SE 464
Week 2

Requirements, Protocols, and Common Architecture Patterns

Functional and Non-Functional Properties

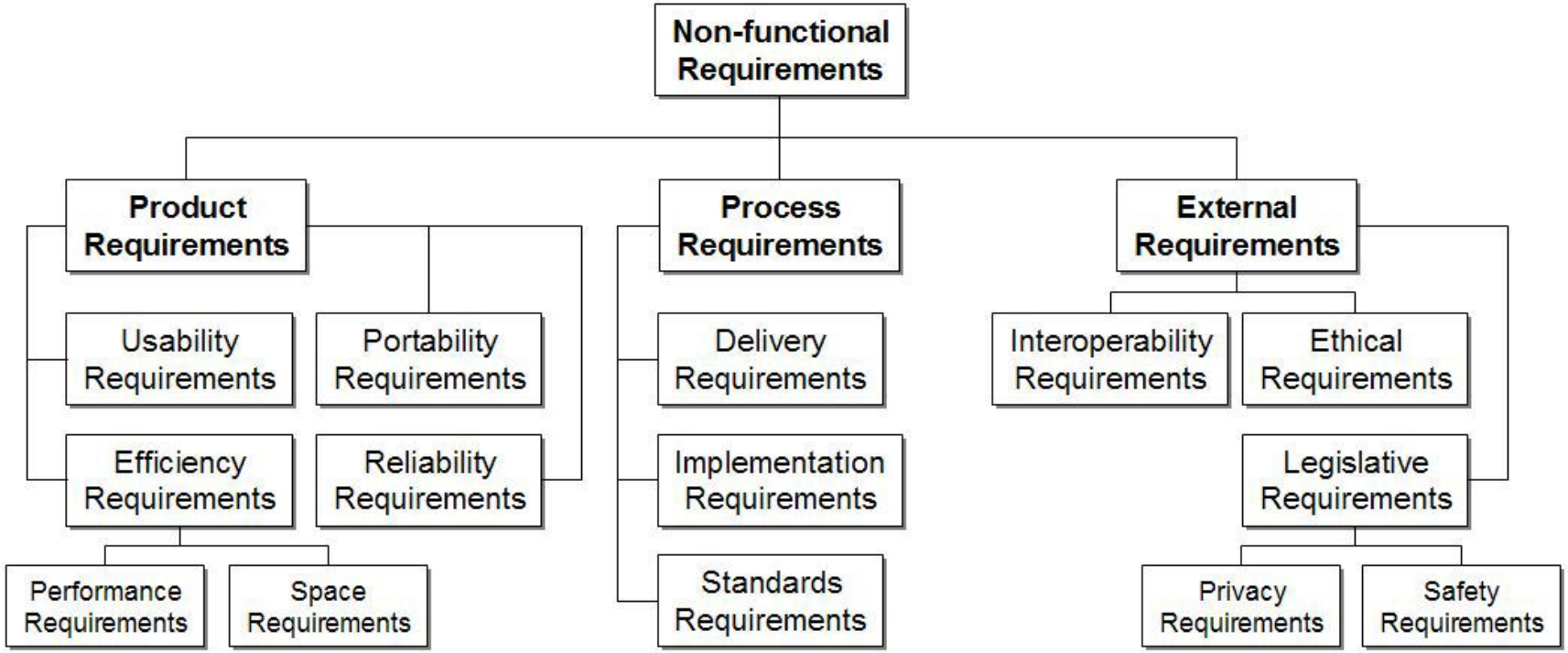
Functional vs Non-F Requirements

“Functional requirements define specific behavior.
Non-functional requirements specify criteria that can be used to judge the operation of a system.
Broadly, functional requirements define what a system is supposed to do and non-functional requirements define how a system is supposed to be.”

Functional vs Non-F Requirements

Functional Properties:
What the system is supposed to do ('the system shall do X').
Non-Functional Properties:
What the system is supposed to be ('the system shall be Y').

Classifications of NFPs



NFPs Explained

A software system's non-functional property (NFP) is a constraint on the manner in which the system implements and delivers its functionality .

Example NFPs:

Efficiency Complexity

Scalability Heterogeneity

Evolvability Dependability

Designing for FPs and NFPs

Engineering products sold based on FPs
Providing the desired functionality is often quite challenging
However, the system's success will ultimately rest on its NFPs
"This system is too slow!"
"It keeps crashing!"
"It has so many security holes!"
"Every time I change this feature I have to reboot!"
"I can't get it to work with my home theater!"

NFP: Efficiency

Efficiency is a quality that reflects a system's ability to meet its performance requirements.

Components:

Keep them "small".

Simple and compact interfaces.

Allow multiple interfaces to the same functionality.

Separate data from processing components.

Separate data from meta data.

NFP: Efficiency

Connectors:

Carefully select connectors.

Be careful of broadcast connectors.

Encourage asynchronous interaction.

Be wary of location/distribution transparency.

Topology:

Keep frequent collaborators “close”.

Consider the efficiency impact of selected styles.

NFP: Complexity

Complexity is a property that is proportional to the size of a system, its volume of constituent elements, their internal structure, and their interdependencies.

Components:

Separate concerns.

Isolate functionality from interaction.

Ensure cohesiveness.

Insulate processing from data format changes.

NFP: Complexity

Connectors:
Isolate interaction from functionality.
Restrict interactions provided by each connector.
Topology:
Eliminate unnecessary dependencies.
Use hierarchical (de)composition.

NFP: Scalability / Heterogeneity

Scalability: The capability of a system to be adapted to meet new size / scope requirements.

Heterogeneity: A system's ability to be composed of, or execute within, disparate parts.

Portability: The ability of a system to execute on multiple platforms while retaining their functional and non-functional properties.

NFP: Scalability / Heterogeneity

Components:

Keep components focused

Simplify interfaces

Avoid unnecessary heterogeneity

Distribute data sources

Replicate data

NFP: Scalability / Heterogeneity

Connectors:

Use explicit connectors

Choose the simplest connectors

Direct vs. indirect connectors

Topology:

Avoid bottlenecks

Place data close to consumer

Location transparency

NFP: Evolvability

Evolvability: The ability to change to satisfy new requirements and environments.

Components:
Same as for complexity.

Goal is to reduce risks by isolating modifications.

NFP: Evolvability

Connectors:
Clearly define responsibilities.

Make connectors flexible.

Topology:
Avoid implicit connectors.

Encourage location transparency.

NFP: Dependability

Reliability: The probability a system will perform within its design limits without failure over time.

Availability: The probability the system is available at a particular instant in time.

Robustness: The ability of a system to respond adequately to unanticipated runtime conditions.

NFP: Dependability

Fault-tolerance: The ability of a system to respond gracefully to failures at runtime.

Faults arise from: environment,

components, connectors, component-connector mismatches.

Survivability: The ability to resist, recover, and adapt to threats.
Sources: attacks, failures, and accidents.

Steps: resist, recognize, recover, adapt.

Safety: The ability to avoid failures that will cause loss of life, injury, or loss to property.

NFP: Dependability

Components:

Control external component dependencies.

Support reflection.

Support exception handling.

Connectors:

Use explicit connectors.

Provide interaction guarantees.

NFP: Dependability

Topology:
Avoid single points of failure.
Enable back-ups.
Support system health monitoring.
Support dynamic adaptation.

Communication Protocols

The following slides were taken with permission from Professor Steve Tarzia of Northwestern University.
https://stevetarzia.com/teaching/310/slides/Scalability%20Lecture%2005%20-%20REST%20APIs%20and%

Application Programming Interfaces (APIs)

An API defines how software can be used by other software.

- The API for a code **library** is the list of functions/classes it provides.
- Software **services** provide network remote procedure call (RPC) APIs.
  - **Network-level APIs** can have any format, but most commonly:
    - REST
    - SOAP *(old)*

    } *built on top of HTTP*
    - Thrift
    - Protocol buffers

    } binary protocols, more efficient than REST.
    - GraphQL
  - Usually includes some form of *authentication:*
    - Service must identify you to give access or personalized data.

HTTP methods and responses

Methods

- **GET**: to request a data
- **POST**: to post data to the server, and perhaps get data back, too.

Less commonly:

- **PUT**: to create a new document on the server.
- **DELETE**: to delete a document.
- **HEAD**: like GET, but just return headers

Response codes

- **200 OK**: success
- **301 Moved Permanently**: redirects to another URL
- **403 Forbidden**: lack permission
- **404 Not Found**: URL is bad
- **500 Internal Server Error**

… and many more

A weather information service (REST API)

HTTP Request

```
GET
http://api.wthr.com/[key]/fore
cast?location=San+Francisco
HTTP/1.1
Accept-Encoding: gzip
Cache-Control: no-cache
Connection: keep-alive
```

HTTP Response

```
HTTP/1.1 200 OK
Content-Length: 2102
Content-Type:
application/json

{ "wind_dir": "NNW",
  "wind_degrees": 346,
  "wind_mph": 22.0,
  "feelslike_f": "66.3",
  "feelslike_c": "19.1",
  "visibility_mi": "10.0",
  "UV": "5", … }
```

Idempotence

- An **idempotent** request can be repeated without changing the result.
- HTTP expects every method **except POST** to be idempotent.
- HTTP proxies/servers may repeat your PUT or DELETE requests, and your REST API implementations should be OK with this.
- For example, creating an Elasticsearch document:
  - **PUT** /my-index/_doc/2345
    {"title": "My Great Article", "txt": "Hi everyone. I'm here to write about…"}
  - **POST** /my-index/_doc
    {"title": "My Great Article", "txt": "Hi everyone. I'm here to write about…"}
- The PUT variation can be repeated and it will just overwrite the doc.
- The POST variation would create duplicate docs if repeated.

REST API semantics must work with HTTP's rules

- Let's say we're developing a a social media application.
- What's wrong with this API definition for deleting my latest post?
  - DELETE /user/[user-id]/feed/posts/latest

- Http DELETE should be *idempotent*.
- However, repeating the request above changes the system state.
- From the services' perspective, repetition of one deletion looks the same as if the user had purposely deleted multiple latest posts.
- What's the solution? Make each deletion look different:
  - DELETE /user/[user-id]/feed/post/[post-id]

REST API example

Twitter REST API documentation

- https://developer.twitter.com/en/docs/tweets/post-and-engage/api-reference/post-statuses-update

Elastic Search: https://www.elastic.co/guide/en/elasticsearch/reference/current/rest-apis.html

Discourse web forum public API documentation:

- https://docs.discourse.org

Output examples, viewable in a web browser:

- https://meta.discourse.org/categories.json
- https://meta.discourse.org/latest.json?category=7
- https://meta.discourse.org/t/3423.json (requires authentication)
- http://ssa-hw2-backend.stevetarzia.com/api/search?query=northwestern&date=2020-04-16

RESTful API design style

- Paths represent "resources" – data or objects in your system.
- GET reads data
- PUT/POST creates or modifies data
- DELETE deletes data

HTTP method should be the only verb

- Representing arbitrary **actions** in REST can be tricky. Usually we can convert an action into an event resource.

Verbs in path are not RESTFUL!

- This is acceptable, but not RESTful: POST /inbox/createMessage
- Here is a RESTful alternative: POST /inbox/message

Resource/noun

- And finally, an even better design: PUT /inbox/message/[uuid]

Remote Procedure Call (RPC)

goal: easy-to-program network communication

hides most details of client/server communication

client call is much like ordinary procedure call

server handlers are much like ordinary procedures

RPC is widely used!

Remote Procedure Call (RPC)

RPC ideally makes net communication look just like fn call:

Client:

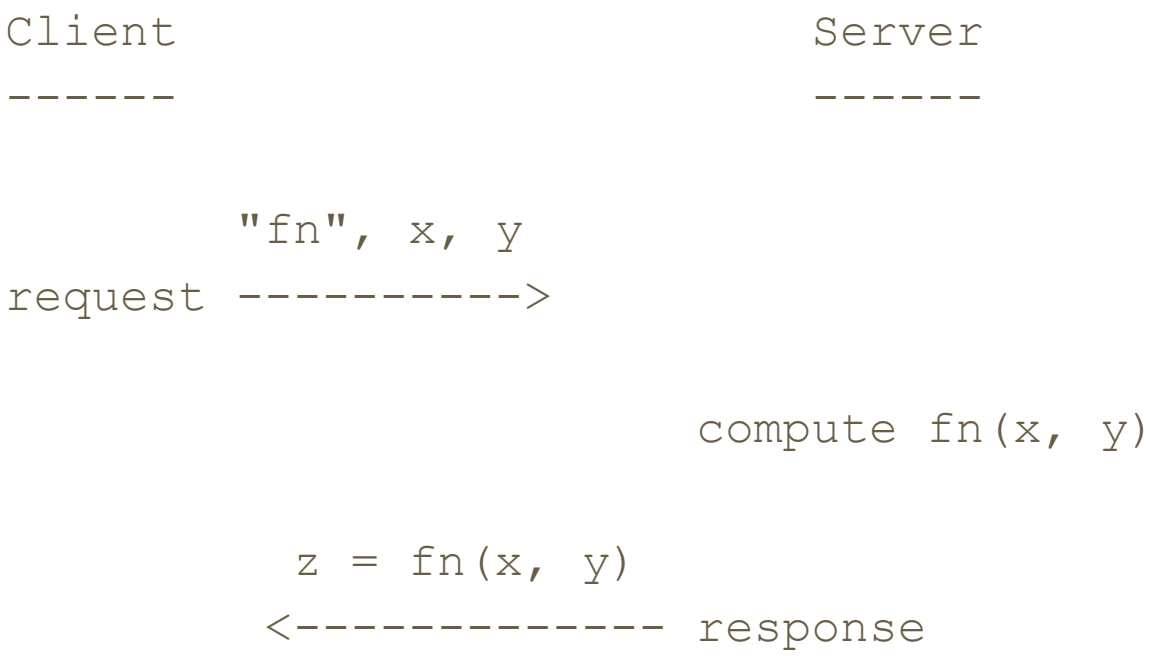
```
z = fn(x, y)
```

Server:

```
fn(x, y) {
  compute
  return z
}
```

RPC aims for this level of transparency

RPC Message Diagram



Pros and Cons of RPC

Advantages:
Simplifies coding by abstracting network communication.
Language neutrality (can be implemented in different languages).
Encapsulates network communications.

Disadvantages:
Error handling can be complex.
Network dependence can create vulnerabilities.
Debugging remote procedures can be more challenging.

gRPC

Definition: gRPC is a modern, high-performance, open-source framework developed by Google for making remote procedure calls (RPC).

Protobufs: Utilizes Protocol Buers (protobufs) for ecient serialization.

Advantages of gRPC

Performance: Optimized for low latency, high efficiency, and scalability, making it suitable for performance-sensitive applications.

Streaming: Supports bidirectional streaming, allowing both the client and server to read and write at the same time.

Deadlines/Timeouts: Clients can specify how long they are willing to wait for an RPC to complete; the server can check and comply with this.

Pluggable: Allows custom authentication, load balancing, and more.
Use Cases: Ideal for microservices architecture, real-time updates, multiplexed connections, etc.

Common Architectures - Microservices vs Monolith

The following slides were taken with permission from Professor Steve Tarzia of Northwestern University.
https://stevetarzia.com/teaching/310/slides/Scalability%20Lecture%2006%20-%20Microservices.pdf

CS-310 Scalable Software Architectures

Lecture 6: Microservices

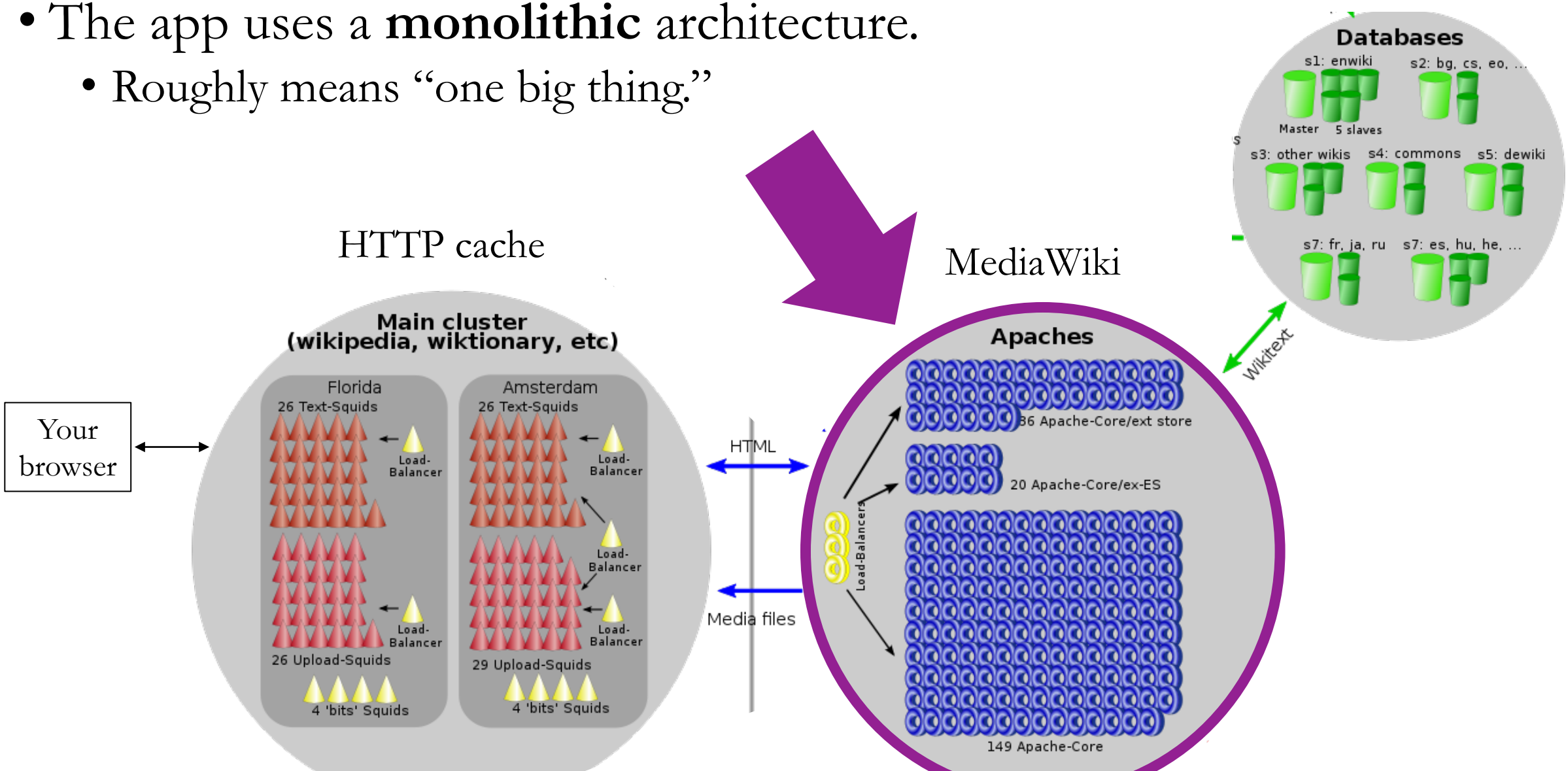
Steve Tarzia

Last time: REST APIs and Data Serialization

- Services are **black boxes**, exposing **network APIs**.
  - Decouples development of different parts of the system.
  - Network APIs define the format and meaning of requests and responses.
- **REST** is the most popular format for network APIs
  - Based on **HTTP** and uses *url*, *method*, *response codes*, usually *JSON bodies*.
- **JSON** is a common data **serialization** format. **XML** is also used.

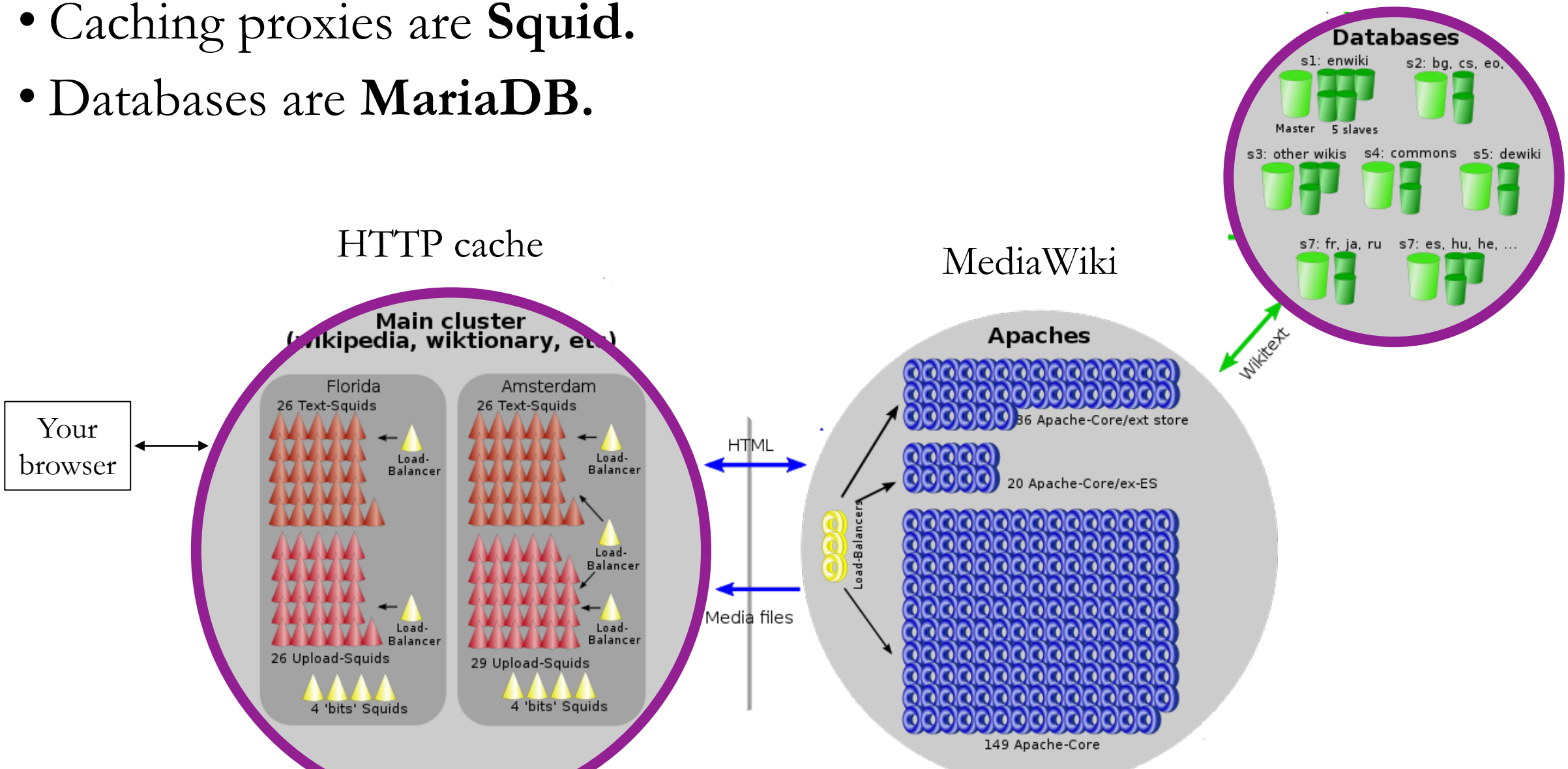
Wikipedia is mostly one big PHP app

- The app uses a **monolithic** architecture.
  - Roughly means "one big thing."



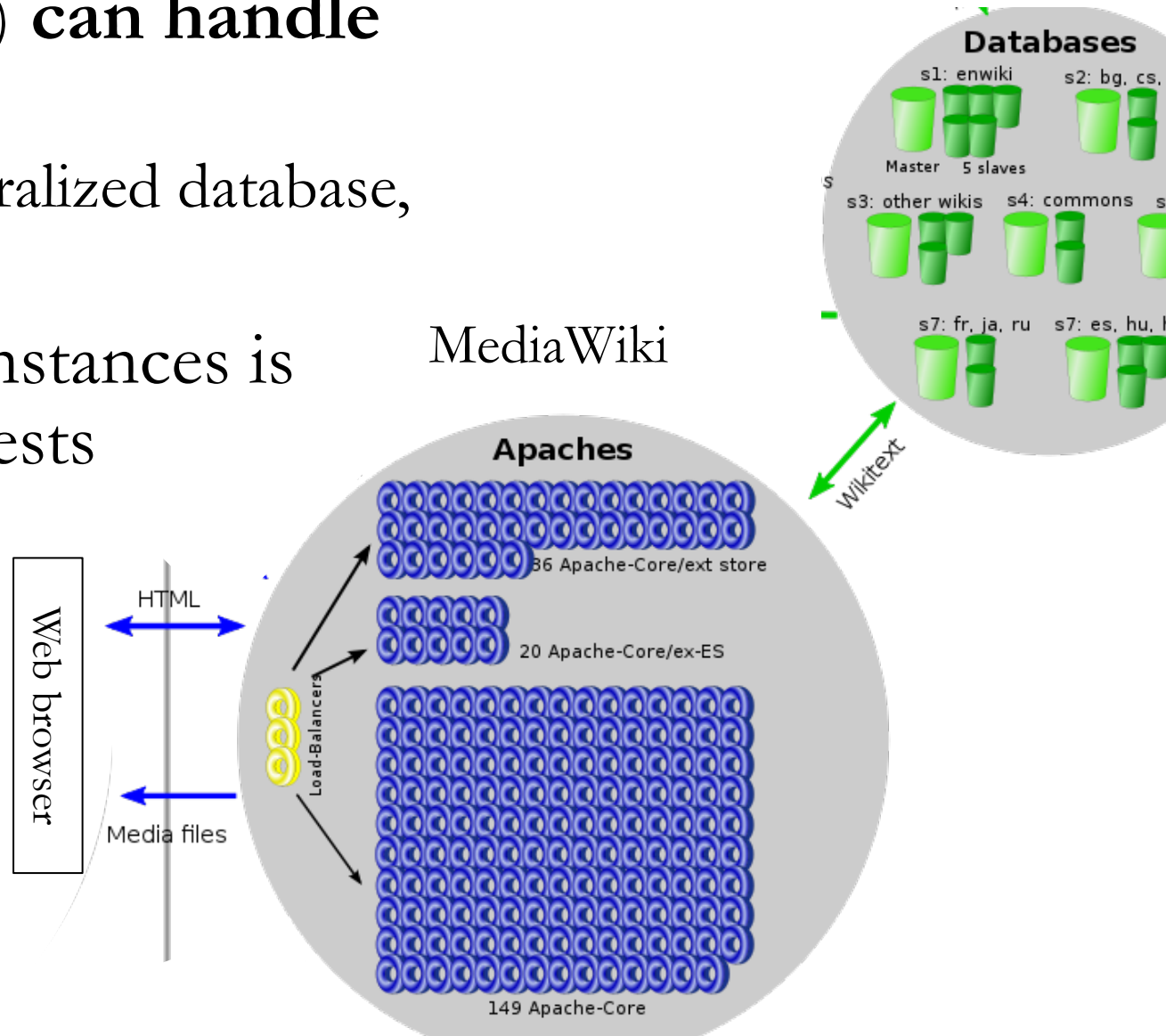
Cache and Database are "off-the-shelf" software

- Caching proxies are **Squid.**
- Databases are **MariaDB.**



Monolithic apps

- Each of instance of Mediawiki (⊙) **can handle any request on its own**.
  - *Caveat:* it needs the help of the centralized database, but let's ignore the DB for now.
- The only reason we have 200 of instances is to handle many independent requests in parallel.
  - They're interchangeable *clones*.
- When a developer is testing new code, they can just run one instance locally.



Advantages of a Monolithic design

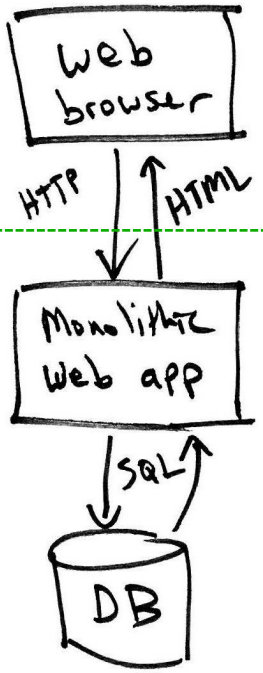
- Easy to build, deploy, test, coordinate, share data.
  - Certainly, the best choice for simple apps.
  - Even some very large services (eg., Facebook) use a monolithic design.

Disadvantages

- Creates bottlenecks in SW development processes.
  - Lots of developers working on one codebase – lots of coordination/merging.
- One huge codebase can lead to messy, fragile code.
  - A change *here* can cause unexpected bugs *there*.
- The whole app must be redeployed for small new functionality.
- Must choose *one* programming language, build system, runtime env.

Breaking up a monolithic architecture, example

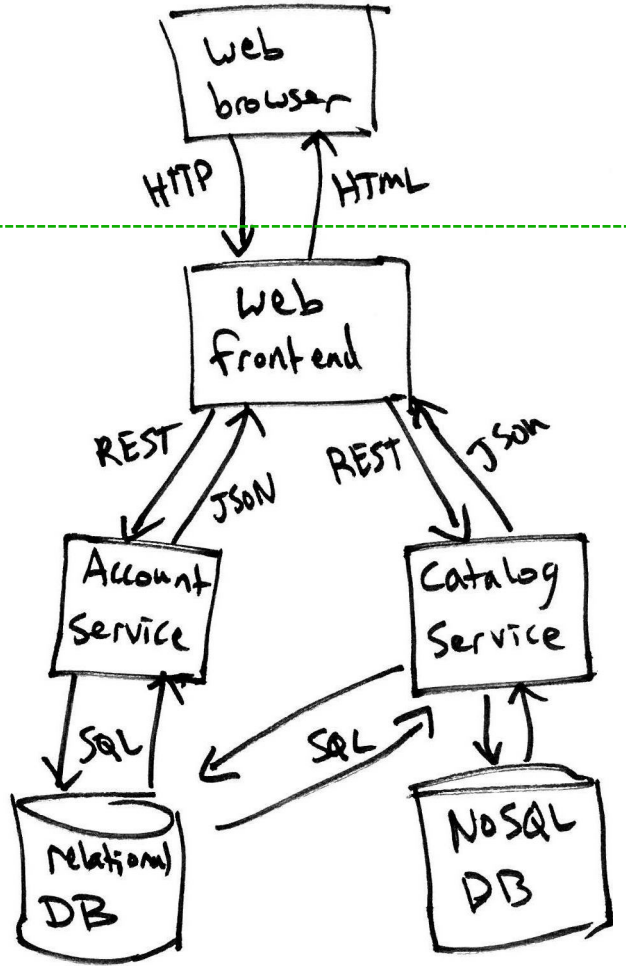
One big service



User's machine

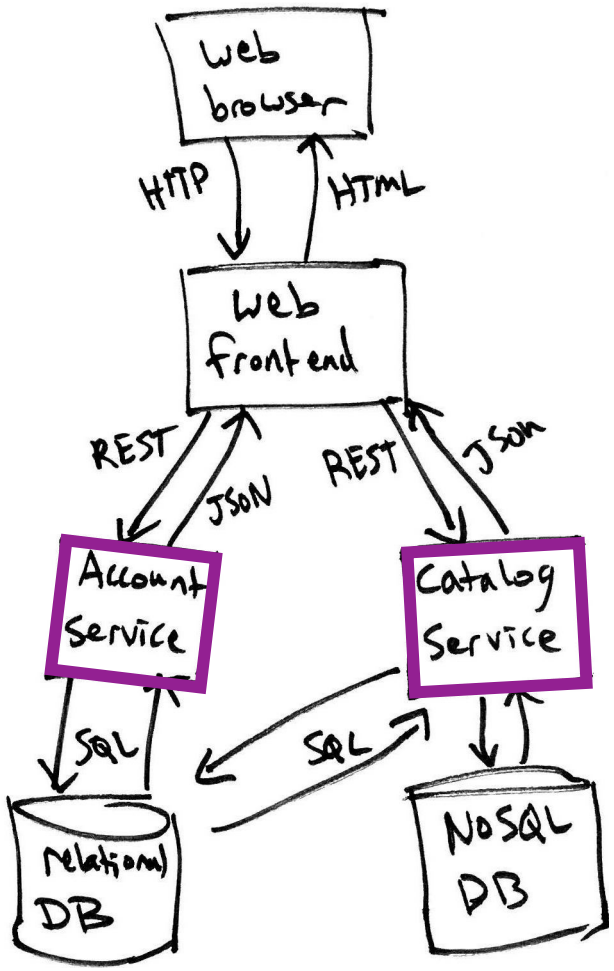
The cloud

Several services



Microservices

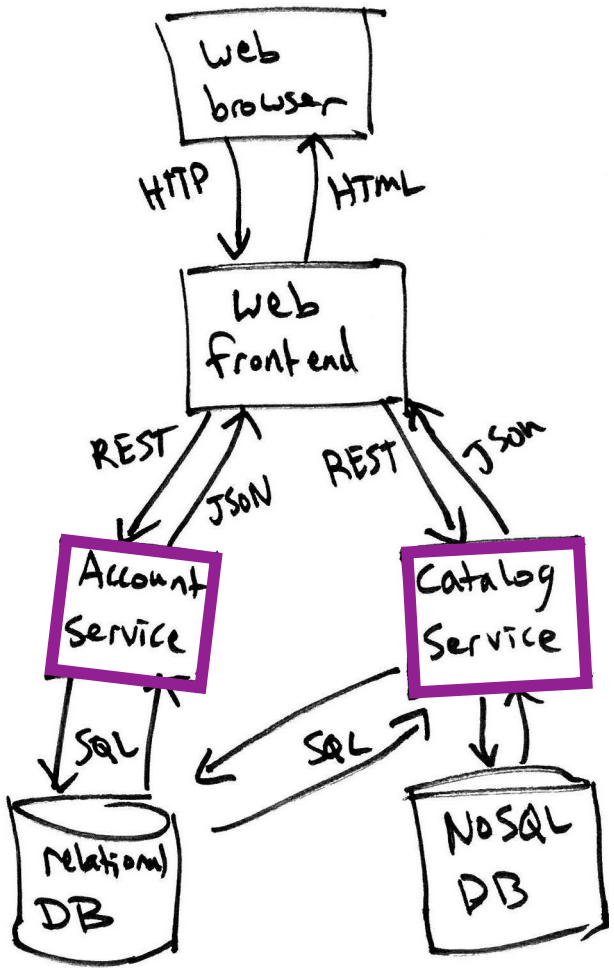
- In a **Service Oriented Architecture** many smaller services work together. Recently, this idea has been re-branded as **Microservices**.
- Responsibilities are split among specialized services.



- To the outside world it still looks like one app, but internally there are many different apps working together.
- Notice that microservices may interact with other services, databases, etc. to do their job.

Microservice interactions

- Each microservice is a **black box** to the rest of the system. It's an independent service.
- Microservice handles requests from the network.
- Implementation (and programming language) details are hidden from the rest of the system.



- However, a clear and language-independent network-level API is needed to specify the format of requests and responses.
  - From last lecture, could be:
    REST, Thrift, ProtoBuf, GraphQL, etc.

A few microservice disadvantages

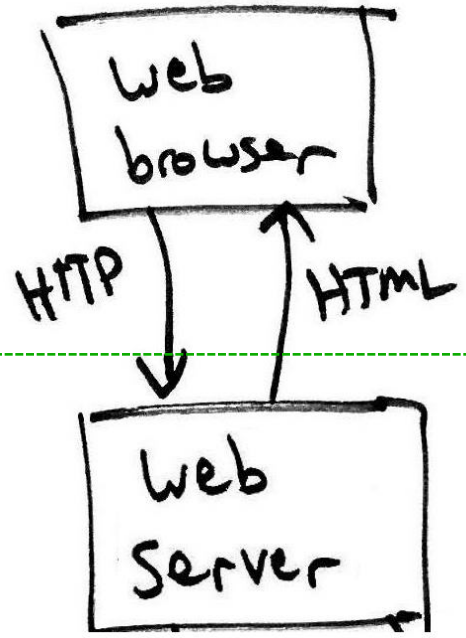
- More difficult to trace through request handling code for debugging, because request handling spans many apps.
  - On the other hand, it's easier to write tests for each smaller service.
- (ACID) **Transactions** are more difficult to support.
  - A monolithic design can manage a single DB connection with a transaction that can be rolled-back.
- Developers get silo-ed/isolated in sub-projects.
  - Collaboration and innovation can be blocked.

Developing a service with a team

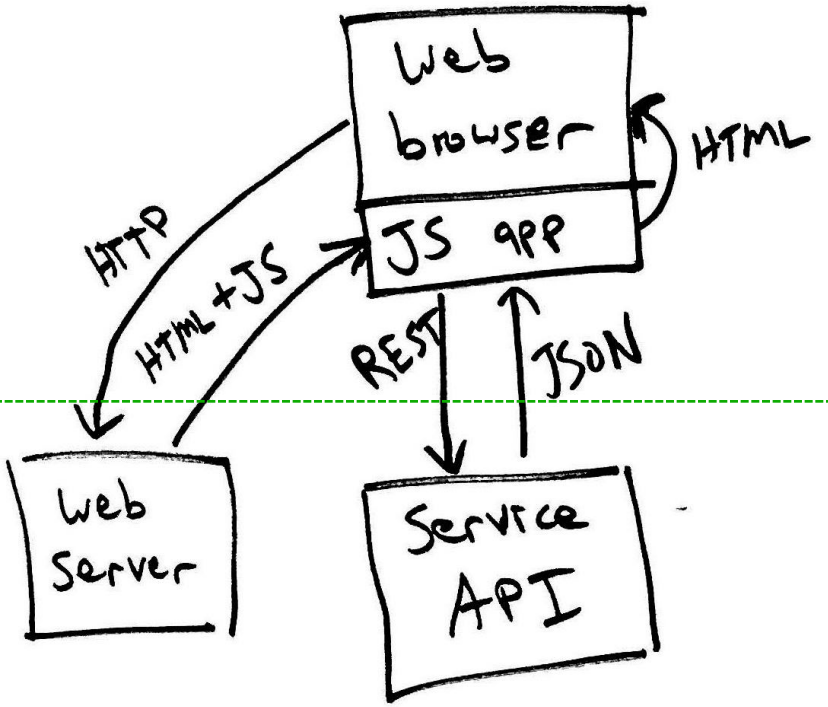
- Microservices isolate codebases with clear network API boundaries, allowing work to proceed in parallel.
- When starting a new project, a typical design process will:
  - Organize the system into several services and databases.
  - Agree on the network-level API for each service.
  - Assign engineers to each service, and build them independently!
- If your application interacts with a service that is under construction, then you can build a quick **mock** of the service.
  - **Mock** services obey the network-level API, but return hard-coded data for testing purposes.

Traditional web app *vs.* JavaScript Single-Page app

- Server generates HTML dynamically.
- Browser doesn't run much JS.
  - Eg., Wikipedia.



- JS app runs in the browser, makes REST requests and generates HTML.
  - Eg., Facebook & other React apps.



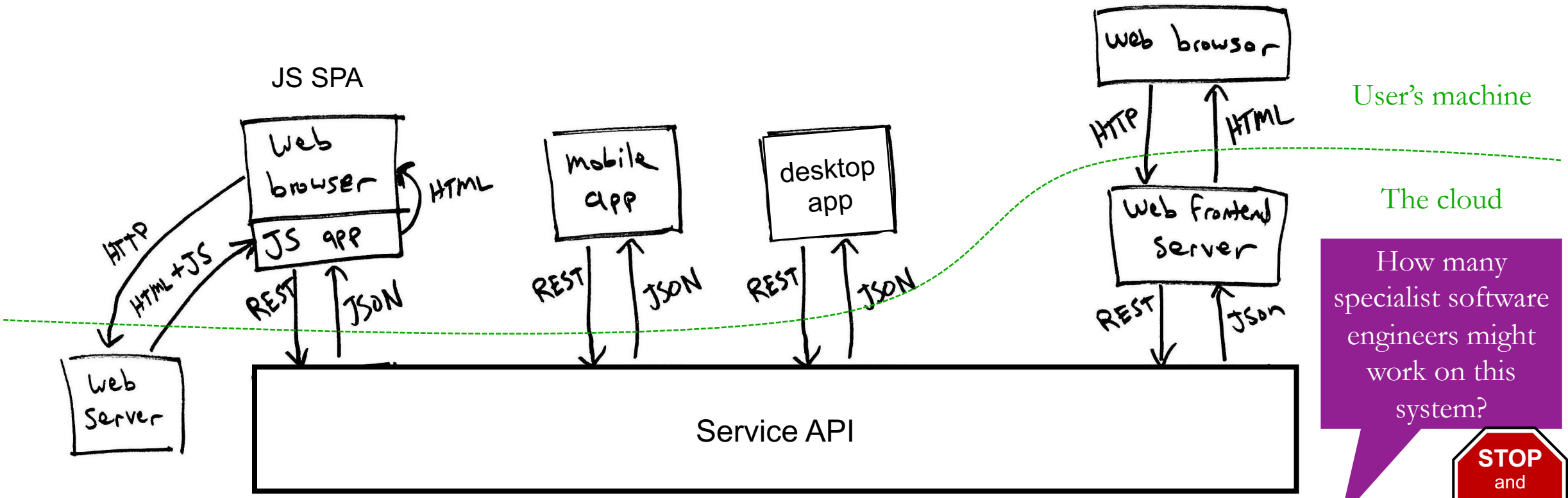
User's machine

The cloud

*Note: many apps combine these two styles.

Cross-platform architecture

- A service API (eg., REST or GraphQL) is essential for supporting mobile and desktop apps with cloud-based data and services.
- A single API can handle all client types.



Review

- Introduced **microservices** as an alternative to **monolithic** design.
- Services are **black boxes**, exposing **network APIs**.
  - Decouples development of different parts of the system.
  - Network APIs define the format and meaning of requests and responses.
- JS **Single-page Applications** (SPAs) interact directly with services.
  - Moves UI concerns away from backend code.
- In a **cross-platform system design**, the same backend service/API can serve mobile, web, and desktop apps.