



Designing APIs

🕒 Created	@February 13, 2022 11:12 PM
▼ Type	Lecture
# Week	8
☰ Lecture #	1
🔗 Lecture URL	https://youtu.be/9IM2zpiWQ7A
🔗 Notion URL	https://21f1003586.notion.site/Designing-APIs-I-1e8ec3628dac4601b70cbb3797026197

What makes a good API?

- There is no strict consensus
- A ton of conflicting opinions

Review of ideas from <https://cloud.google.com/apigee>

Web API Design: The Missing Link

- eBook from Apigee: useful guidelines

Purpose of an API

- To be used to build applications
- Design APIs with developers in mind

Remote Procedure Call → used to connect to a remote server and retrieve data based on some function arguments

Data oriented approach

Entities

- Students
- Courses
- Grades

Actions

- Add, Edit, Delete

Summaries

- List
- Grade point average
- Top students in courses
- ...

Exotic

- Students with names beginning with 'A' who score more than 85% in MAD1

Possibilities

- List of students
 - `http://localhost/getListOfStudents`
- Individual student
 - `/getStudent?id=xyz`
- Add new student
 - `/createNewStudent`
- Edit existing student

- /editStudent?id=xyz

Can get pretty complex

- /getTopStudents
- /createStudentAndAddToCourse

Not fundamentally wrong

- Difficult to remember
- Challenging to document, understand


Use conventions

- List of students
 - http://localhost/students
- Individual student
 - http://localhost/student/123
- Add new student
 - POST .../student
- Edit existing student
 - PATCH .../student/123

URL conventions

- Nouns in the URL are good, verbs are bad
 - /student
 - PATCH /student/123

instead of

- /create/student
- /edit/student/123
- Verbs in HTTP → use the Method
- well-known URLs
 - Can API be “discovered” by crawling from  ?
 - Standard conventions for listing, posting, etc.

- Permalinks
 - not necessarily human readable
 - unique ID for posts, documents, etc

Query URLs

- /search?course=123&type=student

OR

- /course/123/students

Convention → structured URLs preferred by developers

Why not /course-123-students?

- Can also be used — either way the URL path has to be parsed
 - Same complexity — just difference in developer experience
-

Lecture #2 starts here

Lecture URL: <https://youtu.be/VM-2WK2KG4I>

HTTP Verbs

- GET
 - read data, lists etc
 - cacheable — all data is in the URL
- POST
 - create new object/data
 - Not cacheable in general — data not part of the cache index
 - Can be used for reading data
- PUT / PATCH
 - Update with all the new data / incremental new data → PATCH to be preferred

These are all conventions

Output formats

- Structured data
 - XML → very good
 - JSON → not so good — very limited data types
- Simplicity
 - JSON → Human readable
 - Easy to parse — but can have problems
- JSON + extensions is preferred format at present
 - Not necessarily the best possible

Included links

- Just seeing JSON output, cannot know the query
- Include additional links that give pointers to other useful information
 - Could have been done as part of documentation
 - In some cases more flexible to provide with response

Example → GitHub API

API URL → <https://api.github.com/users/kashifulhaque>

Response

```
{
  "login": "kashifulhaque",
  "id": 37375667,
  "node_id": "MDQ6VXNlcjM3Mzc1NjY3",
  "avatar_url": "https://avatars.githubusercontent.com/u/37375667?v=4",
  "gravatar_id": "",
  "url": "https://api.github.com/users/kashifulhaque",
  "html_url": "https://github.com/kashifulhaque",
  "followers_url": "https://api.github.com/users/kashifulhaque/followers",
  "following_url": "https://api.github.com/users/kashifulhaque/following{/other_user}",
  "gists_url": "https://api.github.com/users/kashifulhaque/gists{/gist_id}",
  "starred_url": "https://api.github.com/users/kashifulhaque/starred{/owner}/{/repo}",
  "subscriptions_url": "https://api.github.com/users/kashifulhaque/subscriptions",
  "organizations_url": "https://api.github.com/users/kashifulhaque/orgs",
  "repos_url": "https://api.github.com/users/kashifulhaque/repos",
  "events_url": "https://api.github.com/users/kashifulhaque/events{/privacy}",
  "received_events_url": "https://api.github.com/users/kashifulhaque/received_events",
  "type": "User",
  "site_admin": false,
  "name": "Kashif",
  "company": null,
```

```
"blog": "https://ifkash.pages.dev",
"location": "Kolkata, India",
"email": null,
"hireable": true,
"bio": "Data Science undergrad @ IIT Madras • I occasionally contribute to @k2-labs ",
"twitter_username": "notifkash",
"public_repos": 44,
"public_gists": 4,
"followers": 24,
"following": 83,
"created_at": "2018-03-14T15:07:06Z",
"updated_at": "2022-02-05T13:07:11Z"
}
```

Authentication

- Token based authentication
- OAuth2 → Good standard used in many places
- JWT (JSON Web Tokens) and other variants possible

Use standard techniques wherever possible

Summary

- Good API design requires experience
- Mostly based on conventions → no rigid rules
- But conventions are important

APIs are designed for developers, not end users

Problems with REST

- Most RESTful APIs are violating some constraint of REST
- REST is an architecture style → Not an API design document
 - Not rigid guidelines — sometimes bending the rules can help
- Chatty — Multiple requests to fetch data for a view
 - First get details of student, the list of courses taken by the student, then details for each course, then aggregate marks in each course ...
- Specific requests permitted — not a general “Query Language”
- Cannot specify “what is needed” — need to break up into individual requests then get the results



Intro to GraphQL

🕒 Created	@February 13, 2022 11:42 PM
▼ Type	Lecture
# Week	8
☰ Lecture #	3
🔗 Lecture URL	https://youtu.be/x4eZPjzM92Q
🔗 Notion URL	https://21f1003586.notion.site/Intro-to-GraphQL-8e4d3ada11d741df8054e6d102f2066c

Why GraphQL

- REST based APIs are endpoint based
 - Specific types of queries permitted
 - Complex data requests must be constructed with multiple GETs
 - `/student?name='A'&age='lt_25'`
 - Special characters? arbitrary queries?
- Multiple data sources

- Modern sites require inputs from multiple sources
- Simultaneous query and fusion of data — at client or at server?
- Declarative programming — what to do, not how to do it
 - Very useful in view construction
 - Improves developer experience
 - Why not for retrieving data as well?

How GraphQL?

How to GraphQL: <https://www.howtographql.com/>

- Engine on the server side to handle requests
- Translate requests in a complex query language to data requests
 - Collect data, filter etc. on the server
 - Respond to the client only with the data needed

What is GraphQL?

- Query language
- Can be used over HTTP
 - Usually with POST
- Send complex queries over POST body

Layer between client and server

- Receives complex queries
- Convert to (multiple) queries to the server, fuse results

Type system

- Specify types of query items
 - String, Int, Collection of items, etc
 - Automatically catch and prevent certain query errors
- Specify relations between items
 - Student → [Course]: student can have list of course

API versioning: Evolve

- Requests are JSON-like
- Add functionality as required
- Deprecate functionality if needed
- Not necessary to define new API versions in most cases

Mutation

- Create/Update/Delete type operations
- Generalized to be any kind of query
 - Alter the underlying data store

Tools

- Apollo server → system to build up GraphQL
 - Connect to multiple backends
 - Define own resolves
- Explorers
 - Google, github, graphql.org
 - dynamically construct and test queries

Summary

- Extension of core API concepts
- Integrate with multiple data sources
- Complex query language
 - Filter data at the server end, send only relevant data to the client
- Does not necessarily reduce server complexity
 - Server may even become more complex
 - But for complex queries this might have been needed anyway



Markup Alternatives

🕒 Created	@February 13, 2022 11:54 PM
▼ Type	Lecture
# Week	8
☰ Lecture #	4
🔗 Lecture URL	https://youtu.be/Po4hmc3-vH4
🔗 Notion URL	https://21f1003586.notion.site/Markup-Alternatives-81e50c1d2d594abaa6faa4f0913bed4b

Why HTML?

- General enough markup for all text
- “Living standard”
 - Can adapt to many new functional requirements
- Extensible
 - WebComponents, JS enable new tags if needed

Focus on “semantic” content → leave styling to CSS

Why not HTML?

- Structured data communication
 - JSON often used, but not really designed for this
 - XML etc. much better, but overkill
- Virtual Reality, new environments
 - VRML
 - X3D
- Still relatively verbose for humans

Text-based markup

- Write almost like normal text
- Use inline markers: **strong**, *emphasised*, # HEADING
- Many alternatives
 - Markdown
 - ReStructured Text (RST) → documentations
 - AsciiDoc

Markdown Example

```
# This is a heading

And this is a regular paragraph

* A bullet list
* Another bullet list

1. A numbered list
2. More numbers

And
[links](https://www.example.com)
```

Why text?

- Uniform character representations agreed upon - ASCII, Unicode
 - Write once, read anywhere
- Guard against obsolescence

- Old file formats not readable
- Not easy way to reverse engineer
- Compact
- Easy for humans, also to read

Why not text?

- Hard to encode “structure”
- Ambiguity possible
 - Parsing the format may not be unique
- More focused towards English and Roman alphabet
 - Possible to create reasonable equivalents in other languages/scripts

Compile/Convert

- Systematic conversion between markup formats
 - Just like any other language \rightarrow language conversion
- Easier between structured languages
 - XML - SGML etc
- Custom compilers

Pandoc: “Swiss Army Knife” tool to convert between formats

Mixed functionality

- Programs mixed with documentation
 - Web/Weave, Doxygen with similar comment-oriented systems
- JSX, Vue
 - Mix JS with templates and HTML structure

Summary

- Markup design has lots of constraints
 - Structured data
- Focus more on human interface

- Understandable markup
 - Easy to write and transmit
- Compilers/Converters to handle interchange



JAM Approach

🕒 Created	@February 14, 2022 12:09 AM
▼ Type	Lecture
# Week	8
☰ Lecture #	5
🔗 Lecture URL	https://youtu.be/YrcQ_aD3rYg
🔗 Notion URL	https://21f1003586.notion.site/JAM-Approach-4dc672223eac4cdb84587a164c43e3cb

JAM

- JavaScript
- APIs
- Markup

What does an app need?

- Data store → What the app is for

- Access and retrieval → APIs
- SQL, NoSQL, GraphQL
- User Interface → to interact with the user
 - Vanilla HTML + forms → request/response
 - JavaScript → interactivity, closer to native
- Business logic → what should be done with the data
 - Backend computation → Python, Go, Node.js
 - Frontend computation → JavaScript, WebAssembly (WASM)

Content Management Systems

Example → Blog application

- CRUD for posts, comments
- Ratings for posts, comments
- User management
- Analytics

All these are data manipulation → can be independent of the UI

Wordpress

- One of the oldest and most popular
- Handles both data storage (backend) and templating (frontend)
- Also provides API → <https://developer.wordpress.org/rest-api/>

API can be used to build a CMS without frontend

Static Site Generators

- NextJS, NuxtJS, Gatsby
 - JS based → useful for interactive sites, complex designs, plugins
- Jekyll, Hugo
 - Primarily text oriented
 - Blogs, homepages

Why Static Site Generators (SSGs)?

- Servers can focus on delivering content
 - Static files faster to fetch
 - “compile time” optimization to reduce the file transfer

“First Contentful Paint”

- Pure HTML allows easy transfer and parsing, can be displayed quickly

JS Hydration

- Static HTML transferred from server — no interactivity
- “Hydrate” the HTML with event handlers
 - Inject interactivity after initial rendering is complete
- Delayed, but still fast enough
 - Good combination of speed and interactivity

JAMStack → pinnacle of web app development?

- Takes care of storage + logic + presentation type apps
 - APIs flexible enough to handle any backend
 - Markup easy to change or compile
 - JS powerful enough to emulate any other behaviour
- Other developments?
 - Real-time communications
 - New interface devices, displays
- JAM approach general enough to extend with APIs
 - Until hit by performance issues
 - Wait for the Next Big Thing