2026 HackIllinois Track: Best Web API

Goal: Build a well-crafted API, focusing on the surface, developer experience, and correctness. Focus on clarity, predictable behavior, and useful documentation that will delight developers.

Overview

Application programming interfaces (APIs) are everywhere. They are software that enable users, developers, and computers to exchange data between each other. 15 years ago, Stripe started with an API that let you create a card transaction with just 7 lines of code. Today, we want your team to build an API that delivers a delightful experience.

Submission Requirements

- Build an API with one or more endpoints that performs a valuable action or exposes useful data.
    - Note: Frontend work is optional; submissions will be judged on the quality of the API itself.
    - Minimum requirement: users should be able to interact with it using tools such as cURL/Postman.
- The API must be queryable over HTTP
    - A minimum of one endpoint is required, though additional endpoints are permissible.
    - Minimum requirement: The API must be operational on localhost.
    - **Bonus consideration:** The API is publicly accessible.
- Documentation and usage examples must be provided to enable a developer to effectively call and test the API.
    - Minimum requirement: Documentation within the repository's readme file.
    - **Bonus consideration:** Provision of a hosted documentation page.
- Guidelines for evaluation:
    - The submission should demonstrate innovation, utility, or entertainment value.
    - Endpoints must return successful responses (HTTP 200 or equivalent) when provided with expected data.
    - Adherence to standards and best practices is encouraged, including the return of informative error messages/statuses and intuitive handling of edge cases.
    - **Bonus consideration:** The API incorporates methods beyond GETS; uses state in an interesting way.

Acceptable and Encouraged API Types

- The API may consist of multiple endpoints – for instance, a POST endpoint to facilitate object creation and a corresponding GET endpoint to retrieve detailed information about that object.
    - Consider the user's potential objectives and how your APIs can serve as foundational components for achieving those goals.
- Various API types are permissible, including (but not limited to, as referenced in this Postman blog): REST, gRPC, and GraphQL.
- Qualifying examples and ideas to get you started:
    - A curated GitHub list of public APIs
    - Predominantly GET-oriented examples (i.e., minimal or no write operations):
        - Wrappers for university resources (e.g., availability of facilities like restrooms or classrooms).
            - Example: Pre-calculating certain data points and making them accessible via an API.
        - Quote retrieval services, such as https://breakingbadquotes.xyz/.
        - Even straightforward utilities, such as a calculator function.
    - Stateful examples, likely incorporating POST operations:
        - These may be simpler to construct by wrapping or combining existing APIs (e.g., integrating a calendar service with a large language model like ChatGPT).
        - Existing analogues include: URL shortening services, weather data APIs, PasteBin/Gist implementations, and counter mechanisms (e.g., to track webpage visits).

Evaluation Metrics

- **Functionality**
    - Does the API work as described? Do endpoints return expected results for valid inputs (HTTP 200/2xx)?
    - Are error conditions handled and surfaced with appropriate status codes?
- **Usefulness & Creativity**
    - Is there a clear user or developer use case? Is the API solving a meaningful problem or providing a unique capability, even if it’s niche?
- **API design & attention to detail**
    - **API Design**
        - Consistent and logical naming of endpoints and resources.

        - Implementation of pagination, filtering, and search functionalities where appropriate.
        - Idempotency and predictable behavior for operations that modify state.
    - **Attention to detail**
        - Consistent and logical endpoint/resource naming.
        - Are various usage scenarios thoroughly considered and addressed?
            - Ex: If there is a POST endpoint to create some backend object, is there a corresponding GET endpoint? If we expect users to want a list of many objects, is there a way to do so (LIST or get_many)?
            - Ex: For an API managing persons, is searching supported by criteria other than just an identifier, such as by name or email?
- **Documentation & Developer Experience**
    - Can users readily figure out how to use the API? (Poor usability is a critical flaw)
    - If something goes wrong, are users able to easily figure out why?
    - Explanation of the tech stack employed.

Stripe Prizes:

- 1st Prize: $2000 for the team + JBL headphones for each team member
- Honorable mention: $500 for the team + $100 Amazon Gift Card for each team member