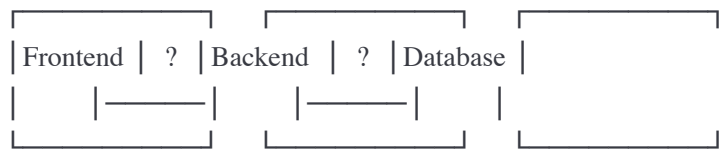
Chapter 9: APIs and Communication Protocols

Introduction: Why APIs Matter

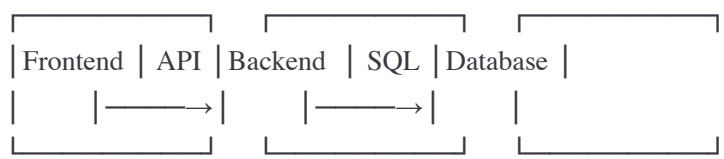
APIs (Application Programming Interfaces) are how systems talk to each other.

Without APIs:



No standard way to communicate!

With APIs:



Clear contract, documented, predictable!

---

1. REST API Design Principles

REST (Representational State Transfer) - The most common API style.

Core Principles

1. Resource-Based URLs

Concept: URLs represent resources (nouns), not actions (verbs).

❌ BAD (Action-based):

```
GET  /getUsers
GET  /getUserById?id=123
POST /createUser
POST /updateUser
POST /deleteUser
```

✓ GOOD (Resource-based):

```
GET    /users          # List all users
GET    /users/123      # Get specific user
POST   /users          # Create user
PUT    /users/123      # Update user (full)
PATCH  /users/123      # Update user (partial)
DELETE /users/123      # Delete user
```

URL is a noun (users), HTTP method is the verb!

Nested Resources:

```
GET    /users/123/posts          # User's posts
GET    /users/123/posts/456      # Specific post
POST   /users/123/posts          # Create post for user
DELETE /users/123/posts/456      # Delete user's post

GET    /posts/456/comments       # Post's comments
POST   /posts/456/comments       # Add comment to post
```

---

2. HTTP Methods (Verbs)

Method	Operation	Safe?	Idempotent?
GET	Read	Yes	Yes
POST	Create	No	No
PUT	Replace	No	Yes
PATCH	Update	No	No*
DELETE	Remove	No	Yes
HEAD	Headers only	Yes	Yes
OPTIONS	Get options	Yes	Yes

Safe: Doesn't modify data

Idempotent: Same result if called multiple times

*PATCH can be designed to be idempotent

Examples:

```
http
```

```
# GET - Read (Safe, Idempotent)
GET /users/123 HTTP/1.1
Host: api.example.com

# Call it 100 times, same result, no side effects

# POST - Create (Not Idempotent)
POST /users HTTP/1.1
Host: api.example.com
Content-Type: application/json

{
  "name": "John Doe",
  "email": "john@example.com"
}

# Call it twice → Creates 2 users!

# PUT - Replace (Idempotent)
PUT /users/123 HTTP/1.1
Host: api.example.com
Content-Type: application/json

{
  "name": "John Doe Updated",
  "email": "john.updated@example.com"
}

# Call it 100 times → Same final state

# PATCH - Partial Update
PATCH /users/123 HTTP/1.1
Host: api.example.com
Content-Type: application/json

{
  "email": "newemail@example.com"
}

# Only updates specified fields

# DELETE - Remove (Idempotent)
DELETE /users/123 HTTP/1.1
Host: api.example.com
```

```
# Call it 100 times → User deleted (same result)
```

---

3. HTTP Status Codes

Code	Meaning
2xx	SUCCESS
200	OK (general success)
201	Created (resource created)
204	No Content (success, no response body)
3xx	REDIRECTION
301	Moved Permanently
302	Found (temporary redirect)
304	Not Modified (cached version ok)
4xx	CLIENT ERROR
400	Bad Request (invalid input)
401	Unauthorized (not authenticated)
403	Forbidden (authenticated but no access)
404	Not Found
409	Conflict (duplicate, version mismatch)
422	Unprocessable Entity (validation fail)
429	Too Many Requests (rate limited)
5xx	SERVER ERROR
500	Internal Server Error
502	Bad Gateway
503	Service Unavailable
504	Gateway Timeout

---

4. Request/Response Format

Consistent JSON Structure:

```
javascript
```

```
// Success Response
{
  "data": {
    "id": 123,
    "name": "John Doe",
    "email": "john@example.com"
  },
  "meta": {
    "timestamp": "2024-01-20T10:00:00Z",
    "version": "1.0"
  }
}

// List Response with Pagination
{
  "data": [
    {"id": 1, "name": "User 1"},
    {"id": 2, "name": "User 2"}
  ],
  "pagination": {
    "page": 1,
    "pageSize": 20,
    "totalPages": 50,
    "totalItems": 1000
  },
  "links": {
    "self": "/users?page=1",
    "next": "/users?page=2",
    "prev": null,
    "first": "/users?page=1",
    "last": "/users?page=50"
  }
}

// Error Response
{
  "error": {
    "code": "VALIDATION_ERROR",
    "message": "Invalid email format",
    "details": [
      {
        "field": "email",
        "issue": "must be valid email address"
      }
    ]
  },
```

```
  "meta": {
    "timestamp": "2024-01-20T10:00:00Z",
    "requestId": "abc-123-def-456"
  }
}
```

Complete REST API Example (Node.js/Express)

javascript

```
const express = require('express');
const app = express();

app.use(express.json());

// In-memory database (for demonstration)
let users = [
  { id: 1, name: 'John Doe', email: 'john@example.com' },
  { id: 2, name: 'Jane Smith', email: 'jane@example.com' }
];
let nextId = 3;

// 1. LIST - Get all users
app.get('/api/v1/users', (req, res) => {
  // Pagination
  const page = parseInt(req.query.page) || 1;
  const limit = parseInt(req.query.limit) || 10;
  const startIndex = (page - 1) * limit;
  const endIndex = page * limit;

  const paginatedUsers = users.slice(startIndex, endIndex);

  res.json({
    data: paginatedUsers,
    pagination: {
      page,
      limit,
      total: users.length,
      totalPages: Math.ceil(users.length / limit)
    }
  });
});

// 2. GET - Get single user
app.get('/api/v1/users/:id', (req, res) => {
  const user = users.find(u => u.id === parseInt(req.params.id));

  if (!user) {
    return res.status(404).json({
      error: {
        code: 'NOT_FOUND',
        message: 'User not found'
      }
    });
  }
```

```
  res.json({ data: user });
});

// 3. CREATE - Create new user
app.post('/api/v1/users', (req, res) => {
  const { name, email } = req.body;

  // Validation
  if (!name || !email) {
    return res.status(400).json({
      error: {
        code: 'VALIDATION_ERROR',
        message: 'Name and email are required',
        details: [
          { field: 'name', issue: 'required' },
          { field: 'email', issue: 'required' }
        ]
      }
    });
  }

  // Check for duplicate email
  if (users.find(u => u.email === email)) {
    return res.status(409).json({
      error: {
        code: 'CONFLICT',
        message: 'User with this email already exists'
      }
    });
  }

  const newUser = {
    id: nextId++,
    name,
    email
  };

  users.push(newUser);

  res.status(201)
     .location(`/api/v1/users/${newUser.id}`)
     .json({ data: newUser });
});

// 4. UPDATE - Full update (PUT)
app.put('/api/v1/users/:id', (req, res) => {
  const user = users.find(u => u.id === parseInt(req.params.id));
```

```
  if (!user) {
    return res.status(404).json({
      error: {
        code: 'NOT_FOUND',
        message: 'User not found'
      }
    });
  }

  const { name, email } = req.body;

  // PUT requires all fields
  if (!name || !email) {
    return res.status(400).json({
      error: {
        code: 'VALIDATION_ERROR',
        message: 'PUT requires all fields (name, email)'
      }
    });
  }

  user.name = name;
  user.email = email;

  res.json({ data: user });
});

// 5. PATCH - Partial update
app.patch('/api/v1/users/:id', (req, res) => {
  const user = users.find(u => u.id === parseInt(req.params.id));

  if (!user) {
    return res.status(404).json({
      error: {
        code: 'NOT_FOUND',
        message: 'User not found'
      }
    });
  }

  // Update only provided fields
  if (req.body.name !== undefined) user.name = req.body.name;
  if (req.body.email !== undefined) user.email = req.body.email;

  res.json({ data: user });
});
```

```javascript
// 6. DELETE - Remove user
app.delete('/api/v1/users/:id', (req, res) => {
  const index = users.findIndex(u => u.id === parseInt(req.params.id));

  if (index === -1) {
    return res.status(404).json({
      error: {
        code: 'NOT_FOUND',
        message: 'User not found'
      }
    });
  }

  users.splice(index, 1);

  res.status(204).send(); // No content
});

// Error handling middleware
app.use((err, req, res, next) => {
  console.error(err);
  res.status(500).json({
    error: {
      code: 'INTERNAL_ERROR',
      message: 'Something went wrong'
    }
  });
});

app.listen(3000, () => {
  console.log('API running on port 3000');
});
```

REST API Best Practices

1. Filtering, Sorting, and Searching

```javascript
```

```
// Filtering
GET /users?role=admin&status=active

app.get('/api/v1/users', (req, res) => {
  let filtered = users;

  if (req.query.role) {
    filtered = filtered.filter(u => u.role === req.query.role);
  }

  if (req.query.status) {
    filtered = filtered.filter(u => u.status === req.query.status);
  }

  res.json({ data: filtered });
});

// Sorting
GET /users?sort=name          # Ascending
GET /users?sort=-createdAt    # Descending (- prefix)

app.get('/api/v1/users', (req, res) => {
  let sorted = [...users];

  if (req.query.sort) {
    const sortField = req.query.sort.replace('-', '');
    const sortOrder = req.query.sort.startsWith('-') ? -1 : 1;

    sorted.sort((a, b) => {
      if (a[sortField] < b[sortField]) return -1 * sortOrder;
      if (a[sortField] > b[sortField]) return 1 * sortOrder;
      return 0;
    });
  }

  res.json({ data: sorted });
});

// Field Selection (sparse fieldsets)
GET /users?fields=id,name,email  # Only return specified fields

app.get('/api/v1/users', (req, res) => {
  let result = users;

  if (req.query.fields) {
    const fields = req.query.fields.split(',');
```

```
    result = users.map(user => {
      const filtered = {};
      fields.forEach(field => {
        if (user[field] !== undefined) {
          filtered[field] = user[field];
        }
      });
      return filtered;
    });
  }

  res.json({ data: result });
});

// Search
GET /users?q=john  # Full-text search

app.get('/api/v1/users', (req, res) => {
  let result = users;

  if (req.query.q) {
    const query = req.query.q.toLowerCase();
    result = users.filter(u =>
      u.name.toLowerCase().includes(query) ||
      u.email.toLowerCase().includes(query)
    );
  }

  res.json({ data: result });
});
```

2. HATEOAS (Hypermedia)

Concept: Include links to related resources in responses.

```
javascript
```

```
{
  "data": {
    "id": 123,
    "name": "John Doe",
    "email": "john@example.com"
  },
  "links": {
    "self": "/users/123",
    "posts": "/users/123/posts",
    "friends": "/users/123/friends",
    "edit": "/users/123",
    "delete": "/users/123"
  }
}

// Client can discover available actions!
```

3. Content Negotiation

javascript

```
// Client specifies desired format
GET /users/123
Accept: application/json

GET /users/123
Accept: application/xml

// Server responds appropriately
app.get('/api/v1/users/:id', (req, res) => {
  const user = users.find(u => u.id === parseInt(req.params.id));

  if (!user) {
    return res.status(404).json({ error: 'Not found' });
  }

  const accept = req.get('Accept');

  if (accept.includes('application/xml')) {
    res.type('application/xml');
    res.send(`
      <user>
        <id>${user.id}</id>
        <name>${user.name}</name>
        <email>${user.email}</email>
      </user>
    `);
  } else {
    res.json({ data: user });
  }
});
```

---

2. GraphQL Basics

Problem with REST:

Scenario: Display user profile with posts and comments

REST Approach (Multiple Requests):
1. GET /users/123 # Get user
2. GET /users/123/posts # Get user's posts
3. GET /posts/456/comments # Get post 1 comments
4. GET /posts/457/comments # Get post 2 comments
... (N+1 problem!)

Total: 1 + 1 + N requests!

GraphQL Approach (Single Request):
1. POST /graphql with query:

```
{
  user(id: 123) {
    name
    email
    posts {
      title
      comments {
        text
        author
      }
    }
  }
}
```

Total: 1 request! Get exactly what you need.

GraphQL Schema

graphql

```
# Define types
type User {
  id: ID!
  name: String!
  email: String!
  posts: [Post!]!
  friends: [User!]!
}

type Post {
  id: ID!
  title: String!
  content: String!
  author: User!
  comments: [Comment!]!
  createdAt: DateTime!
}

type Comment {
  id: ID!
  text: String!
  author: User!
  post: Post!
}

# Query operations (read)
type Query {
  user(id: ID!): User
  users(limit: Int, offset: Int): [User!]!
  post(id: ID!): Post
  posts(authorId: ID): [Post!]!
  search(query: String!): [SearchResult!]!
}

# Mutation operations (write)
type Mutation {
  createUser(name: String!, email: String!): User!
  updateUser(id: ID!, name: String, email: String): User!
  deleteUser(id: ID!): Boolean!

  createPost(title: String!, content: String!): Post!
  addComment(postId: ID!, text: String!): Comment!
}

# Subscription operations (real-time)
type Subscription {
```

```
  newPost: Post!
  newComment(postId: ID!): Comment!
}

# Union type for search results
union SearchResult = User | Post | Comment
```

GraphQL Server (Node.js)

javascript

```
const { ApolloServer, gql } = require('apollo-server');

// Type definitions
const typeDefs = gql`
  type User {
    id: ID!
    name: String!
    email: String!
    posts: [Post!]!
  }

  type Post {
    id: ID!
    title: String!
    content: String!
    author: User!
    comments: [Comment!]!
  }

  type Comment {
    id: ID!
    text: String!
    author: User!
  }

  type Query {
    user(id: ID!): User
    users: [User!]!
    post(id: ID!): Post
  }

  type Mutation {
    createUser(name: String!, email: String!): User!
    createPost(title: String!, content: String!, authorId: ID!): Post!
  }
`;

// In-memory data
const users = [
  { id: '1', name: 'John Doe', email: 'john@example.com' },
  { id: '2', name: 'Jane Smith', email: 'jane@example.com' }
];

const posts = [
  { id: '1', title: 'First Post', content: 'Content...', authorId: '1' },
  { id: '2', title: 'Second Post', content: 'Content...', authorId: '1' }
```

```
];

const comments = [
  { id: '1', text: 'Great post!', authorId: '2', postId: '1' }
];

// Resolvers
const resolvers = {
  Query: {
    user: (parent, args) => {
      return users.find(u => u.id === args.id);
    },
    users: () => users,
    post: (parent, args) => {
      return posts.find(p => p.id === args.id);
    }
  },

  Mutation: {
    createUser: (parent, args) => {
      const newUser = {
        id: String(users.length + 1),
        name: args.name,
        email: args.email
      };
      users.push(newUser);
      return newUser;
    },

    createPost: (parent, args) => {
      const newPost = {
        id: String(posts.length + 1),
        title: args.title,
        content: args.content,
        authorId: args.authorId
      };
      posts.push(newPost);
      return newPost;
    }
  },

  // Field resolvers
  User: {
    posts: (user) => {
      return posts.filter(p => p.authorId === user.id);
    }
  },
```

```
  Post: {
    author: (post) => {
      return users.find(u => u.id === post.authorId);
    },
    comments: (post) => {
      return comments.filter(c => c.postId === post.id);
    }
  },

  Comment: {
    author: (comment) => {
      return users.find(u => u.id === comment.authorId);
    }
  }
};

// Create server
const server = new ApolloServer({ typeDefs, resolvers });

server.listen().then(({ url }) => {
  console.log(`GraphQL server ready at ${url}`);
});
```

GraphQL Queries (Client Side)

javascript

```
// Using fetch
async function queryGraphQL(query, variables) {
  const response = await fetch('http://localhost:4000/graphql', {
    method: 'POST',
    headers: {
      'Content-Type': 'application/json',
    },
    body: JSON.stringify({ query, variables })
  });

  return await response.json();
}

// 1. Simple query
const result = await queryGraphQL(`
  query {
    user(id: "1") {
      name
      email
    }
  }
`);

console.log(result.data.user);
// { name: 'John Doe', email: 'john@example.com' }

// 2. Query with nested data
const result2 = await queryGraphQL(`
  query {
    user(id: "1") {
      name
      email
      posts {
        title
        content
        comments {
          text
          author {
            name
          }
        }
      }
    }
  }
`);
```

```
// 3. Query with variables
const result3 = await queryGraphQL(
  `
    query GetUser($userId: ID!) {
      user(id: $userId) {
        name
        email
      }
    }
  `,
  { userId: "1" }
);

// 4. Mutation
const result4 = await queryGraphQL(`
  mutation {
    createUser(name: "Bob Smith", email: "bob@example.com") {
      id
      name
      email
    }
  }
`);

// 5. Multiple queries in one request
const result5 = await queryGraphQL(`
  query {
    user1: user(id: "1") {
      name
    }
    user2: user(id: "2") {
      name
    }
    allPosts: posts {
      title
    }
  }
`);

// 6. Fragments (reusable fields)
const result6 = await queryGraphQL(`
  fragment UserDetails on User {
    id
    name
    email
  }
```

```
  query {
    user1: user(id: "1") {
      ...UserDetails
      posts {
        title
      }
    }
    user2: user(id: "2") {
      ...UserDetails
    }
  }
`);
```

GraphQL vs REST

Feature	REST	GraphQL
Requests	Multiple	Single
Over/Under fetch	Common	Rare
Versioning	Required	Not needed
Learning curve	Easy	Moderate
Caching	Easy (HTTP)	Complex
File uploads	Easy	Complex
Real-time	Need WebSocket	Built-in (subs)
Tooling	Mature	Growing

When to use GraphQL:
✓ Complex, nested data requirements
✓ Multiple client types (web, mobile, desktop)
✓ Frequent schema changes
✓ Need real-time updates

When to use REST:
✓ Simple CRUD operations
✓ Need HTTP caching
✓ Team familiar with REST
✓ File upload/download heavy

---

3. gRPC and Protocol Buffers

gRPC: High-performance RPC (Remote Procedure Call) framework.

Why gRPC?

REST API Call:
1. Serialize data to JSON (slow)
2. Send over HTTP/1.1 (text-based)
3. Parse JSON on server (slow)
4. Process
5. Serialize response to JSON
6. Parse JSON on client

gRPC Call:
1. Serialize data to binary (fast!)
2. Send over HTTP/2 (binary, multiplexed)
3. Deserialize binary on server (fast!)
4. Process
5. Serialize response to binary
6. Deserialize binary on client

Result: 5-10x faster!

Protocol Buffers (.proto file)

protobuf

```
// user.proto
syntax = "proto3";

package userservice;

// Message definitions (like structs)
message User {
  int32 id = 1;
  string name = 2;
  string email = 3;
  repeated Post posts = 4;  // repeated = array
}

message Post {
  int32 id = 1;
  string title = 2;
  string content = 3;
  int32 author_id = 4;
}

message GetUserRequest {
  int32 user_id = 1;
}

message GetUserResponse {
  User user = 1;
}

message CreateUserRequest {
  string name = 1;
  string email = 2;
}

message CreateUserResponse {
  User user = 1;
}

message ListUsersRequest {
  int32 page = 1;
  int32 page_size = 2;
}

message ListUsersResponse {
  repeated User users = 1;
  int32 total = 2;
}
```

```
// Service definition (like interface)
service UserService {
  // Unary RPC (single request, single response)
  rpc GetUser(GetUserRequest) returns (GetUserResponse);
  rpc CreateUser(CreateUserRequest) returns (CreateUserResponse);
  rpc ListUsers(ListUsersRequest) returns (ListUsersResponse);

  // Server streaming (single request, stream of responses)
  rpc StreamUsers(ListUsersRequest) returns (stream User);

  // Client streaming (stream of requests, single response)
  rpc CreateUsers(stream CreateUserRequest) returns (CreateUserResponse);

  // Bidirectional streaming
  rpc Chat(stream ChatMessage) returns (stream ChatMessage);
}

message ChatMessage {
  int32 user_id = 1;
  string text = 2;
  int64 timestamp = 3;
}
```

---

gRPC Server (Python)

python

```python
# Generated from .proto file: user_pb2.py, user_pb2_grpc.py

import grpc
from concurrent import futures
import user_pb2
import user_pb2_grpc

# In-memory data
users = [
    user_pb2.User(id=1, name='John Doe', email='john@example.com'),
    user_pb2.User(id=2, name='Jane Smith', email='jane@example.com')
]

class UserServiceServicer(user_pb2_grpc.UserServiceServicer):
    """Implementation of UserService"""

    def GetUser(self, request, context):
        """Unary RPC"""
        user = next((u for u in users if u.id == request.user_id), None)

        if not user:
            context.set_code(grpc.StatusCode.NOT_FOUND)
            context.set_details('User not found')
            return user_pb2.GetUserResponse()

        return user_pb2.GetUserResponse(user=user)

    def CreateUser(self, request, context):
        """Unary RPC"""
        new_user = user_pb2.User(
            id=len(users) + 1,
            name=request.name,
            email=request.email
        )
        users.append(new_user)

        return user_pb2.CreateUserResponse(user=new_user)

    def ListUsers(self, request, context):
        """Unary RPC with pagination"""
        page = request.page or 1
        page_size = request.page_size or 10

        start = (page - 1) * page_size
        end = start + page_size
```

```
        paginated = users[start:end]

        return user_pb2.ListUsersResponse(
            users=paginated,
            total=len(users)
        )

    def StreamUsers(self, request, context):
        """Server streaming RPC"""
        for user in users:
            yield user
            # Can send users as they're retrieved from DB
            # Client receives them one by one

    def CreateUsers(self, request_iterator, context):
        """Client streaming RPC"""
        created_count = 0

        for create_request in request_iterator:
            new_user = user_pb2.User(
                id=len(users) + 1,
                name=create_request.name,
                email=create_request.email
            )
            users.append(new_user)
            created_count += 1

        return user_pb2.CreateUserResponse(
            user=user_pb2.User(id=created_count)
        )

def serve():
    server = grpc.server(futures.ThreadPoolExecutor(max_workers=10))
    user_pb2_grpc.add_UserServiceServicer_to_server(
        UserServiceServicer(), server
    )
    server.add_insecure_port('[::]:50051')
    server.start()
    print('gRPC server started on port 50051')
    server.wait_for_termination()

if __name__ == '__main__':
    serve()
```

gRPC Client (Python)

python

```python
import grpc
import user_pb2
import user_pb2_grpc

def run():
    # Create channel
    channel = grpc.insecure_channel('localhost:50051')
    stub = user_pb2_grpc.UserServiceStub(channel)

    # 1. Unary RPC - Get user
    request = user_pb2.GetUserRequest(user_id=1)
    response = stub.GetUser(request)
    print(f"User: {response.user.name} ({response.user.email})")

    # 2. Unary RPC - Create user
    request = user_pb2.CreateUserRequest(
        name='Bob Smith',
        email='bob@example.com'
    )
    response = stub.CreateUser(request)
    print(f"Created user: {response.user.id}")

    # 3. Server streaming - Stream users
    request = user_pb2.ListUsersRequest()
    for user in stub.StreamUsers(request):
        print(f"Received user: {user.name}")

    # 4. Client streaming - Create multiple users
    def generate_requests():
        for i in range(5):
            yield user_pb2.CreateUserRequest(
                name=f'User {i}',
                email=f'user{i}@example.com'
            )

    response = stub.CreateUsers(generate_requests())
    print(f"Created {response.user.id} users")

    # 5. Error handling
    try:
        request = user_pb2.GetUserRequest(user_id=999)
        response = stub.GetUser(request)
    except grpc.RpcError as e:
        print(f"Error: {e.code()} - {e.details()}")

    channel.close()
```

```
if __name__ == '__main__':
    run()
```

---

gRPC Features

1. BINARY PROTOCOL
   - Smaller payloads (vs JSON)
   - Faster serialization/deserialization

2. HTTP/2
   - Multiplexing (multiple requests on one connection)
   - Bidirectional streaming
   - Header compression

3. STREAMING
   - Server streaming: Server sends stream of data
   - Client streaming: Client sends stream of data
   - Bidirectional: Both stream simultaneously

4. CODE GENERATION
   - Generate client/server code from .proto
   - Type-safe
   - Multiple languages (Go, Python, Java, etc.)

5. DEADLINES/TIMEOUTS
   Built-in timeout support

6. CANCELLATION
   Cancel requests mid-flight

When to use gRPC:
✓ Microservices communication
✓ Real-time communication
✓ High performance needed
✓ Polyglot environment (multiple languages)
✓ Streaming data

When NOT to use gRPC:
✗ Browser clients (limited support)
✗ Need human-readable API
✗ HTTP caching important
✗ Simple CRUD REST is sufficient

---

4. WebSockets for Real-Time Communication

Problem: HTTP is request-response. Server can't push to client.

```
HTTP Polling (Inefficient):
Client: "Any updates?" → Server: "No"
[wait 1 second]
Client: "Any updates?" → Server: "No"
[wait 1 second]
Client: "Any updates?" → Server: "Yes! Here's data"

Problem: Many wasted requests, high latency

WebSocket (Efficient):
Client: Upgrade to WebSocket
Server: Connection open
[connection stays open]
Server: "New data!" → Client receives immediately
Server: "More data!" → Client receives immediately

Benefits: Real-time, low latency, bidirectional
```

WebSocket Connection Lifecycle

```
1. HANDSHAKE (HTTP Upgrade)
   Client → Server:
   GET /chat HTTP/1.1
   Upgrade: websocket
   Connection: Upgrade
   Sec-WebSocket-Key: x3JJHMbDL1EzLkh9GBhXDw==

   Server → Client:
   HTTP/1.1 101 Switching Protocols
   Upgrade: websocket
   Connection: Upgrade
   Sec-WebSocket-Accept: HSmrc0sMlYUkAGmm5OPpG2HaGWk=

2. CONNECTION OPEN
   - Full-duplex communication
   - Low overhead (small frames)
   - Both can send anytime

3. MESSAGE EXCHANGE
   Client → Server: {"type": "chat", "message": "Hello"}
```

Server → Client: {"type": "chat", "message": "Hi there"}

4. CONNECTION CLOSE
   Either side can close
   Close frame sent
   Connection terminated

WebSocket Server (Node.js)

javascript

```
const WebSocket = require('ws');

const wss = new WebSocket.Server({ port: 8080 });

// Store connected clients
const clients = new Set();

wss.on('connection', (ws, req) => {
  console.log('New client connected');
  clients.add(ws);

  // Send welcome message
  ws.send(JSON.stringify({
    type: 'welcome',
    message: 'Connected to chat server'
  }));

  // Handle incoming messages
  ws.on('message', (data) => {
    console.log('Received:', data.toString());

    try {
      const message = JSON.parse(data);

      // Broadcast to all clients
      broadcast(message);

    } catch (err) {
      console.error('Invalid message:', err);
      ws.send(JSON.stringify({
        type: 'error',
        message: 'Invalid message format'
      }));
    }
  });

  // Handle connection close
  ws.on('close', () => {
    console.log('Client disconnected');
    clients.delete(ws);
  });

  // Handle errors
  ws.on('error', (err) => {
    console.error('WebSocket error:', err);
    clients.delete(ws);
```

```
  });

  // Heartbeat to detect dead connections
  ws.isAlive = true;
  ws.on('pong', () => {
    ws.isAlive = true;
  });
});

// Broadcast message to all connected clients
function broadcast(message) {
  const data = JSON.stringify(message);

  clients.forEach((client) => {
    if (client.readyState === WebSocket.OPEN) {
      client.send(data);
    }
  });
}

// Ping clients every 30 seconds
setInterval(() => {
  clients.forEach((ws) => {
    if (!ws.isAlive) {
      console.log('Terminating dead connection');
      ws.terminate();
      clients.delete(ws);
      return;
    }

    ws.isAlive = false;
    ws.ping();
  });
}, 30000);

console.log('WebSocket server running on port 8080');
```

WebSocket Client (Browser)

javascript

```
// Connect to WebSocket server
const ws = new WebSocket('ws://localhost:8080');

// Connection opened
ws.addEventListener('open', (event) => {
  console.log('Connected to server');

  // Send message
  ws.send(JSON.stringify({
    type: 'chat',
    user: 'John',
    message: 'Hello everyone!'
  }));
});

// Listen for messages
ws.addEventListener('message', (event) => {
  console.log('Message from server:', event.data);

  const data = JSON.parse(event.data);

  switch (data.type) {
    case 'welcome':
      console.log('Welcome:', data.message);
      break;

    case 'chat':
      displayMessage(data.user, data.message);
      break;

    case 'notification':
      showNotification(data.message);
      break;
  }
});

// Connection closed
ws.addEventListener('close', (event) => {
  console.log('Disconnected from server');

  // Attempt reconnection
  setTimeout(reconnect, 3000);
});

// Connection error
ws.addEventListener('error', (error) => {
```

```
  console.error('WebSocket error:', error);
});

// Send chat message
function sendMessage(message) {
  if (ws.readyState === WebSocket.OPEN) {
    ws.send(JSON.stringify({
      type: 'chat',
      user: currentUser,
      message: message
    }));
  } else {
    console.error('WebSocket not connected');
  }
}

// Reconnection logic
function reconnect() {
  console.log('Attempting to reconnect...');
  ws = new WebSocket('ws://localhost:8080');
  // Re-attach event listeners
}

// Display message in UI
function displayMessage(user, message) {
  const messageElement = document.createElement('div');
  messageElement.textContent = `${user}: ${message}`;
  document.getElementById('messages').appendChild(messageElement);
}
```

WebSocket Use Cases

1. CHAT APPLICATIONS
   - Real-time messaging
   - Typing indicators
   - Presence (online/offline)

2. LIVE UPDATES
   - Stock prices
   - Sports scores
   - News feeds
   - Social media updates

3. COLLABORATIVE EDITING
   - Google Docs-style editing

- Code editors (CodePen, VS Code Live Share)
- Whiteboard applications

4. GAMING
- Multiplayer games
- Real-time game state synchronization

5. NOTIFICATIONS
- Push notifications
- Alert systems
- Status updates

6. IOT / SENSOR DATA
- Real-time sensor readings
- Device control
- Dashboard updates

7. LIVE STREAMING METADATA
- Viewer count
- Comments/reactions
- Stream statistics

---

5. API Versioning Strategies

Why version? Breaking changes would break existing clients!

Strategy 1: URL Versioning (Most Common)

GET /api/v1/users
GET /api/v2/users
GET /api/v3/users

Pros:
✓ Clear and explicit
✓ Easy to route
✓ Can run multiple versions simultaneously
✓ Easy to deprecate old versions

Cons:
✗ URLs change between versions
✗ Duplication of code

Implementation:

javascript

```
const express = require('express');
const app = express();

// Version 1
app.get('/api/v1/users/:id', (req, res) => {
  const user = getUser(req.params.id);

  // V1 format
  res.json({
    id: user.id,
    name: user.name,
    email: user.email
  });
});

// Version 2 (added additional fields)
app.get('/api/v2/users/:id', (req, res) => {
  const user = getUser(req.params.id);

  // V2 format (more fields)
  res.json({
    id: user.id,
    name: user.name,
    email: user.email,
    profile: {
      avatar: user.avatar,
      bio: user.bio
    },
    createdAt: user.createdAt
  });
});

// Version 3 (breaking change: renamed field)
app.get('/api/v3/users/:id', (req, res) => {
  const user = getUser(req.params.id);

  // V3 format (username instead of name)
  res.json({
    id: user.id,
    username: user.name,  // Renamed!
    email: user.email,
    profile: {
      avatar: user.avatar,
      bio: user.bio
    },
    createdAt: user.createdAt
```

```
  });
});
```

Strategy 2: Header Versioning

```
GET /api/users
Accept-Version: v1

GET /api/users
Accept-Version: v2
```

Pros:
✓ Clean URLs
✓ Standard HTTP headers

Cons:
✗ Less visible
✗ Harder to test (need to set headers)
✗ Caching more complex

Implementation:

```javascript
```

```
app.get('/api/users/:id', (req, res) => {
  const version = req.get('Accept-Version') || 'v1';
  const user = getUser(req.params.id);

  switch (version) {
    case 'v1':
      res.json({
        id: user.id,
        name: user.name
      });
      break;

    case 'v2':
      res.json({
        id: user.id,
        name: user.name,
        profile: {
          avatar: user.avatar
        }
      });
      break;

    default:
      res.status(400).json({
        error: 'Unsupported API version'
      });
  }
});
```

Strategy 3: Query Parameter Versioning

GET /api/users?version=1
GET /api/users?version=2

Pros:
✓ Easy to test
✓ Can be optional (default version)

Cons:
✗ Pollutes query parameters
✗ Can be confused with filtering

Implementation:

javascript

```javascript
app.get('/api/users/:id', (req, res) => {
  const version = req.query.version || '1';
  const user = getUser(req.params.id);

  if (version === '1') {
    res.json({ id: user.id, name: user.name });
  } else if (version === '2') {
    res.json({ id: user.id, name: user.name, profile: user.profile });
  } else {
    res.status(400).json({ error: 'Invalid version' });
  }
});
```

Strategy 4: Content Negotiation

GET /api/users
Accept: application/vnd.myapi.v1+json

GET /api/users
Accept: application/vnd.myapi.v2+json

Pros:
✓ RESTful (uses Accept header correctly)
✓ Clean URLs

Cons:
✗ Complex
✗ Custom MIME types
✗ Not beginner-friendly

Implementation:

javascript

```
app.get('/api/users/:id', (req, res) => {
  const accept = req.get('Accept');
  const user = getUser(req.params.id);

  if (accept.includes('vnd.myapi.v1+json')) {
    res.type('application/vnd.myapi.v1+json');
    res.json({ id: user.id, name: user.name });
  } else if (accept.includes('vnd.myapi.v2+json')) {
    res.type('application/vnd.myapi.v2+json');
    res.json({ id: user.id, name: user.name, profile: user.profile });
  } else {
    res.status(406).json({ error: 'Unsupported media type' });
  }
});
```

Versioning Best Practices

1. MAINTAIN BACKWARD COMPATIBILITY
   - Don't break existing clients
   - Deprecate gradually
   - Give advance notice

2. VERSION ONLY WHEN NECESSARY
   ✓ Breaking changes (renamed/removed fields)
   ✓ Changed behavior
   ✗ New optional fields (backward compatible!)
   ✗ Bug fixes

3. DEPRECATION POLICY
   - Announce deprecation
   - Give timeline (e.g., 6 months)
   - Provide migration guide
   - Monitor usage
   - Finally sunset old version

Example:

javascript

```
// Add deprecation warning
app.get('/api/v1/users/:id', (req, res) => {
  // Add warning header
  res.set('Warning', '299 - "API v1 is deprecated. Please migrate to v2 by 2024-12-31"');
  res.set('X-API-Deprecation-Info', 'https://docs.example.com/migration-guide');

  // Return data
  const user = getUser(req.params.id);
  res.json(user);
});

// Log usage for monitoring
app.use('/api/v1/*', (req, res, next) => {
  logDeprecatedAPIUsage(req.path, req.ip);
  next();
});
```

---

6. Rate Limiting and Throttling

Why rate limit?

- Prevent abuse
- Ensure fair usage
- Protect server resources
- Prevent DDoS attacks

Rate Limiting Algorithms

1. Fixed Window

Algorithm:
- Window: 1 minute
- Limit: 100 requests
- Reset at start of each minute

Timeline:
00:00 - Request 1-100 allowed
00:01 - Counter resets, requests 1-100 allowed again

Problem:
00:00:59 - 100 requests
00:01:00 - 100 requests
Result: 200 requests in 2 seconds! (burst)

Implementation:

javascript

```javascript
const rateLimit = {};

function fixedWindowRateLimit(userId, limit = 100, windowMs = 60000) {
  const now = Date.now();
  const windowStart = Math.floor(now / windowMs) * windowMs;
  const key = `${userId}:${windowStart}`;

  if (!rateLimit[key]) {
    rateLimit[key] = 0;
  }

  rateLimit[key]++;

  if (rateLimit[key] > limit) {
    return {
      allowed: false,
      remaining: 0,
      resetAt: windowStart + windowMs
    };
  }

  return {
    allowed: true,
    remaining: limit - rateLimit[key],
    resetAt: windowStart + windowMs
  };
}

// Cleanup old windows
setInterval(() => {
  const now = Date.now();
  for (const key in rateLimit) {
    const timestamp = parseInt(key.split(':')[1]);
    if (now - timestamp > 120000) {  // 2 minutes old
      delete rateLimit[key];
    }
  }
}, 60000);
```

---

2. Sliding Window Log

Algorithm:
- Track timestamp of each request
- Remove requests older than window
- Count remaining requests

Timeline:
00:00:00 - Request (logged)
00:00:30 - Request (logged)
00:00:59 - Request (logged)
00:01:00 - Check: Remove requests before 00:00:00
    Only 2 requests in last minute!

Benefits: No burst problem!

Implementation:

javascript

```
const requestLogs = new Map();

function slidingWindowLog(userId, limit = 100, windowMs = 60000) {
  const now = Date.now();

  if (!requestLogs.has(userId)) {
    requestLogs.set(userId, []);
  }

  const log = requestLogs.get(userId);

  // Remove old requests
  const cutoff = now - windowMs;
  while (log.length > 0 && log[0] < cutoff) {
    log.shift();
  }

  if (log.length >= limit) {
    return {
      allowed: false,
      remaining: 0,
      resetAt: log[0] + windowMs
    };
  }

  // Add current request
  log.push(now);

  return {
    allowed: true,
    remaining: limit - log.length,
    resetAt: log[0] + windowMs
  };
}
```

---

3. Token Bucket

Algorithm:
- Bucket holds tokens
- Each request consumes 1 token
- Tokens refill at constant rate
- If bucket empty, request denied

Example:
- Bucket capacity: 100 tokens
- Refill rate: 10 tokens/second
- Request arrives: Takes 1 token
- Allows bursts (up to bucket capacity)
- But sustained rate limited to refill rate

Implementation:

```javascript
```

```
const buckets = new Map();

function tokenBucket(userId, capacity = 100, refillRate = 10) {
  const now = Date.now();

  if (!buckets.has(userId)) {
    buckets.set(userId, {
      tokens: capacity,
      lastRefill: now
    });
  }

  const bucket = buckets.get(userId);

  // Refill tokens based on time elapsed
  const timePassed = (now - bucket.lastRefill) / 1000; // seconds
  const tokensToAdd = timePassed * refillRate;
  bucket.tokens = Math.min(capacity, bucket.tokens + tokensToAdd);
  bucket.lastRefill = now;

  if (bucket.tokens < 1) {
    return {
      allowed: false,
      remaining: 0,
      retryAfter: (1 - bucket.tokens) / refillRate
    };
  }

  // Consume token
  bucket.tokens -= 1;

  return {
    allowed: true,
    remaining: Math.floor(bucket.tokens),
    retryAfter: 0
  };
}
```

---

4. Leaky Bucket

Algorithm:
- Queue requests in bucket
- Process at constant rate
- If bucket full, reject new requests

Like water dripping from bucket:
- Water poured in (requests arrive)
- Small hole at bottom (constant processing)
- If bucket overflows, water spills (request rejected)

Benefits: Smooth output rate

Implementation:

```javascript
```

```javascript
const queues = new Map();

function leakyBucket(userId, capacity = 100, leakRate = 10) {
  const now = Date.now();

  if (!queues.has(userId)) {
    queues.set(userId, {
      requests: [],
      lastLeak: now
    });
  }

  const queue = queues.get(userId);

  // Leak requests (process at constant rate)
  const timePassed = (now - queue.lastLeak) / 1000;
  const requestsToLeak = Math.floor(timePassed * leakRate);

  for (let i = 0; i < requestsToLeak && queue.requests.length > 0; i++) {
    queue.requests.shift();
  }

  queue.lastLeak = now;

  if (queue.requests.length >= capacity) {
    return {
      allowed: false,
      queued: queue.requests.length,
      retryAfter: queue.requests.length / leakRate
    };
  }

  // Add to queue
  queue.requests.push(now);

  return {
    allowed: true,
    queued: queue.requests.length,
    retryAfter: 0
  };
}
```

Rate Limiting in Practice (Express Middleware)

```javascript
```

```
const express = require('express');
const redis = require('redis');
const app = express();

// Using Redis for distributed rate limiting
const redisClient = redis.createClient();

// Rate limiting middleware
function rateLimiter(options = {}) {
  const {
    windowMs = 60000,      // 1 minute
    max = 100,             // 100 requests
    message = 'Too many requests',
    statusCode = 429
  } = options;

  return async (req, res, next) => {
    // Identify client (by IP or API key)
    const key = req.ip || req.get('X-Forwarded-For');
    const redisKey = `rate_limit:${key}`;

    try {
      // Get current count
      const current = await redisClient.get(redisKey);
      const count = current ? parseInt(current) : 0;

      if (count >= max) {
        // Rate limit exceeded
        const ttl = await redisClient.ttl(redisKey);

        res.set({
          'X-RateLimit-Limit': max,
          'X-RateLimit-Remaining': 0,
          'X-RateLimit-Reset': Date.now() + (ttl * 1000),
          'Retry-After': ttl
        });

        return res.status(statusCode).json({
          error: {
            code: 'RATE_LIMIT_EXCEEDED',
            message: message
          }
        });
      }

      // Increment counter
```

```
      if (count === 0) {
        // First request in window
        await redisClient.setex(redisKey, Math.ceil(windowMs / 1000), 1);
      } else {
        await redisClient.incr(redisKey);
      }

      // Set rate limit headers
      res.set({
        'X-RateLimit-Limit': max,
        'X-RateLimit-Remaining': max - count - 1,
        'X-RateLimit-Reset': Date.now() + windowMs
      });

      next();

    } catch (err) {
      console.error('Rate limiter error:', err);
      // Fail open (allow request if rate limiter fails)
      next();
    }
  };
}

// Apply to all routes
app.use(rateLimiter({
  windowMs: 60000,  // 1 minute
  max: 100          // 100 requests per minute
}));

// Different limits for different endpoints
app.use('/api/login', rateLimiter({
  windowMs: 900000,  // 15 minutes
  max: 5,            // Only 5 login attempts
  message: 'Too many login attempts'
}));

app.use('/api/expensive-operation', rateLimiter({
  windowMs: 60000,
  max: 10  // Only 10 per minute for expensive operation
}));

// Routes
app.get('/api/users', (req, res) => {
  res.json({ users: [] });
});
```

```
app.post('/api/login', (req, res) => {
  // Login logic
  res.json({ token: 'abc123' });
});
```

Rate Limiting by User Tier

javascript

```
function getTierLimits(userTier) {
  const tiers = {
    free: { max: 100, windowMs: 3600000 },    // 100/hour
    basic: { max: 1000, windowMs: 3600000 },  // 1000/hour
    premium: { max: 10000, windowMs: 3600000 },// 10000/hour
    enterprise: { max: 100000, windowMs: 3600000 } // 100k/hour
  };

  return tiers[userTier] || tiers.free;
}

function tierBasedRateLimiter(req, res, next) {
  const user = req.user; // From authentication middleware
  const tier = user?.tier || 'free';
  const limits = getTierLimits(tier);

  // Apply tier-specific rate limit
  rateLimiter(limits)(req, res, next);
}

app.use('/api', authenticate, tierBasedRateLimiter);
```

Key Takeaways

1. **REST APIs:**
   - Resource-based URLs
   - HTTP methods as verbs
   - Proper status codes
   - URL versioning most common
2. **GraphQL:**
   - Single endpoint

- Query exactly what you need
- No over/under fetching
- Great for complex data requirements

3. **gRPC:**
   - Binary protocol (fast!)
   - Code generation
   - Streaming support
   - Perfect for microservices
4. **WebSockets:**
   - Real-time bidirectional
   - Persistent connection
   - Great for chat, live updates
   - Lower overhead than polling
5. **API Versioning:**
   - URL versioning most popular
   - Plan for breaking changes
   - Deprecate gracefully
   - Document migrations
6. **Rate Limiting:**
   - Token bucket for flexibility
   - Sliding window for accuracy
   - Use Redis for distribution
   - Different tiers for users

Practice Problems

1. Design a REST API for a blog platform (posts, comments, users)
2. When would you choose GraphQL over REST? When would you choose REST?
3. Implement a rate limiter that allows 100 requests per minute but also 1000 requests per day
4. Design a real-time collaborative editing API (like Google Docs)

Ready to continue with more chapters or dive deeper into any topic?