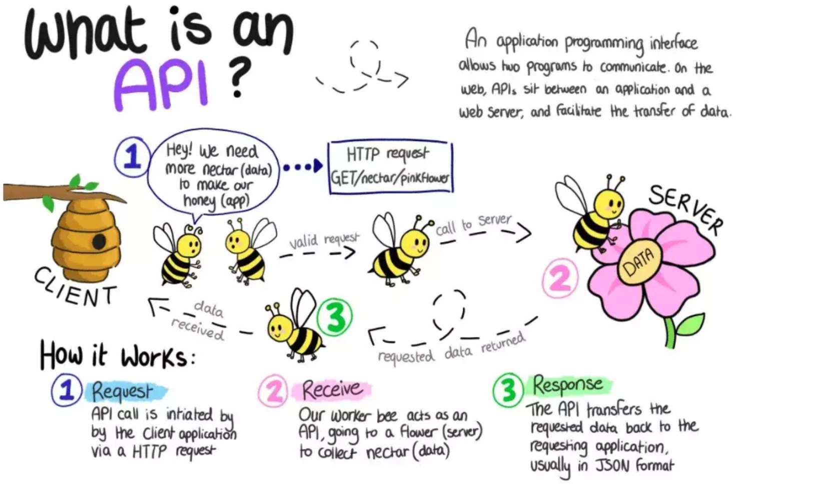
API-First Design

▼ What is an API?

- An API is a set of rules and protocols for building and interacting with software applications. It defines how different software components should interact, specifying the methods, data formats, and conventions to be followed.
- APIs enable communication between different software systems, allowing them to share data and functionality securely and efficiently.



An application programming interface allows two programs to communicate. On the web, APIs sit between an application and a web server, and facilitate the transfer of data.

1 Request: API call is intiated by by the Client application via a HTTP request

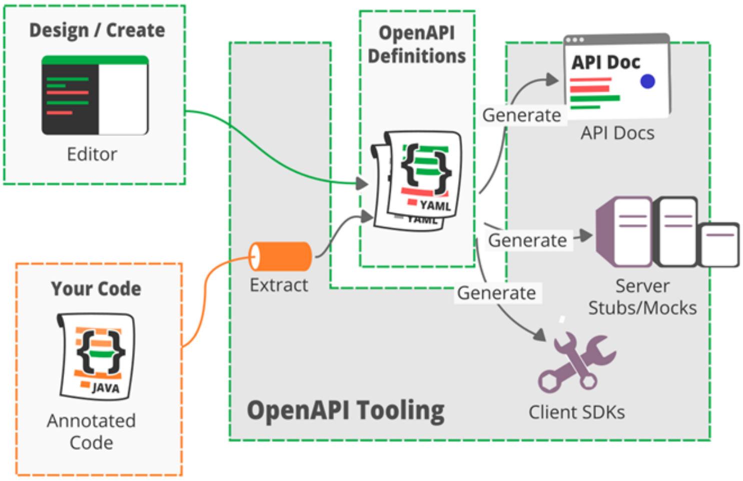
2 Receive: Our worker bee acts as an API, going to a flower (server) to collect nectar (data)

3 Response: The API transfers the requested data back to the requesting application, usually in JSON format

▼ API-First Approach:

- The API-first approach is a development methodology where APIs are designed and documented before any code is written for the underlying application or service.
- **Process:**
  - **Design Phase:** Developers and stakeholders collaborate to define the API's endpoints, request/response formats, error messages, and authentication methods.
  - **Documentation:** The API is thoroughly documented using specifications like OpenAPI (formerly Swagger), ensuring clarity and consistency.

- **Implementation:** Development teams use the API design as a contract, building their services to adhere strictly to the defined API specifications.



Why API-First Matters in Microservices:

- **Consistency Across Teams:**
    - **Unified Standards:** When multiple teams work on different microservices, starting with a well-defined API ensures everyone adheres to the same standards.
    - **Reduced Miscommunication:** Clear API contracts minimize misunderstandings between teams regarding data formats, endpoints, and expected behaviors.
- **Reduces Integration Risks:**
    - **Early Validation:** Designing the API upfront allows teams to identify and resolve potential integration issues before they become costly problems.
    - **Parallel Development:** Frontend and backend teams can work simultaneously. Frontend developers can use mock APIs based on the API specifications, accelerating the development process.

Analogy:

- **Blueprint of a Building:**
    - Just as architects create detailed blueprints before construction begins, software teams design APIs first to serve as a blueprint for development. This blueprint outlines how different components (rooms/services) connect and interact, ensuring the final structure (application) is cohesive and functional.

API design guide | Cloud API Design Guide | Google Cloud

A set of guidelines for designing APIs that are consistent with Google AIPs.

https://cloud.google.com/apis/design

Google Cloud API design tips | Google Cloud Blog

API design best practices maximize value and efficiency.

https://cloud.google.com/blog/products/api-management/google-cloud-api-design-tips

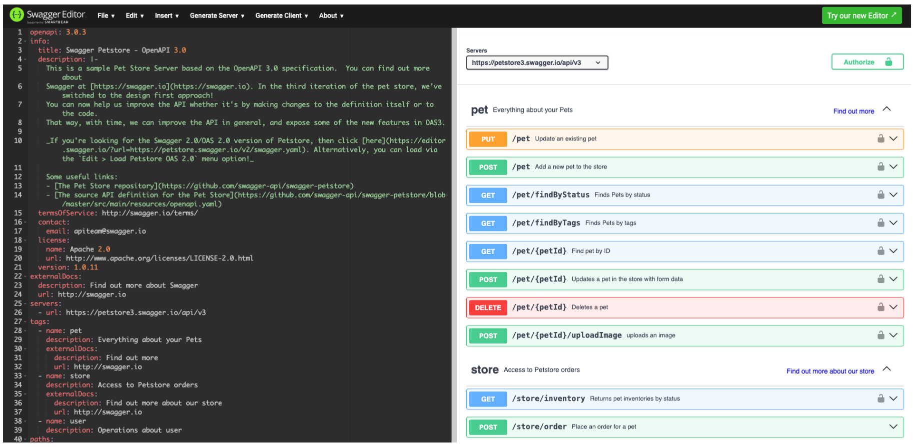
Benefits of API-First Design in Microservices

1. **Faster Development:**

- **Parallel Workstreams:**
    - Once the API is defined, backend and frontend teams can work independently.
    - Backend developers focus on service implementation, while frontend developers can use mock APIs to develop user interfaces.
- **Reduced Dependencies:**
    - Teams are less dependent on each other's timelines, leading to faster overall development cycles.

2. **Scalability:**
    - **Evolving Architecture:**
        - An API-first approach accommodates future changes. New features or services can be added without impacting existing ones.
    - **Modular Growth:**
        - Services can be scaled individually based on demand, improving resource utilization.
3. **Better Developer Experience:**
    - **Comprehensive Documentation:**
        - Well-documented APIs make it easier for developers to understand and integrate with services.
    - **Onboarding Ease:**
        - New team members or third-party developers can quickly get up to speed using the API documentation.

▼ Demo: `Swagger` Editor



```yaml
openapi: 3.0.3
info:
  title: Swagger Petstore - OpenAPI 3.0
  description: |-
    This is a sample Pet Store Server based on the OpenAPI 3.0 specification.  You can find out more about
    Swagger at [https://swagger.io](https://swagger.io). In the third iteration of the pet store, we've switched to the design first approach!
    You can now help us improve the API whether it's by making changes to the definition itself or to the code.
    That way, with time, we can improve the API in general, and expose some of the new features in OAS3.

    _If you're looking for the Swagger 2.0/OAS 2.0 version of Petstore, then click [here](https://editor.swagger.io/?url=https://petstore.swagger.io/v2/swagger.yaml). Alternatively, you can load via the `Edit > Load Petstore OAS 2.0` menu option!_

    Some useful links:
    - [The Pet Store repository](https://github.com/swagger-api/swagger-petstore)
    - [The source API definition for the Pet Store](https://github.com/swagger-api/swagger-petstore/blob/master/src/main/resources/openapi.yaml)
  termsOfService: http://swagger.io/terms/
  contact:
    email: apiteam@swagger.io
  license:
    name: Apache 2.0
    url: http://www.apache.org/licenses/LICENSE-2.0.html
  version: 1.0.11
externalDocs:
  description: Find out more about Swagger
  url: http://swagger.io
servers:
  - url: https://petstore3.swagger.io/api/v3
tags:
  - name: pet
    description: Everything about your Pets
    externalDocs:
      description: Find out more
      url: http://swagger.io
  - name: store
    description: Access to Petstore orders
    externalDocs:
      description: Find out more about our store
      url: http://swagger.io
  - name: user
    description: Operations about user
paths:
```

Servers
https://petstore3.swagger.io/api/v3

pet Everything about your Pets — Find out more

PUT `/pet` Update an existing pet
POST `/pet` Add a new pet to the store
GET `/pet/findByStatus` Finds Pets by status
GET `/pet/findByTags` Finds Pets by tags
GET `/pet/{petId}` Find pet by ID
POST `/pet/{petId}` Updates a pet in the store with form data
DELETE `/pet/{petId}` Deletes a pet
POST `/pet/{petId}/uploadImage` uploads an image

store Access to Petstore orders — Find out more about our store

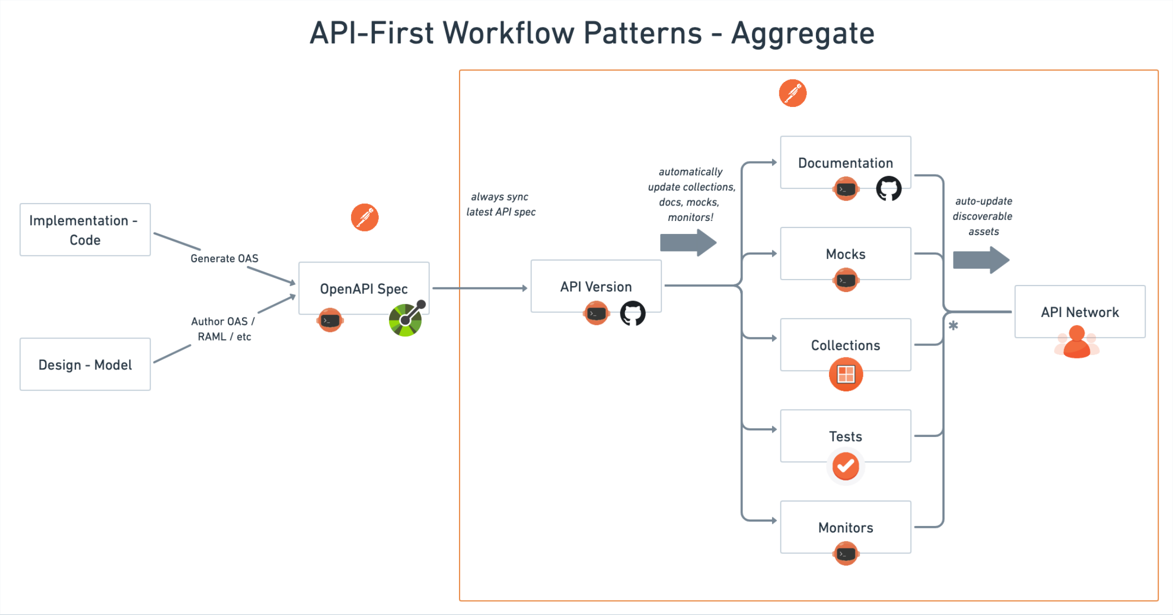
GET `/store/inventory` Returns pet inventories by status
POST `/store/order` Place an order for a pet

🧑🏽‍💻 OpenAPI and {Swagger}

- An interactive, web-based tool for creating and editing OpenAPI specifications.
- **Interactive API Design**: The left side of the editor allows users to define the OpenAPI (formerly Swagger) specification using YAML or JSON. This includes specifying:
    - **Endpoints**: Define paths (e.g., `/users`, `/products`).
    - **HTTP Methods**: Specify methods like `GET`, `POST`, `PUT`, `DELETE`.
    - **Request Parameters**: Define query, path, or body parameters for the API (e.g., `/users/{id}`).
    - **Request/Response Models**: Specify the structure of the data being sent and received by defining schemas, response codes, and data types.
- **Live Preview**: On the right side of the editor, users can immediately see:
    - **Interactive API Documentation**: This mimics how the final API documentation will look, and allows users to try out API requests directly from the documentation.
    - **Real-Time Updates**: Any changes made to the YAML or JSON on the left side are immediately reflected in the interactive documentation on the right.

   - **API Try-Out Functionality**: In the right-hand documentation, you can use the "`Try it out`" button to interact with mock APIs based on the current API design.

▼ The API-First Design Workflow


API-First Workflow Patterns - Aggregate
Implementation - Code
Design - Model
Generate OAS
Author OAS / RAML / etc
OpenAPI Spec
always sync latest API spec
API Version
automatically update collections, docs, mocks, monitors!
Documentation
Mocks
Collections
Tests
Monitors
auto-update discoverable assets
API Network

1. **Define API Contracts:**
   - **Use OpenAPI/Swagger:**
     - Specify endpoints, HTTP methods, request parameters, response formats, and error codes.
     - Ensure all stakeholders agree on the API's functionality and design.
   - **Benefits:**
     - Creates a clear agreement (contract) between teams.
     - Serves as a single source of truth for development and documentation.
2. **Mock APIs:**
   - **Purpose:**
     - Allow frontend developers to start building and testing against the API without waiting for the backend implementation.
   - **Tools:**
     - **Mock Servers:** Automatically generated from the API specification to simulate API responses.
     - **Mockoon or Stoplight:** Tools for creating local mock servers.
3. **Test APIs:**
   - **Automated Testing:**
     - Use tools like Postman or automated test suites to verify that the API behaves as specified.
   - **Continuous Integration:**
     - Integrate API tests into CI/CD pipelines to ensure ongoing compliance with the API contract.
4. **Implement APIs:**
   - **Backend Development:**
     - Developers implement the service logic, ensuring it adheres strictly to the API specification.
   - **Validation:**
     - Regularly test the implemented API against the contract to prevent deviations.

https://youtu.be/YRzpziA35Mg?si=9qALgG_9dU6YtcD4

Building Scalable APIs for Microservices

1. **Stateless Communication:**
   - **Definition:**
     - Each API request contains all the necessary information for the server to process it, without relying on stored context from previous requests.
   - **Benefits:**
     - Simplifies scaling because servers do not need to share session information.
     - Improves reliability and performance in distributed systems.
2. **Versioning:**
   - **Purpose:**
     - Allows APIs to evolve without breaking existing clients.
   - **Methods:**
     - **URI Versioning:** Including the version in the URL (e.g., `/v1/users`).
     - **Header Versioning:** Using custom headers to specify the API version.
   - **Best Practices:**
     - Deprecate old versions gracefully, providing clients time to migrate.
3. **Rate Limiting & Throttling:**
   - **Definition:**
     - **Rate Limiting:** Restricting the number of API calls a client can make in a given time frame.
     - **Throttling:** Controlling the flow of requests to ensure system stability.
   - **Benefits:**
     - Protects services from being overwhelmed by excessive requests.
     - Ensures fair usage among all clients.
4. **Load Balancing:**
   - **Purpose:**
     - Distributes incoming network traffic across multiple servers.
   - **Benefits:**
     - Enhances availability and reliability.
     - Improves response times and resource utilization.

Case Study: Netflix

Netflix is renowned for pioneering the use of **microservices** in modern software architecture, and their approach to building **scalable APIs** has become a benchmark for handling large-scale distributed systems.

Api Gateway – Netflix TechBlog

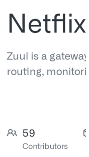
Read writing about Api Gateway in Netflix TechBlog. Learn about Netflix's world class engineering efforts, company culture, product developments and more.

https://netflixtechblog.com/tagged/api-gateway

GitHub - Netflix/zuul: Zuul is a gateway service that provides dynamic routing, monitoring, resiliency, security, and more.

Zuul is a gateway service that provides dynamic routing, monitoring, resiliency, security, and more. - Netflix/zuul

https://github.com/Netflix/zuul

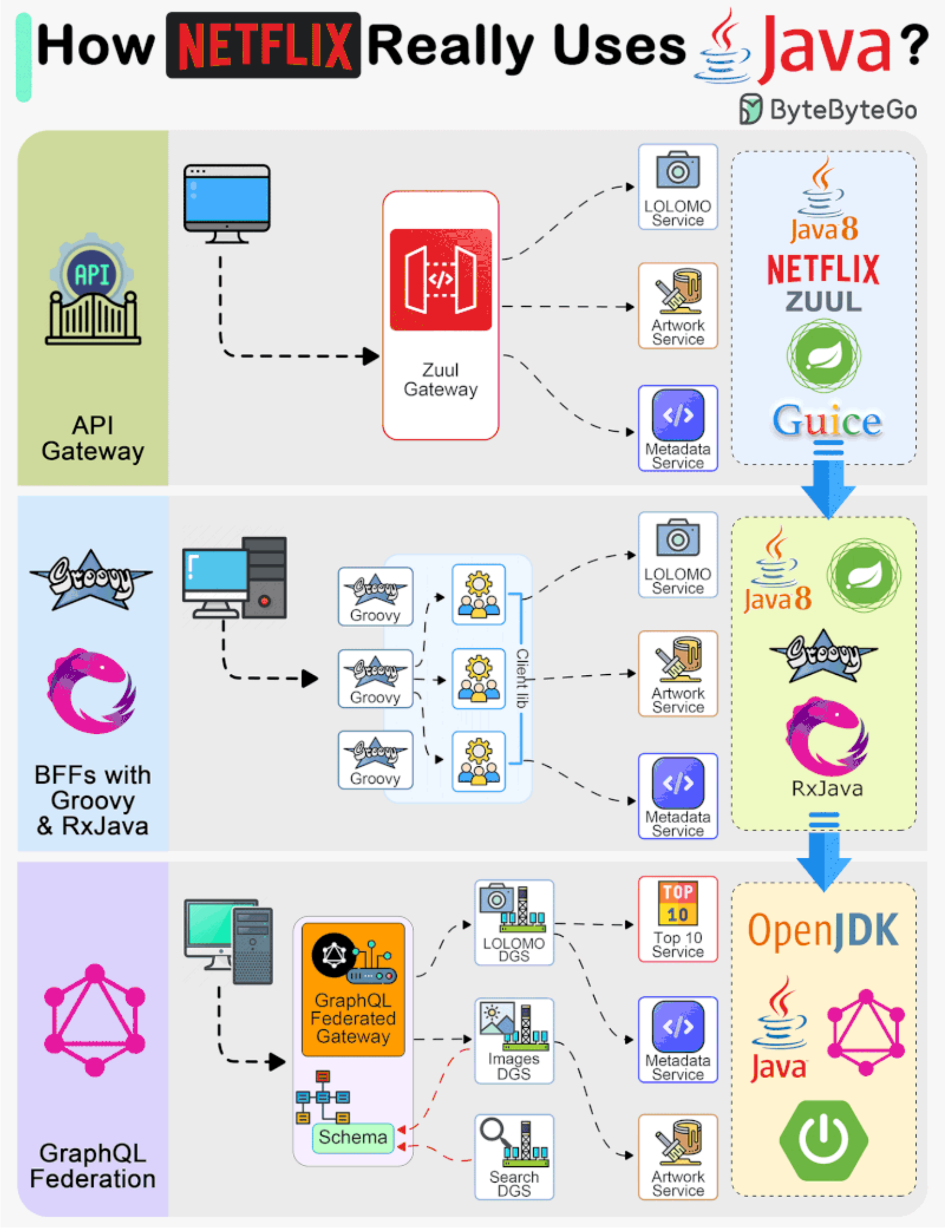


Netflix

Zuul is a gateway routing, monitori

59 Contributors

https://youtu.be/CZ3wIuvmHeM?si=vPYbhwWKYU-9Uz-2



▼ API Design Best Practices

1. **Meaningful Resource Names:**

- **Guidelines:**
  - Use nouns to represent resources (e.g., `/users`, `/orders`).
  - Avoid verbs in endpoint names (e.g., `/createUser` should be `/users` with a POST method).

- **Benefits:**
    - Improves readability and intuitiveness of the API.
    - Aligns with RESTful principles.

2. **HTTP Methods:**
    - **Standard Methods:**
        - **GET:** Retrieve resource(s).
        - **POST:** Create a new resource.
        - **PUT:** Update an existing resource (or create if it doesn't exist).
        - **PATCH:** Partially update a resource.
        - **DELETE:** Remove a resource.
    - **Idempotency:**
        - Methods like GET, PUT, and DELETE should be idempotent (same result regardless of how many times they're called).
3. **Error Handling:**
    - **Consistent Responses:**
        - Provide meaningful error messages in a standard format (e.g., JSON with an `error` object).
    - **HTTP Status Codes:**
        - Use appropriate status codes:
            - **200 OK:** Successful request.
            - **201 Created:** Resource successfully created.
            - **400 Bad Request:** Invalid request parameters.
            - **401 Unauthorized:** Authentication required.
            - **403 Forbidden:** Insufficient permissions.
            - **404 Not Found:** Resource not found.
            - **500 Internal Server Error:** Generic server error.
4. **Authentication & Authorization:**
    - **OAuth2:**
        - An industry-standard protocol for authorization.
        - Allows users to grant limited access to their resources on one site to another site without sharing credentials.
    - **JWT Tokens (JSON Web Tokens):**
        - A compact, URL-safe means of representing claims to be transferred between two parties.
        - Commonly used for authentication and information exchange.
    - **API Keys:**
        - Simple tokens that are passed in the request header or query parameters.
        - Suitable for identifying the application or client making the request.

▼ API Security Considerations in Microservices

1. **Authentication:**
    - **OAuth2:**
        - Provides secure delegated access using access tokens.
        - Suitable for third-party access scenarios.
    - **JWT Tokens:**

        - Self-contained tokens with embedded user information.
        - Stateless, eliminating the need for server-side sessions.

2. **Rate Limiting:**
    - **Implementation:**
        - Define thresholds for request rates per API key or IP address.
        - Use tools or middleware to enforce limits.
    - **Benefits:**
        - Protects against DoS attacks.
        - Ensures fair resource allocation.
3. **Input Validation:**
    - **Purpose:**
        - Prevent malicious data from compromising the system.
    - **Best Practices:**
        - Validate data types, formats, and ranges.
        - Use allowlists (preferred over denylists) for permitted values.
        - Sanitize inputs to remove or escape harmful characters.
4. **HTTPS Everywhere:**
    - **Encryption:**
        - Use TLS (Transport Layer Security) to encrypt data in transit.
    - **Benefits:**
        - Protects sensitive information like authentication tokens and personal data.
        - Prevents man-in-the-middle attacks.
