CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Lecture 9: API Design

Gregory Gay
TDA 594/DIT 593 - November 29, 2022

Implementing Features

- Features often implemented as standalone, interchangeable services.
- Services generally accessed through an API.
  - Machine-accessible interface.
  - e.g., REST interface.
- Good API design is crucial to creating a feature reusable in many clients.

REST

- REST is a HTTP-based API format.
  - Services provide resources (through URLs) designed to be consumed by programs rather than by people.
  - Design Principles:
    - Stateless
    - Resource-Based
    - Uniform Interface
    - Links describe relationships
    - Cacheable and monitorable using standard internet tools

Today's Goals

- Creating REST APIs.
- REST design practices.
- Designing reusable APIs.
  - Features that can be substituted for other features.

Hypertext Transfer Protocol (HTTP)

HTTP

- Communication protocol for networked systems.
  - Defines how to exchange or transfer hypertext between nodes in a network.
  - How your computer accesses a webpage.
- Defines an API based on requests.
- Requests performed using **verbs**.
  - I **get** a page, **post** an update, **delete** a photo, **put** up information.

Retrieving Information (GET)

- User types into the browser: http://www.amazon.com
- The browser creates an HTTP request (no body)
- The HTTP request identifies:
  - The desired action: GET ("get me resource")
  - The target machine (www.amazon.com)

Updating Information (POST)

- The user fills in a form on a webpage.
- The browser creates an HTTP request with a body containing form data.
- HTTP request identifies:
  - The action: POST ("here is some updated info")
  - The target machine (amazon.com)
- The body contains data being POSTed (form data)

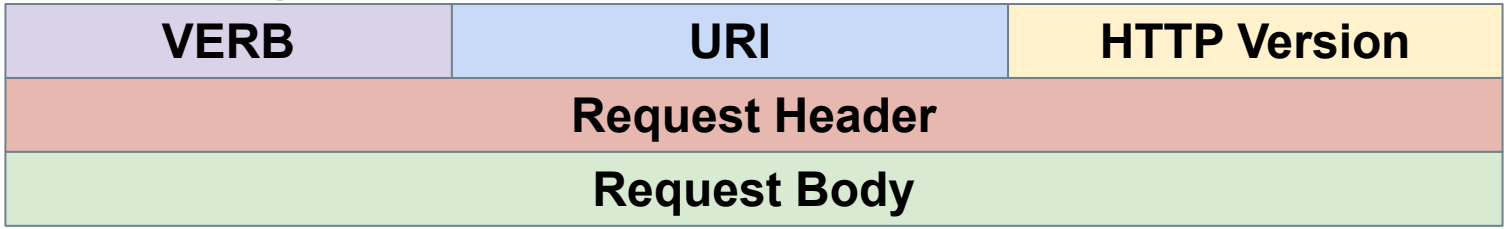
The HTTP API

- Simple set of operations based on CRUD (Create, Retrieve, Update, Delete)
  - **PUT:** “Insert info at this location” (Create/Update)
    - Modifies an existing resource or creates a new one at endpoint.
  - **GET:** “Give me some info” (Retrieve)
  - **POST:** “Create a new subordinate resource at this location” (Create/Update)
    - E.x., add a new message board post.
  - **DELETE:** “Get rid of this info” (Delete)

Additional Verbs

- **HEAD**
  - “Give me the metadata”
- **TRACE**
  - “Show me what changes have been made”
- **OPTIONS**
  - “What verbs have you implemented for this resource?”
- **PATCH**
  - “Apply partial resource modification”

Anatomy of an HTTP Request



- <VERB> is one of the HTTP verbs, <URI> is the resource location
- <Request Header> contains metadata
  - Collection of key-value pairs of headers and their values.
  - Information about the message and its sender like client type, the formats client supports, format type of the message body, cache settings for the response, and more.
- <Request Body> is the actual message content (JSON, XML).

HTTP Request Examples

VERB	URI	HTTP Version
Request Header		
Request Body		

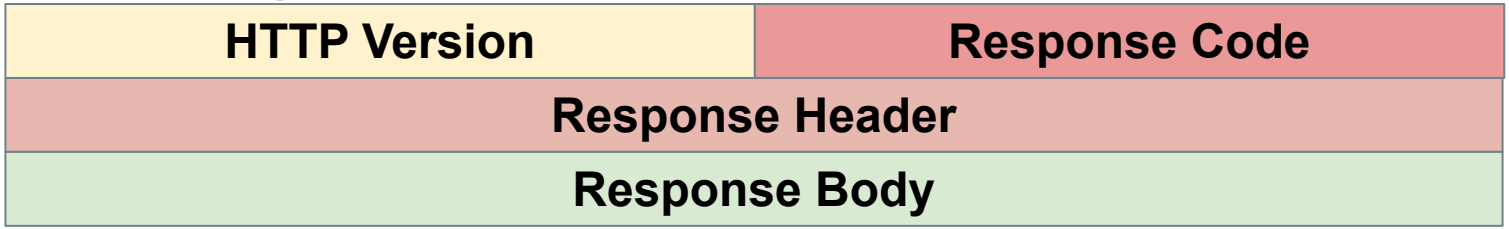
GET:

```
GET http://www.w3.org/Protocols/rfc2616/rfc2616.html HTTP/1.1
Host: www.w3.org, Accept: text/html,application/xhtml+xml,application/xml; …,
User-Agent: Mozilla/5.0 (Windows NT 6.3; WOW64) AppleWebKit/537.36 …,
Accept-Encoding: gzip,deflate,sdch, Accept-Language: en-US,en;q=0.8,hi;q=0.6
```

POST:

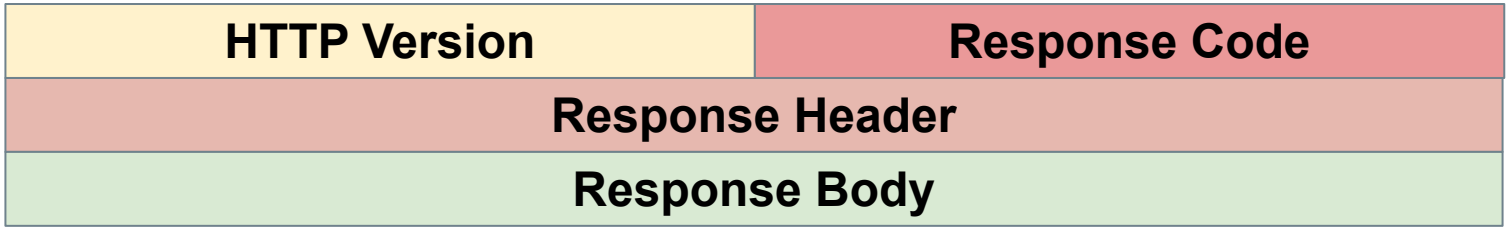
```
POST http://MyService/Person/ HTTP/1.1
Host: MyService, Content-Type: text/xml; charset=utf-8, Content-Length: 123
<?xml version="1.0" encoding="utf-8"?>
<Person><ID>1</ID><Name>M Vaqqas</Name>
<Email>m.vaqqas@gmail.com</Email><Country>India</Country></Person>
```

Anatomy of an HTTP Response



- <Response code> contains request status. 3-digit HTTP status code from a pre-defined list.
- <Response Header> contains metadata and settings about the response message.
- <Response Body> contains the representation if the request was successful.

HTTP Response Example



HTTP/1.1 200 OK

Date: Sat, 23 Aug 2014 18:31:04 GMT, Server: Apache/2, Last-Modified: Wed, 01 Sep 2004 13:24:52 GMT, Accept-Ranges: bytes, Content-Length: 32859, Cache-Control: max-age=21600, must-revalidate, Expires: Sun, 24 Aug 2014 00:31:04 GMT, Content-Type: text/html; charset=iso-8859-1

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd"> <html xmlns='http://www.w3.org/1999/xhtml'> <head><title>Hypertext Transfer Protocol -- HTTP/1.1</title></head> <body> …

HTTP Status Codes

- Common Responses:
  - 200 Ok (succeeded)
  - 201 Created (a new resource)
  - 202 Accepted (not completed)
  - 204 No Content (fulfilled request, nothing to return)
  - 205 Reset content (reload page)
  - 301 Redirection: moved permanently
  - 400 Bad request
  - 401 Unauthorized
  - 404 Not found

Representational State Transfer (REST)

Representational State Transfer

- A Client references a resource using a URI (endpoint).
- A **representation** of the resource is returned.
  - Receiving the representation places the client in a new **state**.
- When user selects a link in `Boeing747.html`, it accesses another resource. New representation places client into another state.
  - Client application **transfers** state with each resource access.

The Core Idea

- REST modeled after natural workflow of the net.
  - Design pattern for web services.
  - A well-designed web app behaves as a network of web pages (a virtual state-machine).
  - User progresses through application by selecting links (state transitions).
    - Resulting in next page (the next state) being transferred to the user and rendered.

REST Fundamentals

- Services offer **resources**.
- All resources have a unique **URI**.
  - URIs tell a client that there's something that can be interacted with.
- HTTP **verbs** are used to retrieve or manipulate resources in a clear, universal manner.

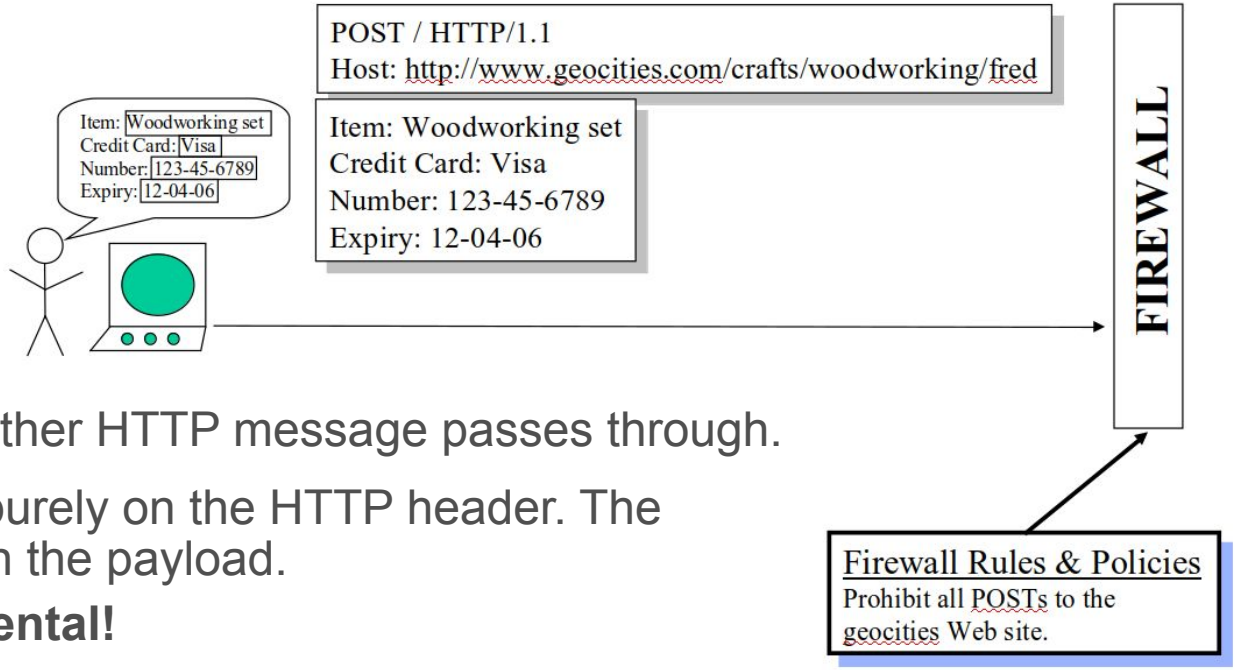
Verb Guarantees

- GET, OPTIONS, TRACE, and HEAD are **safe.**
  - Should not change the resource in any way.
  - Be careful - no technical limitations ensuring safety.
- PUT and DELETE are **idempotent**.
  - Repeated requests have same effect as a single request.
  - Safe operations are also idempotent.
  - POST is *not* idempotent.

Elements of Web Architecture

- **Firewalls** decide which HTTP messages get out, and which get in.
  - These components enforce web *security*.
- **Routers** decide where to send HTTP messages.
  - These components manage web *scalability*.
- **Caches** decide if saved copy of resource used.
  - These components increase web *performance*.

Firewalls



- Firewall decides whether HTTP message passes through.
- All decisions based purely on the HTTP header. The firewall never looks in the payload.
  - **This is fundamental!**
- This message is rejected.

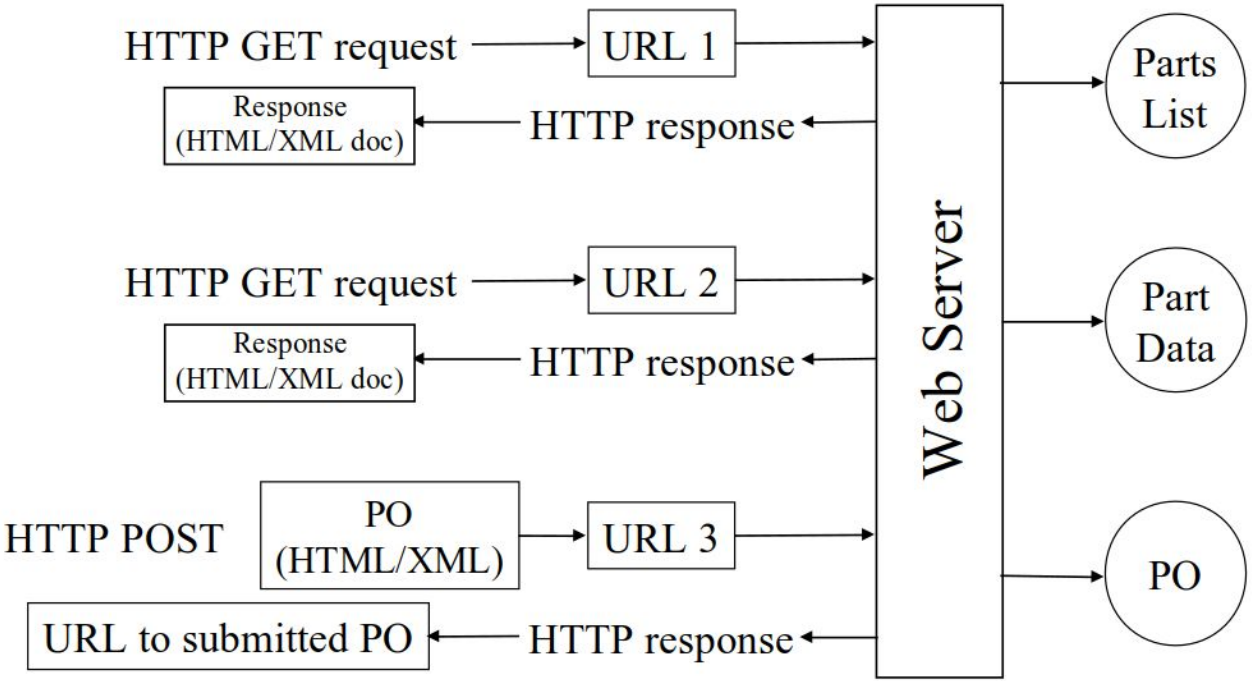
Privacy of Content

- Firewalls, routers, caches base decisions only on HTTP header.
  - **Should never examine the request body.**
- Letter analogy: Postal service doesn't look inside your letter (this is illegal), they just act based on addressing on the outside.
  - The content should not matter, just the metadata.
  - Protects privacy of data.

The Parts Depot Web Store

- Parts Depot, Inc wants to deploy a web service to enable its customers to:
  - Get a list of parts.
  - Get detailed information about a particular part.
  - Submit a Purchase Order (PO).
- How would you architect this?
  - Let’s discuss the RESTful way to design this.

The RESTful Way



Retrieving a List of Parts

Service: Get a list of parts

- Web service offers a URL to parts list **resource**.
- Client posts GET request to URL to get parts list:
  - http://www.parts-depot.com/parts
- **How** the web service generates parts list is transparent to the client.
  - **Enforces loose coupling.**

REST Fundamentals:

1. Create a resource for every service.
2. Identify each resource using a URI.

Data Returned: Parts List

```
<?xml version="1.0"?>
<Parts>
    <Part id="00345" href="http://www.parts-depot.com/parts/00345"/>
    <Part id="00346" href="http://www.parts-depot.com/parts/00346"/>
    <Part id="00347" href="http://www.parts-depot.com/parts/00347"/>
    <Part id="00348" href="http://www.parts-depot.com/parts/00348"/>
</Parts>
```

- Contains links to detailed information about parts.
- Client can get information about a specific part by accessing endpoint identified in a response.

REST Fundamental:

Data that a service returns should link to other data.

- **Design data as network of information**
- **Contrast with OO design, which says to encapsulate information.**

Retrieving Details on a Part

Service: Get detailed information about a particular part

- Web service makes available a URL to each part resource.
- A client can request information on a specific part by posting GET request to http://www.parts-depot.com/parts/00345

Data Returned: Part 00345

```
<?xml version="1.0"?>  <Part>
    <Part-ID>00345</Part-ID>       <Name>Widget-A</Name>
    <Description>This part is used within the frap assembly</Description>
    <Specification href="http://www.parts-depot.com/parts/00345/specification"/>
    <UnitCost currency="USD">0.10</UnitCost>
    <Quantity>10</Quantity>
</Part>
```

- Data is linked to still more data. Part specification may be found by traversing the link.
- Response allows client to get more detailed information.

Let’s take a break!

Designing Services with REST

Designing Services With REST

- Client requests should be **idempotent** – multiple requests should lead to the “same” response.
  - Should be able to retry if transmission fails.
- “Idempotent” refers meaning of information and not necessarily the content.
  - Ex. Endpoint that returns the current time.

PUT and POST

- PUT is idempotent, POST is **not**.
  - Multiple POSTs may create multiple sub-resources.
- PUT requires a full resource ID path.
  - Client creates resource.
- POST does not require full resource ID path.
  - Server notifies client of resource location.
  - Post can still be used for resource updates.

PUT and POST

- PUT http://MyService/Persons/
  - Won't work. PUT requires a complete URI.
- PUT http://MyService/Persons/1
  - Insert a new person, PersonID=1, if it does not already exist or update existing resource with the payload.
- POST http://MyService/Persons/
  - Insert new person (using the payload), generate new ID.
- POST http://MyService/Persons/1
  - Update the existing person where PersonID=1

Handling POSTs

- Other methods are idempotent, but POST creates new resources.
- Multiple POSTs of same data must be harmless.
  - Put message ID in a header or in the message body.
  - This renders multiple posts harmless.
    - Prevents “multiple charge” issue with web stores.

Handling POSTs

- Many ways to do this:
  - Exact: client or server-side unique transaction ID.
  - Heuristic: check and remove “likely duplicates”.
- Wasted IDs are irrelevant.
  - Duplicated POSTs are not acted on by the server
- Server must send back same response original POST got, in case the application is retrying because it lost the response.

Idempotence

- What does this mean, strictly speaking?
  - Call to server must return the same thing each time?
  - No side effects?
- What about changing data?
  - Time-of-day service.
  - Each GET call returns a new time. Is this RESTful?
    - As long as the **resource is constant**.
    - The value does not need to be constant, just how we access it.

Statelessness

- Each request contains all information needed to service the request.
- No client state is held on the server.
  - Benefits in scalability and availability.
  - Performance may be worse (multiple requests may be needed to get information).

Well-Structured URIs

- Avoid using spaces. Use _ (underscore) or – (hyphen) instead.
- Remember that URIs are case insensitive.
- Stay consistent with naming conventions.
- URIs are long lasting.
  - If you change the location of a resource, keep old URI.
  - Use status code 300 and redirect the client.

Well-Structured URIs

- Avoid verbs for resource names unless resource is actually an operation or a process.
  - Bad URIs:
    - http://MyService/FetchPerson/Mike
    - http://MyService/DeletePerson?id=Mike
  - Good URI:
    - http://MyService/Persons/Mike
    - You can apply verbs to this resource.

Food For Thought

What if Parts Depot has a million parts, will there be a million static pages?

http://www.parts-depot/parts/000000

http://www.parts-depot/parts/000001

...

http://www.parts-depot/parts/999999

Food For Thought

- URLs are **logical**.
  - Express what resource is desired, not physical object.
  - Changes to the implementation of the resource will be transparent to clients (loose coupling!).
- All parts stored in a database. Web service will receive URL request, parse it for ID, query the database, and generate the response document at runtime.

Food For Thought

Physical URLs

http://www.parts-depot/parts/000000.html
http://www.parts-depot/parts/000001.html
...
http://www.parts-depot/parts/999999.html

Logical URLs

http://www.parts-depot/parts/000000
http://www.parts-depot/parts/000001
...
http://www.parts-depot/parts/999999

- Physical URLs point to HTML pages.
  - If there are a million parts,we don't want a million HTML pages.
- Changes to how these parts data is represented will effect all clients that were using the old representation.

Activity - Let's Make a Deal

- Contestants are presented with three doors.
  - One leads to a great prize.
  - The others lead to nothing.
  - Users select one door.
  - Host opens one of the other doors.
  - Users can then choose to open their door or the remaining unopened door.

Activity - Let's Make a Deal

You must support:

- Creation of games.
- User selection of a door.
  - The game will open one of the other doors.
- User opening of a door.
- Querying of the current state of the game and outcome (if complete) by user.
- Deletion of a game.

Activity - Let's Make a Deal

Resource	Verb
/games	get – status of games server post – create new game
/games/{gid}	get – status of game (in_play, won, lost) delete – delete the game resource
/games/{gid}/doors	get – status of all doors
/games/{gid}/doors/{1..3}	get – door status {closed, selected, opened} put – update door status

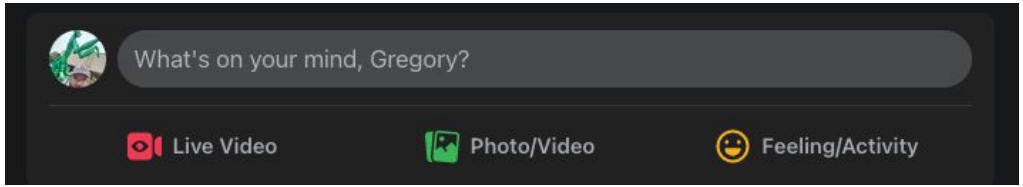
- Use status codes to determine whether an operation is reasonable.
- Once game is finished (won/lost), only GET requests are allowed.

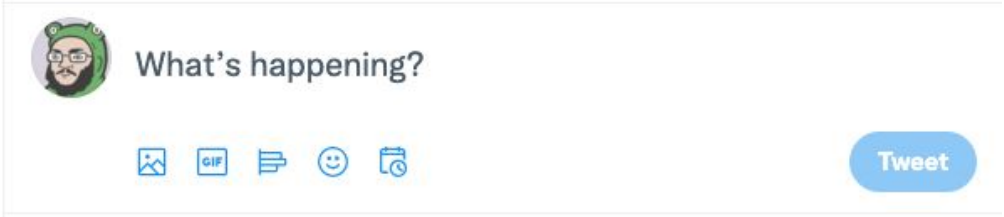
Designing Reusable APIs

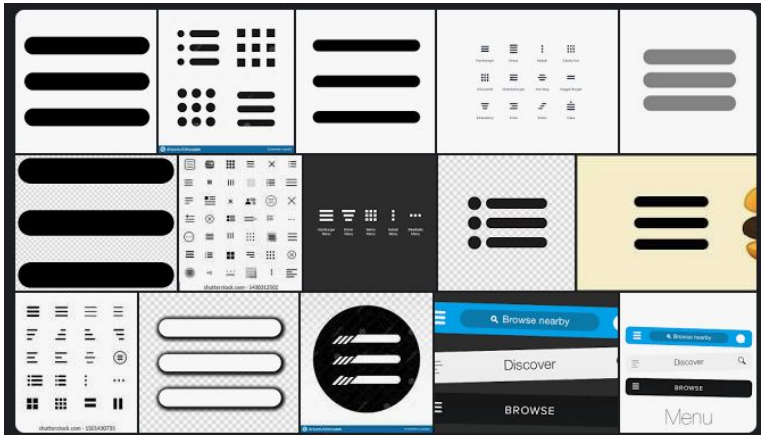
Adapted from Verborgh, R. and Dumontier, M. (2018), “A Web API ecosystem through feature-based reuse”, Internet Computing, IEEE, Vol. 22 No. 3, pp. 29–37.

Human-based Interaction

- Well-designed websites base user interaction on common interaction patterns.
- Interaction patterns are *reused* across the web.



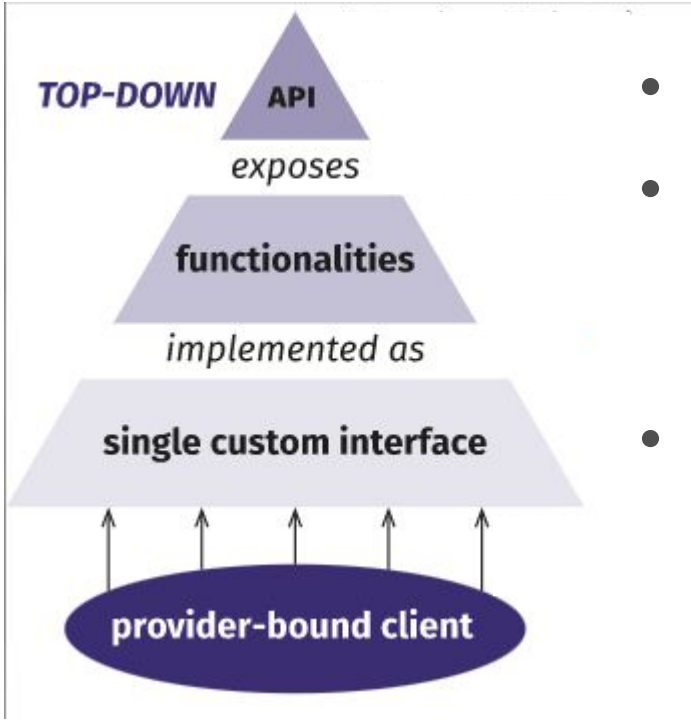




Designing Reusable APIs

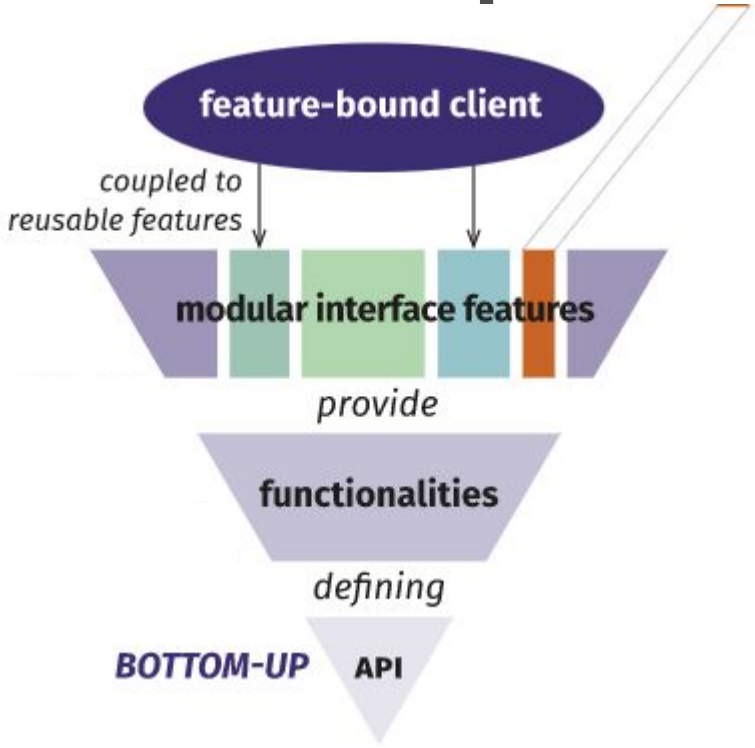
- Individual APIs *are* reusable.
  - We can use one to post a photo to Facebook.
- *However, APIs are often not substitutable.*
  - If we want to post photos to Facebook or Twitter, the APIs can't be swapped.
- API design should center around common interaction patterns.
  - Like human-based interaction.

Top-Down API Design



- API is monolithic.
- Clients couple to specific interface to interact with lower-level parts.
- Only clear invocation mechanisms are parameter names and types.

Bottom-Up API Design



- A feature offers interface to a common function type.
  - Search text, upload file, update status, etc.
  - Should be simple, self-describing.
- Clients couple to select feature APIs, not full system API.
- System APIs reuse features.
  - Whole API may not be identical.

1: Web APIs Consist of Features with Common Interfaces

- A web service should be split into features with their own interfaces.
  - Accessing/updating/sorting list of items, pagination, updating a status, uploading a photo, search.
- Features can be optionally selected by client or enabled/disabled by server.
  - Clients only affected by changes to selected features.
  - Clients can make use of only what they need.

2: Partition Interface for Feature Reuse

- If a feature is available elsewhere, reuse it in your API instead of implementing it yourself.
  - Clients could perform task with any API offering feature.
- If designing a new feature, make it available separately for reuse.
  - Feature-specific repository, documentation.
- Prioritize reuse when possible, make new functionality available as features.

3: API Responses Should Advertise Relevant Features

- Server should include or link to supported features.
  - Support indicated in header of HTTP response or inside response body.
  - Can indicate which optional parts are implemented.
- Clients can determine whether API offers needed features at runtime.

3: API Responses Should Advertise Relevant Features

- Hypermedia: Embed links to resources within JSON body of HTTP response.
- Ex: GET call to entry point returns links to accessible resources.
- GitHub API uses hypermedia to broadcast functionality.

GET /

```
{
    "version": "1.2.3",
    "description": "Example API to
manage orders",
    "links": [
        { "rel": "orders",
          "href": "/orders" },
        { "rel": "customers",
           "href": "/customers"},
        { "rel": "customer-by-id",
           "href": "/customer/{id}"},
        { "rel": "customer-by-email",
          "href": "/customer{?email}"},
        ...
    ]
}
```

4: Features Describe their Functionality and Invocation

- When queried, a feature should describe its functionality and how it is accessed in a standard form (e.g., hypermedia, JSON schema).
  - Reduces need to find documentation on an API.
  - APIs implementing a feature do not need to use same URL structure/parameter names.
- Client can query feature for details at runtime.

We Have Learned

- REST is a web-based API format.
  - Services provide resources (through URLs) designed to be consumed by programs rather than by people.
  - Design Principles:
    - Stateless
    - Resource-Based (URI)
    - Uniform Interface (GET, PUT, POST, DELETE)
    - Links describe relationships
    - Cacheable and monitorable using standard internet tools

We Have Learned

- APIs should be designed to be reusable.
  - APIs should be split into features.
  - Features should have a common interface with compatible features with separate implementations.
  - The overall API should be partitioned into these separate features with their own interfaces.
  - APIs should advertise available features.
  - Features should broadcast their functionality and invocation details.

Next Time

- Testing of Complex Systems

- Assignment 3 - due December 4

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY