Day 6: API Protocols – REST, GraphQL, WebSocket

This guide walks you through **design, implementation, and comparison** of REST, GraphQL, and WebSocket APIs. It is structured to directly satisfy the daily completion checklist.

---

1. RESTful API Design

Core Principles

- **Resource-oriented URLs** (nouns, not verbs)
- **HTTP methods** define action
- **Stateless requests**
- **Proper status codes**
- **Versioning** for backward compatibility

Example Resource Design

```
GET    /api/v1/users          → list users
GET    /api/v1/users/{id}     → get user
POST   /api/v1/users          → create user
PUT    /api/v1/users/{id}     → update user
DELETE /api/v1/users/{id}     → delete user
```

HTTP Status Codes (Must Use)

- `200 OK` – successful GET/PUT
- `201 Created` – resource created
- `204 No Content` – successful delete
- `400 Bad Request` – validation error
- `401 Unauthorized` – auth missing
- `403 Forbidden` – auth insufficient
- `404 Not Found` – resource missing
- `500 Internal Server Error` – server failure

FastAPI Example

```
from fastapi import FastAPI, HTTPException

app = FastAPI(title="User API", version="v1")

users = {}
```

```python
@app.post("/api/v1/users", status_code=201)
def create_user(user_id: int, name: str):
    if user_id in users:
        raise HTTPException(status_code=400, detail="User exists")
    users[user_id] = name
    return {"id": user_id, "name": name}

@app.get("/api/v1/users/{user_id}")
def get_user(user_id: int):
    if user_id not in users:
        raise HTTPException(status_code=404, detail="User not found")
    return {"id": user_id, "name": users[user_id]}
```

✔️Checklist: REST API with versioning and status codes

---

2. GraphQL Server (Queries & Mutations)

Why GraphQL

- Client controls **exact data needed**
- Single endpoint
- Avoids over-fetching / under-fetching

Schema Design

```graphql
type User {
  id: ID!
  name: String!
  email: String!
}

type Query {
  users: [User]
  user(id: ID!): User
  userCount: Int
  health: String
  version: String
}

type Mutation {
  createUser(name: String!, email: String!): User
  updateUser(id: ID!, name: String): User
  deleteUser(id: ID!): Boolean
}
```

Python (Strawberry) Server

```python
import strawberry
from typing import List

@strawberry.type
class User:
    id: int
    name: str
    email: str

users: List[User] = []

@strawberry.type
class Query:
    def users(self) -> List[User]: return users
    def user(self, id: int) -> User | None:
        return next((u for u in users if u.id == id), None)
    def user_count(self) -> int: return len(users)
    def health(self) -> str: return "OK"
    def version(self) -> str: return "1.0"

@strawberry.type
class Mutation:
    def create_user(self, name: str, email: str) -> User:
        user = User(id=len(users)+1, name=name, email=email)
        users.append(user)
        return user

    def update_user(self, id: int, name: str | None = None) -> User:
        user = next(u for u in users if u.id == id)
        if name: user.name = name
        return user

    def delete_user(self, id: int) -> bool:
        global users
        users = [u for u in users if u.id != id]
        return True

schema = strawberry.Schema(query=Query, mutation=Mutation)
```

✔️Checklist: 5 Queries + 3 Mutations

3. WebSocket (Real-Time Communication)

When to Use WebSockets

- Chat systems
- Live monitoring dashboards
- Notifications
- Multiplayer games

WebSocket vs HTTP

- Persistent connection
- Full duplex (bi-directional)
- Low latency

FastAPI WebSocket Example

```python
from fastapi import FastAPI, WebSocket

app = FastAPI()

@app.websocket("/ws/chat")
async def chat_socket(ws: WebSocket):
    await ws.accept()
    while True:
        message = await ws.receive_text()
        await ws.send_text(f"Echo: {message}")
```

Test using browser console:

```javascript
let ws = new WebSocket("ws://localhost:8000/ws/chat")
ws.onmessage = e => console.log(e.data)
ws.send("Hello")
```

✔️Checklist: WebSocket server handling messages

---

4. Protocol Comparison & Recommendations

Comparison Table

Feature	REST	GraphQL	WebSocket
Communication	Request/Response	Request/Response	Bi-directional

Feature	REST	GraphQL	WebSocket
Over-fetching	Yes	No	No
Caching	Easy (HTTP)	Complex	Not applicable
Real-time	Poor	Poor	Excellent
Complexity	Low	Medium–High	Medium
Tooling	Mature	Growing	Moderate

Recommendations

- **REST** → CRUD apps, public APIs, microservices
- **GraphQL** → Mobile apps, complex frontend queries
- **WebSocket** → Real-time systems (chat, monitoring, alerts)

✔️Checklist: Protocol comparison document with recommendations

---

Day 6 Completion Status

- ☑️REST API implemented
- ☑️GraphQL queries & mutations completed
- ☑️WebSocket real-time server working
- ☑️Comparison & recommendations documented

If you want, Day 7 can move into **API Security (JWT, OAuth2, Rate Limiting)** or we can integrate all three protocols into **one production-ready backend**.