
    <html>
        <body>
            <h1>Hello World</h1>
        </body>
    </html>
    """
```

Plain Text:

```
from fastapi.responses import PlainTextResponse
```

Streaming Responses

Stream large data:

```
from fastapi.responses import StreamingResponse
import io

@app.get("/large-file")
def stream_file():
    def generate():
        for i in range(1000):
            yield f"Line {i}\n"

    return StreamingResponse(
        generate(),
        media_type="text/plain"
    )
```

Use for:

Automatic API Documentation

FastAPI generates docs automatically!

Swagger UI:

- Visit: `http://localhost:8000/docs`
- Interactive API testing
- Try requests directly in browser

ReDoc:

- Visit: `http://localhost:8000/redoc`
- Alternative documentation style
- Better for reading

No extra work needed - just write code!

CORS - Cross-Origin Requests

Allow frontend from different domain:

```
from fastapi.middleware.cors import CORSMiddleware

app = FastAPI()

app.add_middleware(
    CORSMiddleware,
    allow_origins=["http://localhost:3000"],
    allow_credentials=True,
    allow_methods=["*"],
    allow_headers=["*"],
)
```

Needed when:

- Frontend on different port/domain

Dependency Injection

Reusable components:

```
from fastapi import Depends

def get_db():
    db = Database()
    try:
        yield db
    finally:
        db.close()

@app.get("/users")
def get_users(db = Depends(get_db)):
    return db.query_users()
```

Common uses:

Background Tasks

Run tasks after response:

```python
from fastapi import BackgroundTasks

def send_email(email: str, message: str):
    # Send email (slow operation)
    print(f"Sending email to {email}")

@app.post("/register")
def register(
    email: str,
    background_tasks: BackgroundTasks
):
    background_tasks.add_task(send_email, email, "Welcome!")
    return {"message": "User registered"}
```

Response sent immediately, email sent in background

Environment Variables

Configuration management:

```
from pydantic_settings import BaseSettings

class Settings(BaseSettings):
    app_name: str = "My API"
    database_url: str
    api_key: str

    class Config:
        env_file = ".env"

settings = Settings()

@app.get("/info")
def info():
    return {"app_name": settings.app_name}
```

API Versioning

Version your endpoints:

```
from fastapi import APIRouter

v1_router = APIRouter(prefix="/v1")
v2_router = APIRouter(prefix="/v2")

@v1_router.get("/users")
def get_users_v1():
    return {"version": "1", "users": []}

@v2_router.get("/users")
def get_users_v2():
    return {"version": "2", "users": [], "total": 0}

app.include_router(v1_router)
app.include_router(v2_router)
```

Putting It All Together

Complete example:

```
from fastapi import FastAPI, HTTPException
from pydantic import BaseModel

app = FastAPI(title="Unit Converter API")

class ConversionRequest(BaseModel):
    value: float
    from_unit: str
    to_unit: str

@app.post("/convert")
def convert(req: ConversionRequest):
    conversions = {
        ("celsius", "fahrenheit"): lambda x: x * 9/5 + 32,
        ("fahrenheit", "celsius"): lambda x: (x - 32) * 5/9,
    }

    key = (req.from_unit.lower(), req.to_unit.lower())
    if key not in conversions:
        raise HTTPException(status_code=400, detail="Conversion not supported")
```

Testing Your API

Using pytest:

```
from fastapi.testclient import TestClient

client = TestClient(app)

def test_read_root():
    response = client.get("/")
    assert response.status_code == 200
    assert response.json() == {"message": "Hello World"}

def test_create_user():
    response = client.post("/users", json={
        "name": "Alice",
        "email": "alice@example.com"
    })
    assert response.status_code == 201
    assert response.json()["name"] == "Alice"
```

Async/Await in FastAPI

For I/O-bound operations:

```
import httpx

@app.get("/fetch")
async def fetch_data():
    async with httpx.AsyncClient() as client:
        response = await client.get("https://api.github.com/users/octocat")
        return response.json()

@app.post("/process")
async def process_file(file: UploadFile):
    contents = await file.read()
    # Process contents
    return {"size": len(contents)}
```

Use async when:

Common Pitfalls

1. Not using response models

- Exposes sensitive data (passwords, etc.)
- Solution: Use `response_model`

2. Blocking operations in async functions

- Don't use `requests` in async functions
- Solution: Use `httpx` or run in thread pool

3. Missing error handling

- Unhandled exceptions crash server
- Solution: Use try/except and HTTPException

4. Not validating input

Best Practices

1. Structure your project:

```
myapi/
├── main.py
├── models.py
├── routers/
│   ├── users.py
│   └── items.py
├── dependencies.py
└── tests/
```

2. Use routers for organization

3. Keep endpoints focused (single responsibility)

4. Document with docstrings

5. Use type hints everywhere

6. Write tests

API Design Principles

1. RESTful conventions:

- Use nouns for resources (`/users`, not `/getUsers`)
- Use HTTP methods correctly (GET, POST, PUT, DELETE)
- Use appropriate status codes

2. Consistency:

- Naming conventions
- Response formats
- Error handling

3. Versioning:

- Plan for changes

What We've Learned

HTTP Fundamentals:

- Methods, status codes, headers
- Request/response structure
- JSON data format

CLI Tools:

- curl for HTTP requests
- jq for JSON processing
- httpie for friendly API testing

FastAPI:

- Building REST APIs

Questions?

Next session: Git fundamentals and API integration