

Day 1 Complete Implementation Guide

API Foundation & Design - "Build the Blueprint"

Tools & Setup Required

Essential Tools (Pre-flight Check):

- **Postman Desktop** (not web version - for API testing)
- **VS Code** with REST Client extension
- **Web browser** with bookmark bar setup
- **Excalidraw** (excalidraw.com - for domain modeling)
- **GitHub** account

Pre-Session Setup Checklist:

```
bash
```

```
# Student environment verification
```

```
✓ Postman installed and working
```

```
✓ VS Code with REST Client extension
```

```
✓ Can access GitHub
```

```
✓ Has drawing/modeling tool access
```

```
✓ Stable internet connection
```

Your Preparation Materials

1. Demo APIs Collection (Postman)

Create a shared Postman workspace with:

```
json
```

```
{
  "collections": [
    {
      "name": "Good APIs",
      "apis": [
        "GitHub API v4 (GraphQL)",
        "Stripe API",
        "Twilio API"
      ]
    },
    {
      "name": "Educational APIs",
      "apis": [
        "JSONPlaceholder",
        "OpenWeather",
        "REST Countries"
      ]
    },
    {
      "name": "Legacy/Problematic APIs",
      "apis": [
        "SOAP Weather Service",
        "Some poorly designed REST API",
        "Inconsistent response format API"
      ]
    }
  ]
}
```

2. TaskFlow Demo Repository

Create GitHub repo: `taskflow-api-demo`

File Structure:

```
taskflow-api-demo/
├── docs/
│   ├── domain-model.md
│   ├── api-specification.yml (OpenAPI)
│   └── architecture-decisions.md
├── domain-analysis/
│   ├── business-domains.md
│   ├── bounded-contexts.md
│   └── ubiquitous-language.md
├── examples/
│   ├── good-apis/
│   ├── bad-apis/
│   └── grpc-examples/
└── workshops/
    ├── domain-modeling-template.md
    ├── api-design-checklist.md
    └── error-response-examples.json
```

3. Domain Modeling Materials

Physical Materials:

- Large sticky notes (3 colors: yellow, blue, pink)
- Markers (black, red, blue)
- Large paper/whiteboards
- Timer (for activities)

Digital Templates:

- Excalidraw board with domain modeling template (shareable link)
- Canva slides template (if needed)
- Simple Google Doc templates for worksheets

Session-by-Session Breakdown

Session 1: 10:15-10:45 AM - API Experience & Types

Your Prep Materials:

1. API Treasure Hunt Worksheet:

markdown

API Treasure Hunt - 10 Minutes

Your Mission: Explore these 5 APIs and document findings

API 1: JSONPlaceholder (Good Example)

- Base URL: <https://jsonplaceholder.typicode.com>
- Try: GET /posts/1
- ****Find:**** What makes this response developer-friendly?

API 2: GitHub API (Excellent Example)

- Base URL: <https://api.github.com>
- Try: GET /users/octocat
- ****Find:**** How does GitHub handle API versioning?

API 3: OpenWeather (Commercial Example)

- Base URL: <https://api.openweathermap.org/data/2.5>
- Try: GET /weather?q=London&appid=demo
- ****Find:**** How do they handle authentication?

API 4: REST Countries (Simple Example)

- Base URL: <https://restcountries.com/v3.1>
- Try: GET /name/france
- ****Find:**** What's good/bad about the response structure?

API 5: Legacy SOAP Example (Problem Example)

- URL: <http://webservices.oorsprong.org/websamples.countryinfo/CountryInfoService.wso>
- ****Find:**** What makes this difficult to work with?

Questions to Answer:

1. Which API would you want to integrate with? Why?
2. Which API would you avoid? Why?
3. What patterns do you notice in the good APIs?

2. API Classification Challenge:

markdown

API Type Classification - 10 Real APIs

Instructions: Match each API to its type and use case

1. ****Stripe Payment API**** → ?
2. ****GraphQL GitHub API**** → ?
3. ****gRPC Google Cloud APIs**** → ?
4. ****Twilio SMS REST API**** → ?
5. ****Facebook Graph API**** → ?
6. ****Discord WebSocket API**** → ?
7. ****AWS S3 REST API**** → ?
8. ****Slack Real-time Messaging**** → ?
9. ****Google Maps JavaScript API**** → ?
10. ****Netflix Internal gRPC Services**** → ?

Types: REST, GraphQL, gRPC, WebSocket, SOAP

Use Cases: Public, Internal, Real-time, Batch, Mobile, Web

Session 2: 11:00-12:00 PM - Domain-Driven TaskFlow Architecture

Your Demo Script:

markdown

TaskFlow API Demo Script (10 minutes)

1. Business Overview (2 min)

"TaskFlow is a project management system. Let me show you the complete API..."

****Show:**** OpenAPI documentation with all endpoints

****Highlight:**** How business language appears in API design

2. Domain Structure (3 min)

****Show file structure:****

taskflow/ |—— teams/ # Team Management Domain |—— projects/ # Project Planning Domain
|—— tasks/ # Task Execution Domain |—— users/ # User Identity Domain |—— notifications/ #
Communication Domain

3. API Design Philosophy (3 min)

****Show examples:****

- `/teams/123/projects` not `/team-projects`
- `/projects/456/tasks/active` not `/tasks?project=456&status=active`
- Error messages using business terms

4. Real Endpoints (2 min)

****Demo live calls:****

- GET /teams/engineering/projects
- POST /projects/{id}/tasks
- PUT /tasks/{id}/assign

Domain Discussion Prompts:

markdown

DDD Discussion Guide (15 minutes)

Questions to Drive Discussion:

Business Domains (5 min)

1. "Looking at TaskFlow, what business capabilities do you see?"
 - Expected: Teams, Projects, Tasks, Users, Notifications
2. "Which of these could exist independently?"
3. "Where do you see the most complexity?"

Bounded Contexts (5 min)

1. "If we had separate development teams, how would you divide TaskFlow?"
2. "What happens when a Team is deleted? How does that affect Projects?"
3. "Should User management be separate from Team management?"

Ubiquitous Language (5 min)

1. "Is a 'Team' the same as an 'Organization'?"
2. "What's the difference between a 'Task' and an 'Issue'?"
3. "How do business stakeholders talk about these concepts?"

Your Role: Guide toward DDD principles without lecturing

Domain Modeling Activity (20 min):

markdown

Domain Modeling Exercise - Hands-On Workshop

Materials Needed:

- Sticky notes (Yellow=Entities, Blue=Services, Pink=Events)
- Large paper/whiteboard
- Markers

Instructions for Students:

Step 1: Entity Identification (5 min)

- Yellow sticky notes
- Write all the "things" in TaskFlow
- One entity per note

Step 2: Service Identification (5 min)

- Blue sticky notes
- Write all the "actions" or "processes"
- Examples: "Assign Task", "Create Project"

Step 3: Event Identification (5 min)

- Pink sticky notes
- Write all the "happenings"
- Examples: "Task Completed", "Team Member Added"

Step 4: Domain Grouping (5 min)

- Group related notes together
- Draw boundaries around groups
- Name each domain

Your Role:

- Walk around, ask probing questions
- "Why did you put User and Team together?"
- "What happens when this event occurs?"
- "How do these domains communicate?"

Session 3: 1:00-1:45 PM - API Design + Documentation

Domain-to-REST Mapping Exercise:

markdown

Domain-to-REST Mapping Challenge (15 minutes)

Given: TaskFlow Domain Model

```
Team {  
  id, name, members[]  
  Projects[]  
}
```

```
Project {  
  id, name, description, status  
  Tasks[], Team  
}
```

```
Task {  
  id, title, description, status, assignee  
  Project, Comments[]  
}
```

Your Challenge: Design REST endpoints

Rules:

1. Use domain language in URLs
2. Follow REST conventions
3. Think about relationships
4. Consider real-world usage

Questions to Consider:

- How do you get all projects for a team?
- How do you assign a task to someone?
- How do you handle nested resources?
- What about bulk operations?

Expected Solutions:

Teams Domain:

GET /teams

POST /teams

GET /teams/{id}

PUT /teams/{id}

DELETE /teams/{id}

GET /teams/{id}/members

POST /teams/{id}/members

DELETE /teams/{id}/members/{userId}

Projects Domain:

GET /teams/{teamId}/projects

POST /teams/{teamId}/projects
GET /projects/{id}
PUT /projects/{id}
DELETE /projects/{id}

Tasks Domain:

GET /projects/{projectId}/tasks
POST /projects/{projectId}/tasks
GET /tasks/{id}
PUT /tasks/{id}
POST /tasks/{id}/assign
POST /tasks/{id}/comments



OpenAPI Contract Activity:

markdown

OpenAPI Contract Design (10 minutes)

Challenge: Write OpenAPI spec for Task creation

Business Requirements:

- Create a task within a project
- Task must have: title, description, priority
- Task can be assigned to team member
- Response should include task ID and creation timestamp

Your OpenAPI Template:

```
``yaml
paths:
  /projects/{projectId}/tasks:
    post:
      summary: Create a new task
      parameters:
        - name: projectId
          in: path
          required: true
          schema:
            type: string
      requestBody:
        required: true
        content:
          application/json:
            schema:
              type: object
              required:
                - title
              properties:
                title:
                  type: string
                  description: "Task title using business language"
            # Students complete this...
      responses:
        '201':
          description: Task created successfully
          content:
            application/json:
              schema:
                # Students define response schema...
```

Discussion Points:

- How do you validate projectId exists?

- What if assignee is not a team member?
- How do you handle duplicate task titles?

Session 4: 1:45-2:30 PM - Error Handling & Production Patterns

Status Code Scenarios Activity:

```markdown

#### # HTTP Status Code Challenge (15 minutes)

## Instructions: Choose the correct status code + explain why

#### #### Scenario 1: Create Task

**\*\*Request:\*\*** POST /projects/123/tasks

**\*\*Situation:\*\*** Project 123 doesn't exist

**\*\*Your Answer:\*\*** \_\_\_\_

**\*\*Options:\*\*** 400, 404, 422, 500

#### #### Scenario 2: Update Task

**\*\*Request:\*\*** PUT /tasks/456

**\*\*Situation:\*\*** Task exists but user doesn't have permission

**\*\*Your Answer:\*\*** \_\_\_\_

**\*\*Options:\*\*** 401, 403, 404, 409

#### #### Scenario 3: Assign Task

**\*\*Request:\*\*** POST /tasks/789/assign {"userId": "abc"}

**\*\*Situation:\*\*** User exists but is not team member

**\*\*Your Answer:\*\*** \_\_\_\_

**\*\*Options:\*\*** 400, 403, 422, 409

#### #### Scenario 4: Delete Project

**\*\*Request:\*\*** DELETE /projects/321

**\*\*Situation:\*\*** Project has active tasks

**\*\*Your Answer:\*\*** \_\_\_\_

**\*\*Options:\*\*** 400, 409, 422, 500

#### #### Scenario 5: Get Team Projects

**\*\*Request:\*\*** GET /teams/999/projects

**\*\*Situation:\*\*** Database is temporarily down

**\*\*Your Answer:\*\*** \_\_\_\_

**\*\*Options:\*\*** 500, 502, 503, 504

#### ## Additional Scenarios:

6. Rate limit exceeded → \_\_\_\_

7. Invalid JSON in request → \_\_\_\_

8. Duplicate team name → \_\_\_\_

9. File upload too large → \_\_\_\_

10. Authentication token expired → \_\_\_\_

### ## Discussion Questions:

- When do you use 422 vs 400?
- What's the difference between 401 and 403?
- How do you handle partial failures?

## Session 5: 2:45-3:30 PM - gRPC Deep Dive

### Your gRPC Demo Materials:

protobuf

```

// taskflow_analytics.proto
syntax = "proto3";

package taskflow.analytics;

service TaskAnalytics {
 // Unary RPC - get team performance
 rpc GetTeamPerformance(TeamRequest) returns (TeamPerformance);

 // Server streaming - real-time task updates
 rpc StreamTaskUpdates(TaskStreamRequest) returns (stream TaskUpdate);

 // Client streaming - batch task updates
 rpc BatchUpdateTasks(stream TaskUpdateRequest) returns (BatchResponse);

 // Bidirectional streaming - real-time collaboration
 rpc CollaborateOnTask(stream TaskCollaboration) returns (stream TaskCollaboration);
}

message TeamRequest {
 string team_id = 1;
 int32 days_back = 2;
}

message TeamPerformance {
 string team_id = 1;
 int32 completed_tasks = 2;
 int32 overdue_tasks = 3;
 float velocity = 4;
 repeated TaskMetric task_metrics = 5;
}

message TaskUpdate {
 string task_id = 1;
 string status = 2;
 string assigned_to = 3;
 int64 timestamp = 4;
}

```

## gRPC vs REST Comparison Demo:

markdown

```
Live Performance Comparison
```

```
Scenario: Get team performance data
```

```
REST Approach:
```

```
```bash
```

```
# Multiple round trips required
```

```
GET /teams/eng/tasks?status=completed&days=30
```

```
GET /teams/eng/tasks?status=overdue
```

```
GET /teams/eng/velocity?days=30
```

```
GET /teams/eng/members
```

gRPC Approach:

```
bash
```

```
# Single call, structured response
```

```
grpc_cli call localhost:50051 GetTeamPerformance "team_id: 'eng', days_back: 30"
```

Key Differences:

1. **Network Efficiency:** gRPC uses HTTP/2, binary protocol
2. **Type Safety:** Protocol buffers vs JSON
3. **Streaming:** Built-in real-time capabilities
4. **Tooling:** Code generation vs manual client code

When to Choose Each:

- **REST:** Public APIs, web browsers, simple CRUD
- **gRPC:** Internal services, real-time features, high performance

Session 6: 3:30-4:00 PM - Integration Reality Check

Legacy API Challenge Materials:

``markdown

Legacy Integration Challenge (15 minutes)

The Problem: You must integrate with this legacy system

Legacy Weather Service Response:

``xml

<?xml version="1.0"?>

<WeatherResponse>

<Status>OK</Status>

<Data>

<Location>New York</Location>

<Temperature>72</Temperature>

<TemperatureUnit>F</TemperatureUnit>

<Humidity>65</Humidity>

<Conditions>Partly Cloudy</Conditions>

<WindSpeed>10</WindSpeed>

<WindDirection>NW</WindDirection>

<LastUpdated>2024-01-15T14:30:00Z</LastUpdated>

</Data>

</WeatherResponse>

Legacy User Management API:

json


```
{
  "result": "success",
  "data": {
    "user_info": {
      "user_id": "12345",
      "user_name": "john_doe",
      "user_email": "john@example.com",
      "user_status": "1",
      "user_created": "1642248600",
      "user_details": {
        "first_name": "John",
        "last_name": "Doe",
        "phone_number": "555-1234"
      }
    }
  }
}
```

Your Challenge:

Design clean, modern API responses that hide this ugliness from your API consumers.

Requirements:

1. Consistent response format
2. Modern field naming
3. Proper HTTP status codes
4. Error handling for when legacy system fails

Expected Solution Pattern:

json

```
{
  "weather": {
    "location": "New York",
    "temperature": {
      "value": 72,
      "unit": "fahrenheit"
    },
    "humidity": 65,
    "conditions": "partly_cloudy",
    "wind": {
      "speed": 10,
      "direction": "northwest"
    },
    "lastUpdated": "2024-01-15T14:30:00Z"
  }
}
```

📊 Slide Templates (Key Slides Only)

Slide 1: Day 1 Agenda

```markdown

# Day 1: API Foundation & Design

## "Build the Blueprint"

#### Today's Journey:

- 10:15 AM - API Reality Check & Types
- 11:00 AM - Domain-Driven API Architecture
- 1:00 PM - Professional API Design
- 1:45 PM - Production Error Handling
- 2:45 PM - gRPC vs REST Deep Dive
- 3:30 PM - Legacy Integration Reality

#### Outcome: Complete API blueprint ready for production

## Slide 2: API Types Decision Matrix

markdown

## # When to Choose Each API Type

| Scenario           | REST           | GraphQL          | gRPC               | WebSocket    |
|--------------------|----------------|------------------|--------------------|--------------|
| Public API         | ✅ First Choice | ⚡ Mobile Apps    | ❌ Too Complex      | ❌ Overkill   |
| Internal Services  | ✅ Simple CRUD  | ⚠️ Consider      | ✅ High Performance | ❌ Usually No |
| Real-time Features | ❌ Polling Only | ⚠️ Subscriptions | ✅ Streaming        | ✅ Perfect    |
| Mobile Apps        | ✅ Standard     | ✅ Efficient      | ⚠️ Size Matters    | ✅ Real-time  |
| Microservices      | ✅ HTTP/JSON    | ❌ Complexity     | ✅ Performance      | ❌ Wrong Tool |

**\*\*Key Insight:\*\*** Choose based on constraints, not preferences

## Slide 3: Domain-Driven API Design

markdown

### # Domain Language in API Design

### ❌ Technical-Focused URLs:

GET /api/v1/user-project-assignments

POST /api/v1/task-status-updates

GET /api/v1/team-member-relationships

### ✅ Business-Focused URLs:

GET /teams/{id}/projects

POST /tasks/{id}/complete

GET /teams/{id}/members

**\*\*Principle:\*\*** Your API should speak the language of your business

## Slide 4: Production Error Handling

markdown

## # Error Response Design Pattern

### ## ❌ Technical Error:

```
``json
{
 "error": "ValidationError",
 "message": "Foreign key constraint failed",
 "code": 500
}
```

### ✅ Business Error:

```
json
{
 "error": {
 "type": "invalid_request",
 "message": "Cannot assign task to user outside the team",
 "details": {
 "task_id": "task_123",
 "user_id": "user_456",
 "team_id": "team_789"
 },
 "suggestion": "Add user to team first, then assign task"
 }
}
```

---

## ## 🎯 Facilitator Notes & Tips

### #### Timing Management:

- **Use a visible timer** for all activities
- **5-minute warning** for longer activities
- **Have backup discussions** ready if activities finish early

### #### Student engagement strategies with 4 students:

- **Pair up for activities** (2 pairs of 2) - more focused discussion
- **Round-robin sharing** - everyone presents in each activity
- **Collaborative board work** - all 4 can work on same Excalidraw
- **Friendly competition** - pairs compete on design challenges

### #### Common Pitfalls to Address:

- **Over-engineering:** "Keep it simple, add complexity when needed"
- **Technical jargon:** "How would you explain this to a product manager?"
- **Perfect solutions:** "What would break first at scale?"

### #### Energy Management:

- **High energy start:** Jump right into activities
- **Post-lunch dip:** Make 1:00 PM highly interactive
- **End strong:** Legacy challenge is fun and relatable

---

## ## ✅ Success Checklist for Day 1

By 4:00 PM, students should be able to:

- [ ] Classify APIs by type and use case
- [ ] Map business domains to API structure
- [ ] Design REST endpoints using domain language
- [ ] Choose appropriate HTTP status codes
- [ ] Explain when to use gRPC vs REST
- [ ] Design clean facades for legacy systems

**Key Deliverable:** Each student has a complete TaskFlow API design document ready for Day 2 implementation.