Incremental APIs

Maciej Gowin @ CoderBrother

Maciej Gowin

- Head of Java Development @ Ryanair Labs
- Mentor @ CoderBrother.pl
- Lecturer "Programming in Java" @ WSB Merito

Agenda

- What is an API?
- Data exchange between systems
- What are HTTP and JSON?
- RestFul API basics
- API versioning options
- Incremental APIs
- Challenges

API

API (Application Programming Interface) acts as interface for communication which enabled different systems or applications to communicate with each other.

For example, when you use **Ryanair app**, it communicates with **web services** which serves data via an API to loads **a list of available routes**.

API

The term API can be used in the context of various communication channels:

- Web APIs
- Library APIs
- Operating System APIs

We will focus on the Web APIs.

Web API

A Web API:

- is an interface that allows applications or services to communicate with each other
- enables developers to integrate functionality, exchange data, or interact with external services

Web API Key Characteristics

- Communication Over HTTP/HTTPS
- Request/Response Model
- Endpoints

Frontend vs Backend

We typically use Web APIs in the context of web development, which encompasses both frontend and backend development.

Frontend is described by web applications and mobile apps.
Backend is describer by the back-end system serving data.

We will examine the communication between the frontend, represented by a website, and the backend, represented by a RESTful API service.

Web API types

- RESTful APIs
  - the most widely used
  - rely on standard HTTP methods (GET, POST, PUT, DELETE)
  - stateless
  - various data formats: JSON, XML
- SOAP (Simple Object Access Protocol) APIs
  - formalized
  - described by XML-based messaging protocol

Web API types

- GraphQL APIs
  - based on query language which allows to retrieve exact data specified by consumer
  - structure defined by the client
- WebSocket APIs
  - enable two-way communication
  - reduced latency

HTTP

- foundation of data communication on the World Wide Web
- protocol used for transmitting resources between a client (like a browser) and a server
- based on Request-Response Model
- stateless protocol, each request is independent

HTTP

HTTP Request

Contains a method (GET, POST, etc.), URL, headers, and sometimes a body (for POST requests).

HTTP Response

Contains a status code (like 200 for success or 404 for "not found"), headers, and the requested content (like HTML).

HTTP Methods

- **GET**: retrieves data from the server
- **POST**: sends data to the server (usually to create or update a resource)
- **PUT**: updates an existing resource on the server
- **DELETE**: removes a resource from the server

HTTP Status Codes

- **200 OK**: successful request
- **404 Not Found**: resource not found
- **500 Internal Server Error**: server-side error

JSON

- lightweight data-interchange format
- easy for humans to read and write
- easy for machines to parse and generate
- widely used in web development to transmit data between services
- organizes data into key-value pairs
- language agnostic

JSON

```
{
    "firstName": "Maciej",
    "lastName": "Gowin",
    "phone": {
        "callingCode": "48",
        "number": "693142041"
    },
    "height": 180,
    "dateOfBirth": "1986-11-21",
    "roles": [
        "developer",
        "mentor",
        "lecturer
    ]
}
```

RESTful API

- describes resources via URI
  - http://api.coderbrother.pl/customers for list of customers
- available operations defined by the HTTP verbs
  - GET to retrieve resource
- result of the operation defined by the HTTP status codes
  - 200 if operation was successful
- data described by common format
  - i.e. JSON
- stateless
  - each request is independent, information is not shared between requests

RESTful API Example

Operation	Resource	Description
GET	/customers	retrieve customers
GET	/customers/{teamId}	retrieve customers
POST	/customers	create customers
PUT	/customers/{teamId}	update customers
DELETE	/customers/{teamId}	delete customers

RESTful API example

```
GET https://api.coderbrother.pl/customers/abcd-1234
Header: Content-Type: application/json
```

```
Status: 200
Header: Content-Type: application/json
Body:
{
  "id": "abcd-1234",
  "name": "Maciej Gowin"
}
```

API evolution

- API evolves and may require changes
- API requires incremental changes

API evolution

Consider changes:

- What if we want to add date of birth to an existing API?
- What if we want to split the name and return separated first and last name?

API evolution

Add date of birth to the model - backward compatibility met

Additive changes are acceptable as long as they do not break existing contract.

```
GET https://api.coderbrother.pl/customers/abcd-1234
```

```
{
  "id": "abcd-1234",
  "name": "Maciej Gowin",
  "dateOfBirth": "1986-11-21"
}
```

API backward compatibility

Replace name with first and last name - backward compatibility not met

Breaking changes define new model.

```
GET https://api.coderbrother.pl/newCustomers/abcd-1234
```

```
{
  "id": "abcd-1234",
  "firstName": "Maciej",
  "lastName": "Gowin"
}
```

API backward compatibility

For incremental changes, we will implement API versioning when a new model is defined for an existing resource.

As an example:

```
GET https://api.coderbrother.pl/v1/customer/abcd-1234
GET https://api.coderbrother.pl/v2/customer/abcd-1234
```

API versioning

URI Versioning

```
/v1/customer/abcd-1234
```

Query Parameter Versioning

```
/customer/abcd-1234?version=1
```

API versioning

Header Versioning

```
/customer/abcd-1234

With a header:
Accept: application/vnd.api.v1+json
```

Subdomain Versioning

```
v1.api.coderbrother.pl/customer/abcd-1234
v2.api.coderbrother.pl/customer/abcd-1234
```

Choosing a Versioning Strategy

Clarity

How easily can users understand the versioning scheme?

Flexibility

Does the versioning method allow for easy changes and updates?

Backward Compatibility

How well does the versioning approach support clients using older versions of the API?

Simplicity

Is the versioning method simple to implement and maintain?

When API Can Be Deprecated?

- different approach driven by the consumer: mobile vs web
- unused versions can be removed
- use of tracing tools to verify the API usage

API vs Sprints

The API must be defined before frontend work begins, which undermines the principle of delivering functionality from the ground up during sprints.

Incremental API

Backend ahead of Frontend

API implementation starts one sprint before.

Model agreement

The team agrees on the API model and works in accordance with it. Model can be defined using tools such as Swagger.

Mocking

Quick API mocks representing the model. Mocks are delivered by tools such as Mockoon or JSON Server

APIs vs Agile methodologies

API planning and versioning for incremental changes are crucial from a technical point of view.

However, there are other challenges to API work in the context of Agile methodologies.

1. Defining Clear Requirements

Challenge: Translating business requirements into API user stories is tricky since APIs often don’t interface directly with users.
Solution: Collaborate with product owners to create technical stories and clarify the business value.

2. Incremental Delivery

Challenge: Delivering usable API increments each sprint is hard, especially when endpoints are interdependent.

Solution: Prioritize high-value endpoints and start with a minimal viable API (MVA).

3. Versioning and Backward Compatibility

Challenge: Frequent changes can break existing clients, making backward compatibility a concern.

Solution: Use versioning strategies and plan for backward compatibility in every sprint.

4. Managing Dependencies

Challenge: Coordinating between teams (frontend, backend) can delay progress.

Solution: Use API contracts and mock services to unblock teams.

5. Non-functional Requirements (NFRs)

Challenge: Addressing performance, security, and scalability in short sprints can be tough.
Solution: Plan NFRs into sprint work and conduct performance and security tests regularly.

6. Testing and Automation

Challenge: Thoroughly testing and automating API tests in each sprint is time-consuming.

Solution: Automate unit, integration, and performance tests using CI/CD pipelines.

7. Documentation

Challenge: Keeping API documentation up-to-date in fast-paced sprints is difficult.
Solution: Use automated tools (Swagger) to generate documentation alongside development.

8. Managing Technical Debt

Challenge: Rapid delivery can lead to technical debt, degrading API quality over time.

Solution: Allocate time in sprints for refactoring and technical improvements.

9. Coordinating Release Cycles

Challenge: Releasing new API versions must align with consumers (frontend, mobile) to avoid disruptions.

Solution: Use feature flags and communicate with teams to synchronize releases.

10. Changing Requirements

Challenge: Constant requirement changes can lead to design instability.

Solution: Use flexible API design patterns and maintain clear communication with stakeholders.

Q&A

Ask me anything.

Meet me at: https://linktr.ee/maciejgowin