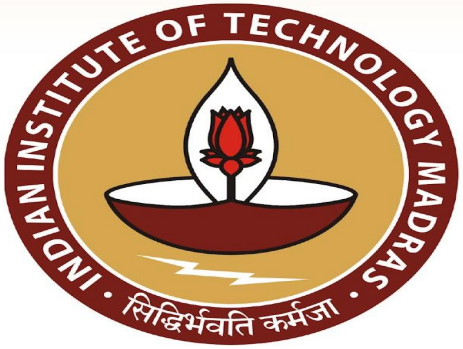

IIT Madras
BSc Degree

Web Apps: Review

Designing APIs

What makes a good API?

- No strict consensus
- Lots 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

Web API

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 scored more than 85% in …

Possibilities

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

Can get complex

- /getTopStudents
- /createStudentAndAddToCourse

Not fundamentally wrong!

- Hard to remember
- Hard to document, understand

Can get complex

- /getTopStudents
- /createStudentAndAddToCourse

Not fundamentally wrong!

- Hard to remember
- Hard 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 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
  - docs.google.com/presentation/d/1r3scu9VKrE4jKHtcsZ88ZMCzQntv3M1kKamHQngNbjM

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

HTTP Verbs

- GET:
  - read data, lists etc
  - cacheable - all data is in URL
- POST:
  - create new object/data
  - Not cacheable in general - data not part of cache index
  - Can be used for reading data
- PUT / PATCH
  - Update with all 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 best possible

Included links

- Just seeing JSON output, cannot know 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


```
{
  "login": "nchandra75",
  "id": 3119001,
  "url": "https://api.github.com/users/nchandra75",
  "html_url": "https://github.com/nchandra75",
  "followers_url": "https://api.github.com/users/nchandra75/followers",
  "following_url":
"https://api.github.com/users/nchandra75/following{/other_user}",
  "gists_url": "https://api.github.com/users/nchandra75/gists{/gist_id}",
  "starred_url":
"https://api.github.com/users/nchandra75/starred{/owner}{/repo}",
```

Authentication

- Token based authentication
- OAuth2 : good standard used in many places
- JWT (JSON Web Token) and other variants possible

Use standard techniques where 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, then list of courses taken by student, then details of 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 steps of how to get the result

GraphQL

Why?

- 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?

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

- Receive complex queries
- Convert to (multiple) queries to 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 courses

API versioning: Evolve

- Requests are JSON-like
- Add functionality as required
- Deprecate functionality if needed
- Not necessary to defined new API version 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 resolvers
- 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 server end, send only relevant data to client
- Does not necessarily reduce server complexity
  - Server may even become more complex
  - But for complex queries this may have been needed anyway

Markup Alternatives

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): documentation
  - AsciiDoc

Markdown example:

```
# This is a heading

And this is a regular paragraph.

* A bullet list
* Another bullet

1. A numbered list
2. More numbers

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

Why text?

- Uniform character representations agreed upon - ASCII, Unicode
  - Write once, read anywhere
- Guard against obsolescence
  - Old file formats not readable
  - No 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 <-> 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 and 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

- 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, NodeJS, …
  - Frontend computation: JS

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 user interface!

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, home pages

Why SSGs?

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

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 communication
  - New interface devices, displays

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 communication
  - New interface devices, displays
- JAM approach general enough to extend with APIs
  - Until hit by performance issues
  - Wait for the Next Big Thing!