(CS360) Fundamentals of Software Engineering

Software architectures, HTTP and REST

Lecture 3

Dr. Qobiljon Toshnazarov

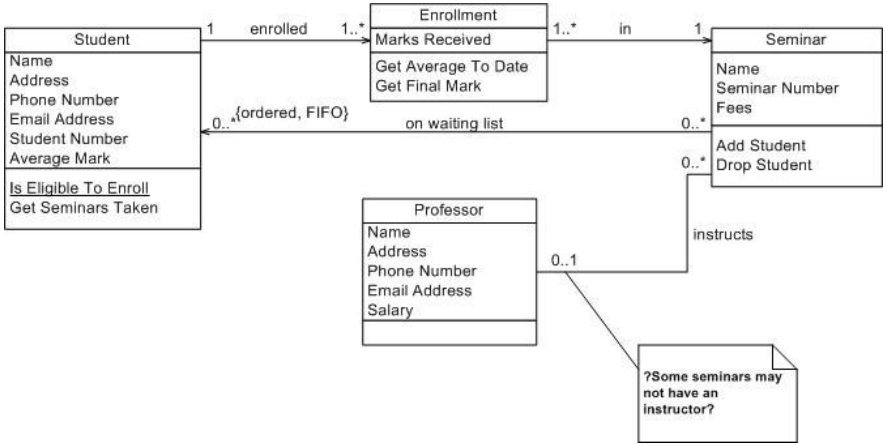
Outline

New Uzbekistan University

- Software architectures
  - Monolithic, layered, pipeline, plugin/microkernel, event-driven, and microservice architectures
- HTTP basics
  - HTTP message components – the client-server communication
  - “HTTP is stateless”?
  - “HTTP is not sessionless?”
- REST protocols
  - Basic REST principles – RESTful interfaces

Design we saw so far in this class

New Uzbekistan University

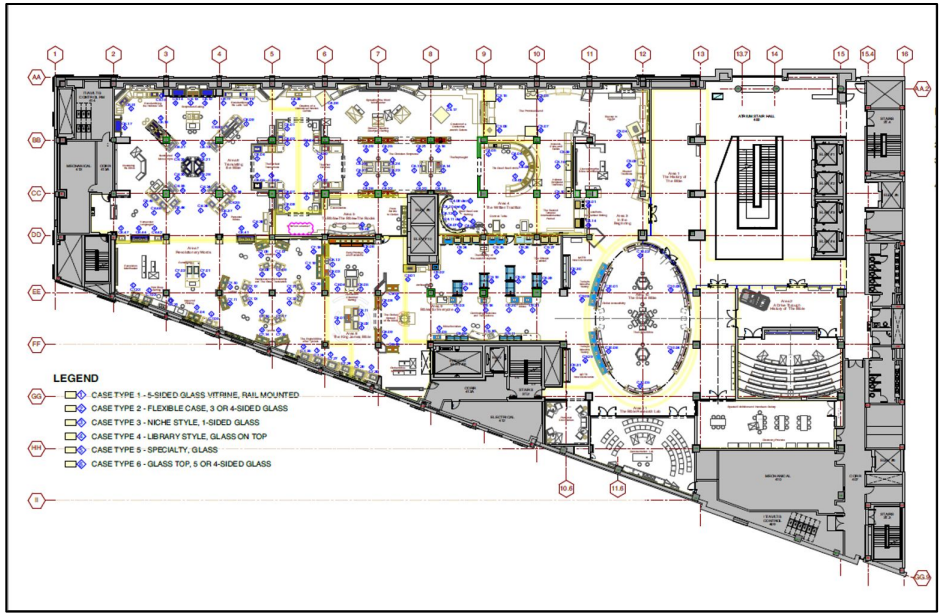


Metaphor: architecture of a building

Design at larger scales

New Uzbekistan University

- Metaphor: architecture of a building
- How do the pieces fit together in larger designs? What do we reuse?



Design at larger scales

New Uzbekistan University

- Metaphor: architecture of a building
- How do the pieces fit together in larger designs? What do we reuse?
- How do we organize into teams?

Design at larger scales

New Uzbekistan University

- How well will our system work in its context?

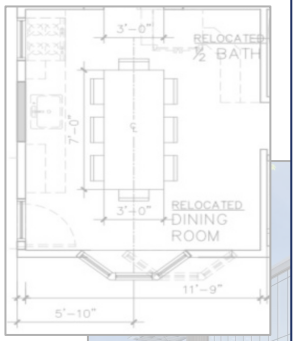
https://ozguvenmimarlik.com/en/architectural-project/urban-design-implementation/tashkent-new-uzbekistan-university

Design at larger scales

New Uzbekistan University

- How well will our system work in its context?

So, when we say “software architecture” we are talking about high-level design of the software with all its parts included.



https://ozguvenmimarlik.com/en/architectural-project/urban-design-implementation/tashkent-new-uzbekistan-university

Goal: create a high-level model of the system

New Uzbekistan University

- Abstract details away into reusable components
  - Abstraction can be critically necessary, especially for medium-to-large size SW
- Allows for analysis of high-level design before implementation
- Enables exploration of design alternatives
- Reduce risks associated with building the software

Properties of “Software Architectures”

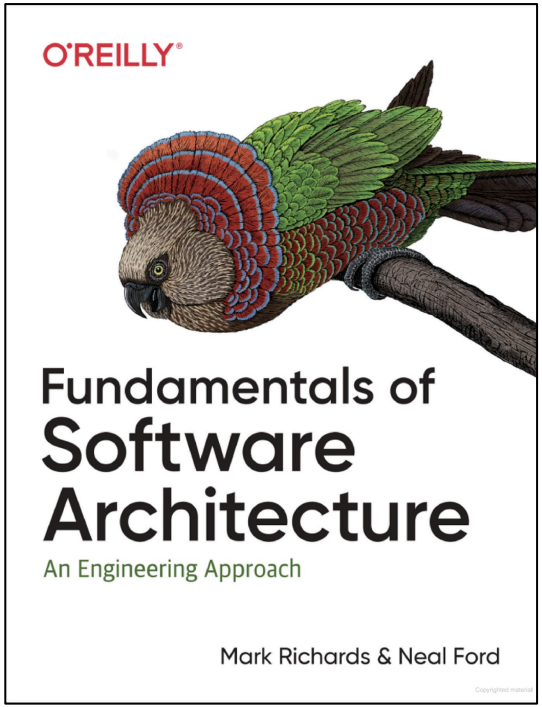
New Uzbekistan University

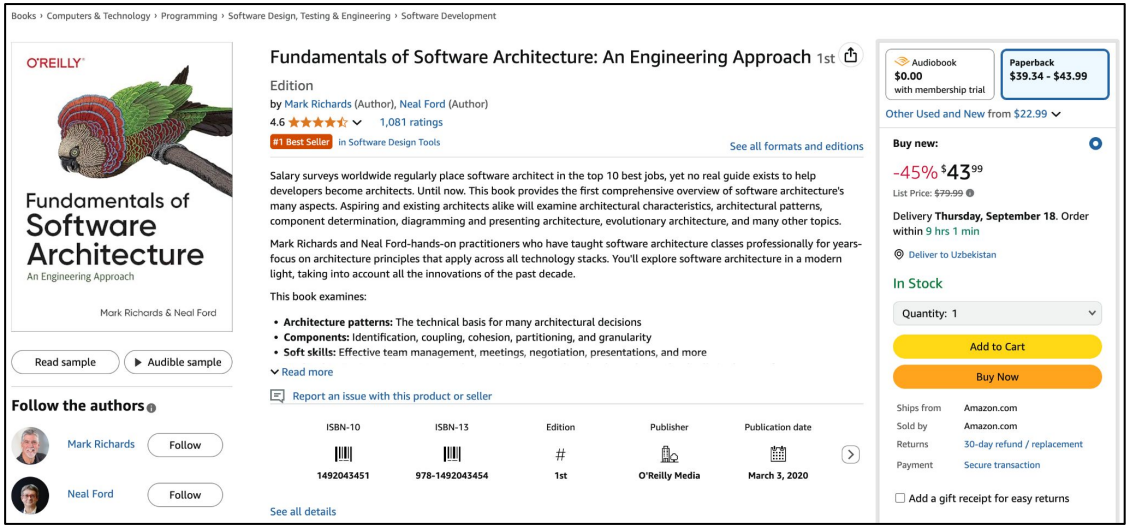
Table 4-1. Common operational SW architecture characteristics	
Term	**Definition**
Availability	How long the system will need to be available (if 24/7, steps need to be in place to allow the system to be up and running quickly in case of any failure).
Continuity	Disaster recovery capability.
Performance	Includes stress testing, peak analysis, analysis of the frequency of functions used, capacity required, and response times. Performance acceptance sometimes requires an exercise of its own, taking months to complete.
Recoverability	Business continuity requirements (e.g., in case of a disaster, how quickly is the system required to be on-line again?). This will affect the backup strategy and requirements for duplicated hardware.
Reliability/safety	Assess if the system needs to be fail-safe, or if it is mission critical in a way that affects lives. If it fails, will it cost the company large sums of money?
Robustness	Ability to handle error and boundary conditions while running if the internet connection goes down or if there’s a power outage or hardware failure.
Scalability	Ability for the system to perform and operate as the number of users or requests increases.

Table from book “Richards & Ford: Fundamentals of Software Architecture”

Book: Fundamentals of Software Architecture

New Uzbekistan University





- Purchase the ebook on Amazon: [HTML]
- Or download via this link: [PDF]

More properties (contd.)

New Uzbekistan University

Table 4-2. Structural SW architecture characteristics	
Term	**Definition**
Configurability	Ability for the end users to easily change aspects of the software's configuration (through usable interfaces).
Extensibility	How important it is to plug new pieces of functionality in.
Installability	Ease of system installation on all necessary platforms.
Leverageability / reuse	Ability to leverage common components across multiple products.
Localization	Support for multiple languages on entry/query screens in data fields; on reports, multibyte character requirements and units of measure or currencies.
Maintainability	How easy it is to apply changes and enhance the system?
Portability	Does the system need to run on more than one platform? (For example, does the frontend need to run against Oracle as well as SAP DB?
Supportability	What level of technical support is needed by the application? What level of logging and other facilities are required to debug errors in the system?
Upgradeability	Ability to easily/quickly upgrade from a previous version of this application/solution to a newer version on servers and clients.

Table from book "Richards & Ford: Fundamentals of Software Architecture"

Even more properties (contd.)

New Uzbekistan University

Table 4-3. Cross-cutting SW architecture characteristics	
Term	**Definition**
Accessibility	Access to all your users, including those with disabilities like colorblindness or hearing loss.
Archivability	Will the data need to be archived or deleted after a period of time? (For example, customer accounts are to be deleted after three months or marked as obsolete and archived to a secondary database for future access.)
Authentication	Security requirements to ensure users are who they say they are.
Authorization	Security requirements to ensure users can access only certain functions within the application (by use case, subsystem, webpage, business rule, field level, etc.).
Legal	What legislative constraints is the system operating in (data protection, Sarbanes Oxley, GDPR, etc.)? What reservation rights does the company require? Any regulations regarding the way the application is to be built or deployed?
Privacy	Ability to hide transactions from internal company employees (encrypted transactions so even DBAs and network architects cannot see them).
Security	Does the data need to be encrypted in the database? Encrypted for network communication between internal systems? What type of authentication needs to be in place for remote user access?
Supportability	What level of technical support is needed by the application? What level of logging and other facilities are required to debug errors in the system?
Usability/achievability	Level of training required for users to achieve their goals with the application/solution. Usability requirements need to be treated as seriously as any other architectural issue.

Table from book “Richards & Ford: Fundamentals of Software Architecture”

Even more properties (contd.)

New Uzbekistan University

Table 4-3. Cross-c	
Term	
Accessibility	
Archivability	to be deleted
Authentication	
Authorization	se, subsystem,
Legal	hat reservation yed?
Privacy	network
Security	rnal systems?
Supportability	e required to debug errors in the system?
Usability/achievability	Level of training required for users to achieve their goals with the application/solution. Usability requirements need to be treated as seriously as any other architectural issue.

Of course, we don't have time to go through and study study these in any detail, or to try to discuss how any particular architecture might rate on any of them…

You could write a whole book about that…

Table from book "Richards & Ford: Fundamentals of Software Architecture"

Our goal this week is:

New Uzbekistan University

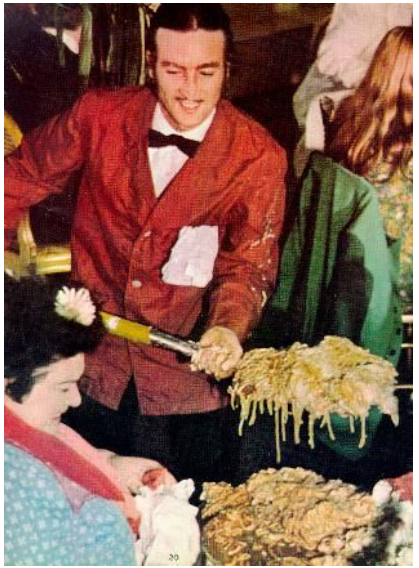
- Just talk about some different top-level organizations.
- And knowing the top-level organization gives you the first clue about:
  - How to understand the system
  - Where to look for bugs or explain behaviors
  - How to organize into teams
  - How to find modification and extension points

Remember the overall goal of making software systems understandable and manageable by humans.

Architecture #0: Monolithic

New Uzbekistan University

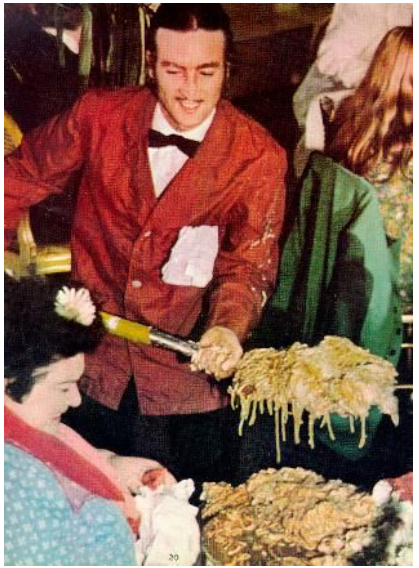
- A single app, with no particular organization
- People also call it “monolith”
  - Also informally as a "spaghetti code" because there is no organization at all in this, just putting bunch of code in one place
- May still have useful interfaces for some degree of encapsulation and modularity.



Architecture #0: Monolithic

New Uzbekistan University

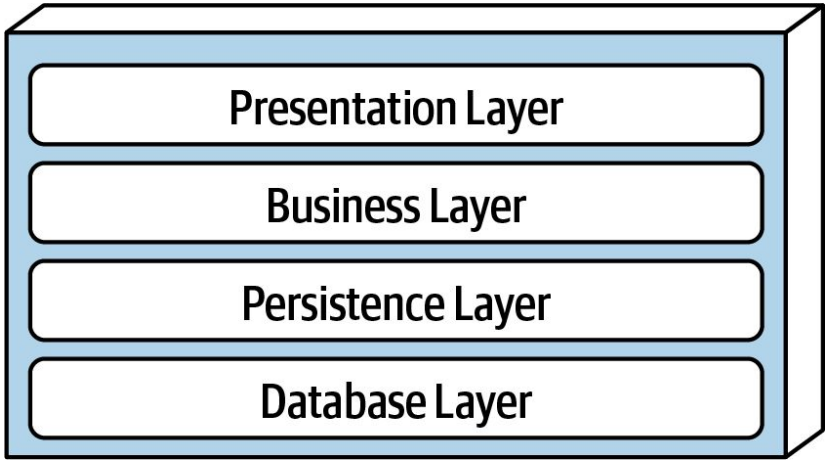
- OK for single-developer, short-lived projects
- But…
  - What happens if you want to add a new developer?
  - What happens if you need to come back to the code later?



Architecture #1: Layered

New Uzbekistan University

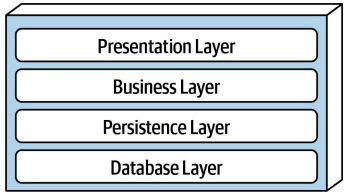
- Each layer depends on services from the layer above or below
- Organize teams by layers
  - Different layers require different expertise
- When the layers are run on separate pieces of hardware, they are sometimes called "tiers"



Architecture #1: Layered (contd.)

New Uzbekistan University

- Typical organization used in operating systems (OS)
- Layers communicate through procedure calls and callbacks (sometimes called "up-calls")



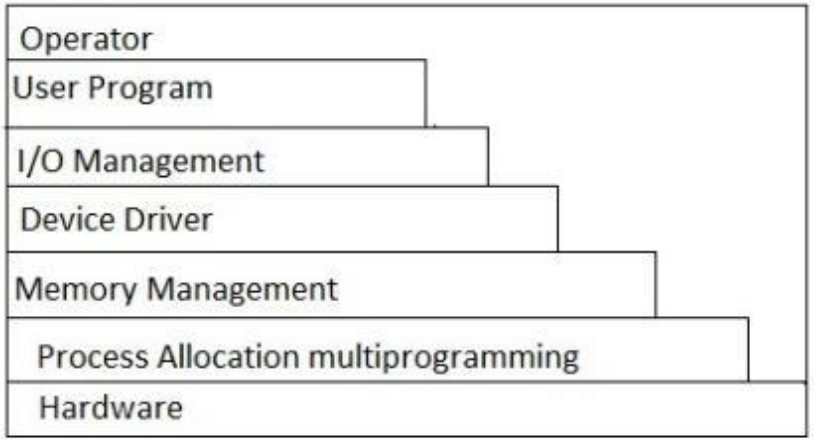
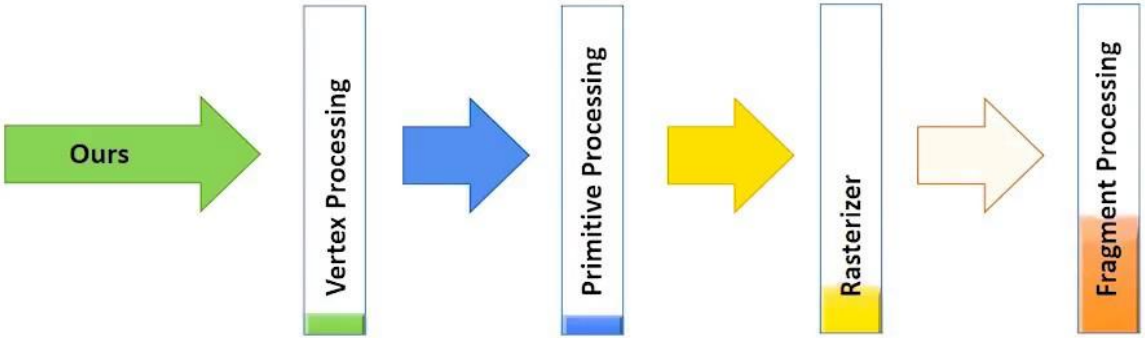


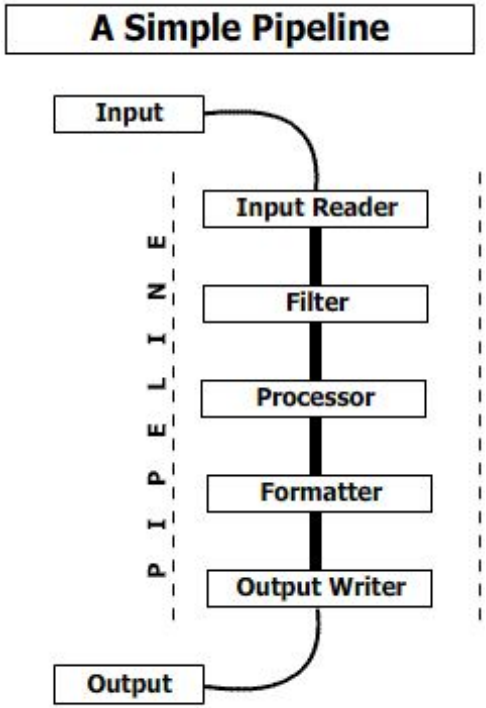
fig:- layered Architecture

Architecture #2: Pipeline

New Uzbekistan University

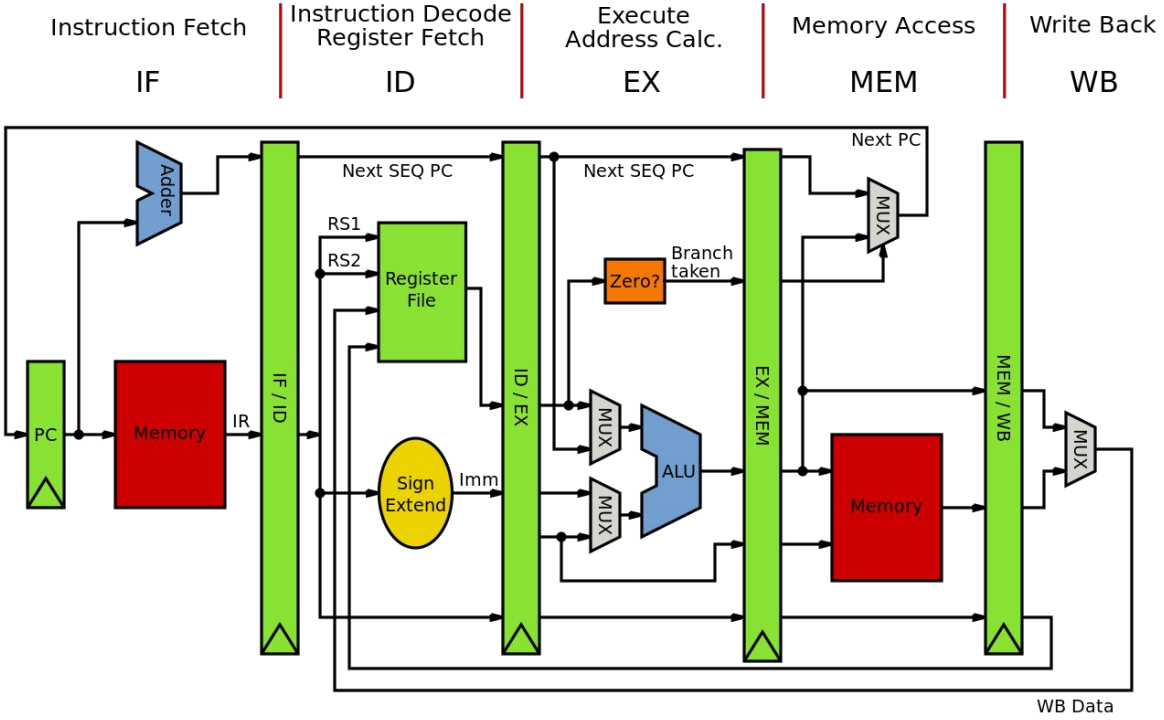
Good for complex straight-line processes, e.g.: image processing





Also good for visualizing hardware

New Uzbekistan University



How do the stages communicate?

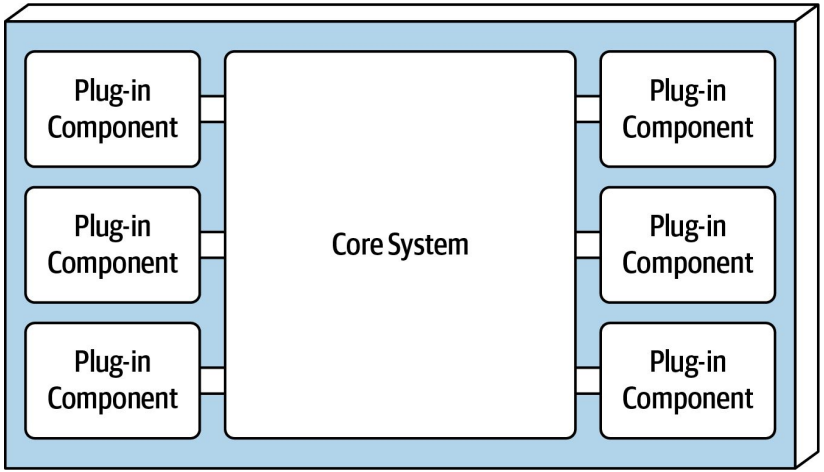
New Uzbekistan University

- That's the next-level decision
  - Data-push: each stage invokes the next
  - Demand-pull: each stage demands data from its predecessor
  - Queues, buffers…

Architecture #3: Plugins

New Uzbekistan University

- Highly extensible, also called “microkernels” architecture
- System consists of a small core components called the "microkernels"
- Includes lots of hooks for adding other services



- Each microkernels performs an essential functions (part of SW)
- Plug-ins can be designed by small, less-experienced teams– even by users!
- Connection methods may vary

Plugin architecture examples

New Uzbekistan University

- VSCode (internal org. + extension marketplace)
- Wordpress (defaults + extension marketplace)
- git clients
- etc.

```
$ ls .git/hooks
applypatch-msg.sample      pre-applypatch.sample      pre-rebase.sample
commit-msg.sample          pre-commit.sample          pre-receive.sample
fsmonitor-watchman.sample  prepare-commit-msg.sample  update.sample
post-update.sample         pre-push.sample
```

Express.js uses a microkernel architecture

New Uzbekistan University

- express.js depends on plug-ins:

```
app.get('/transcripts', (req,res) => {
   console.log('Handling GET/transcripts')
   let data = db.getAll()
   console.log(data)
   res.status(200).send(data)
})
```

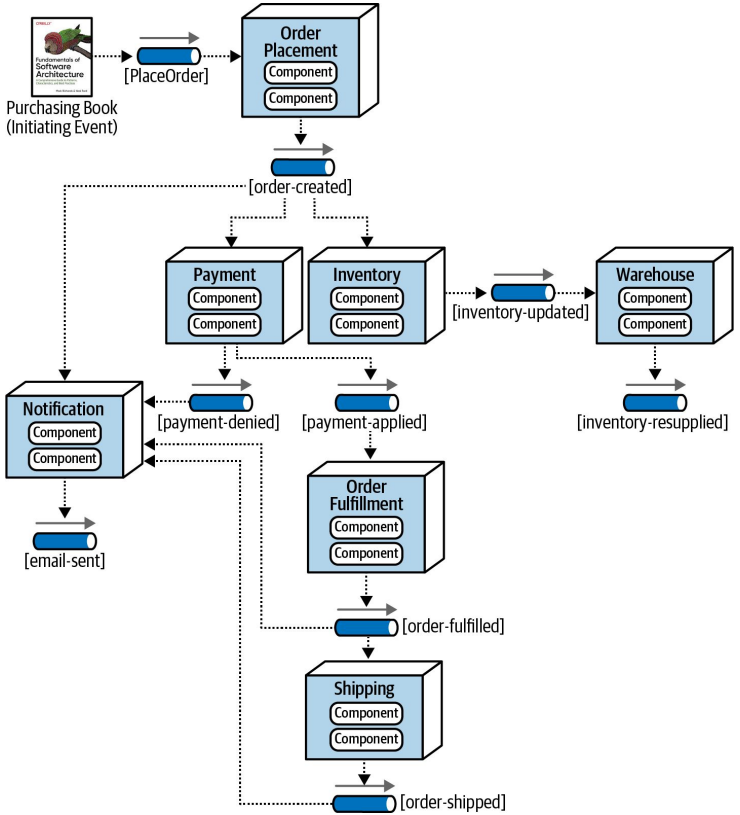
`app.get()` is a hook that adds a handler to the server.

The handlers are ordered (the first matching handler is executed), and can be pipelined, so a handler can invoke another handler if desired.

Architecture #4: Event-driven

New Uzbekistan University

- Metaphor: a bunch of bureaucrats shuffling papers
- Each processing unit has an in-box and one or more out-boxes
- Each unit takes a task from its inbox, processes it, and puts the results in one or more outboxes.
- Stages are typically connected by asynchronous message queues.
- Conditional flow



Architecture #5: Microservice

New Uzbekistan University

- Architecture where the overall task (or SW logic) is divided into different SW components
- Each component is implemented independently
  - Each component is independently replaceable and updatable
  - Parts of SW are highly decoupled!
- Components can be built as libraries, but more usually as web services
  - Services communicate via HTTP, typically REST

Microservice example by AWS

New Uzbekistan University

"Microservices architectures separate functionalities into cohesive verticals according to specific domains, rather than technological layers."

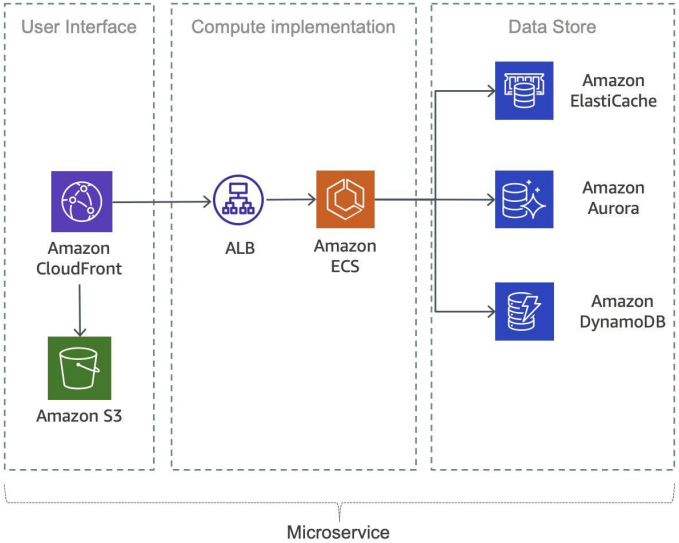
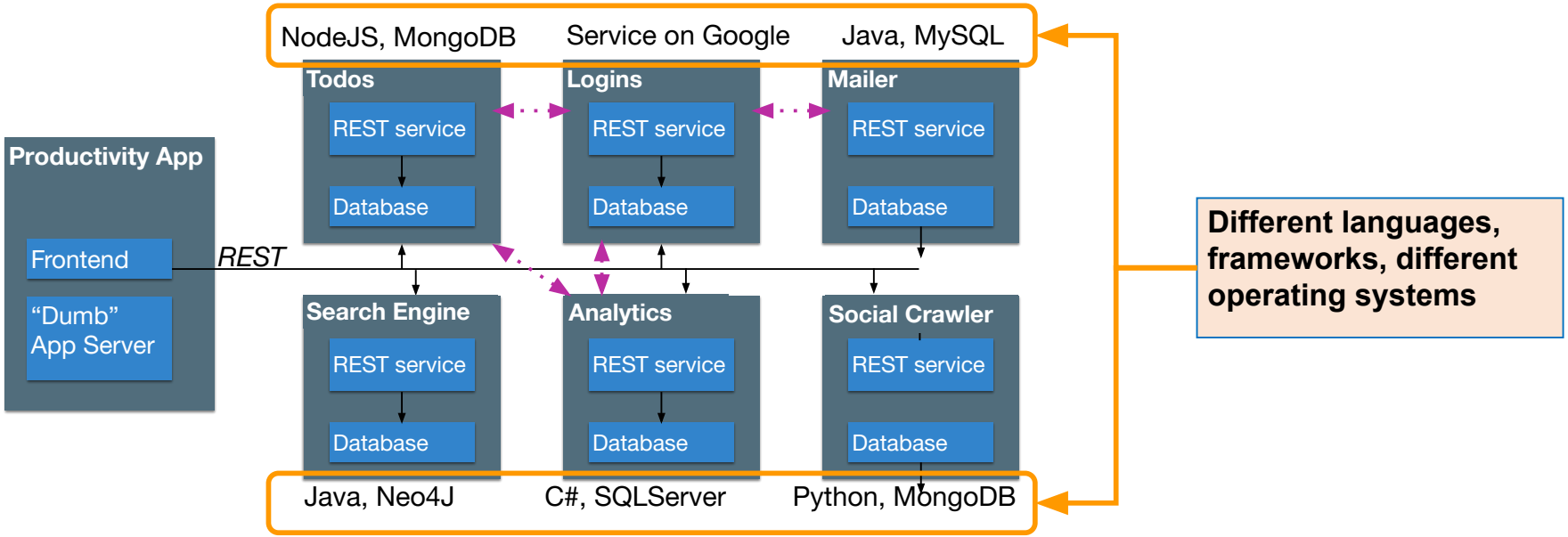


Figure from AWS whitepaper on "Implementing Microservices on AWS"
https://docs.aws.amazon.com/whitepapers/latest/microservices-on-aws/simple-microservices-architecture-on-aws.html

Another schematic example of microservice

New Uzbekistan University



Microservice: pros and cons

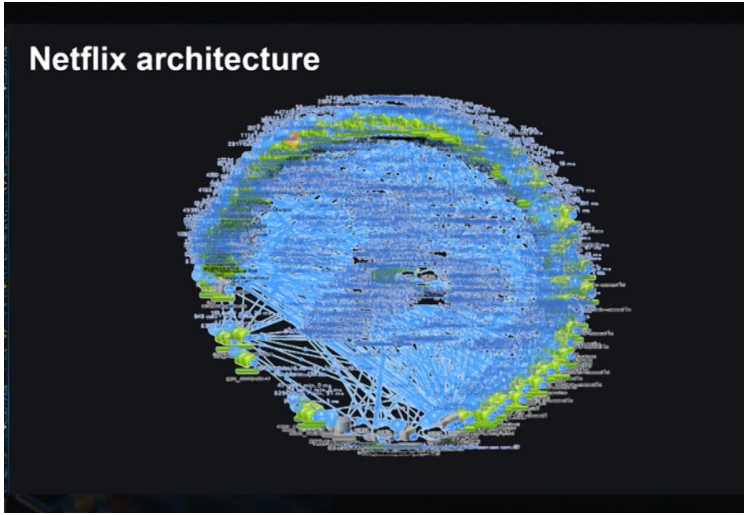
New Uzbekistan University

- Pros
  - Services may scale differently, so can be implemented on hardware appropriate for each resource - CPU, RAM, Disk, etc.
  - Great for software part too - language, framework, OS, etc.
  - Services are independent, so parts of SW can be developed and deployed independently (by completely separate teams)
- Cons
  - Service discovery?
  - Should services have some organization, or are they all equals?
  - Overall system complexity

Architecture #5: Microservice

New Uzbekistan University

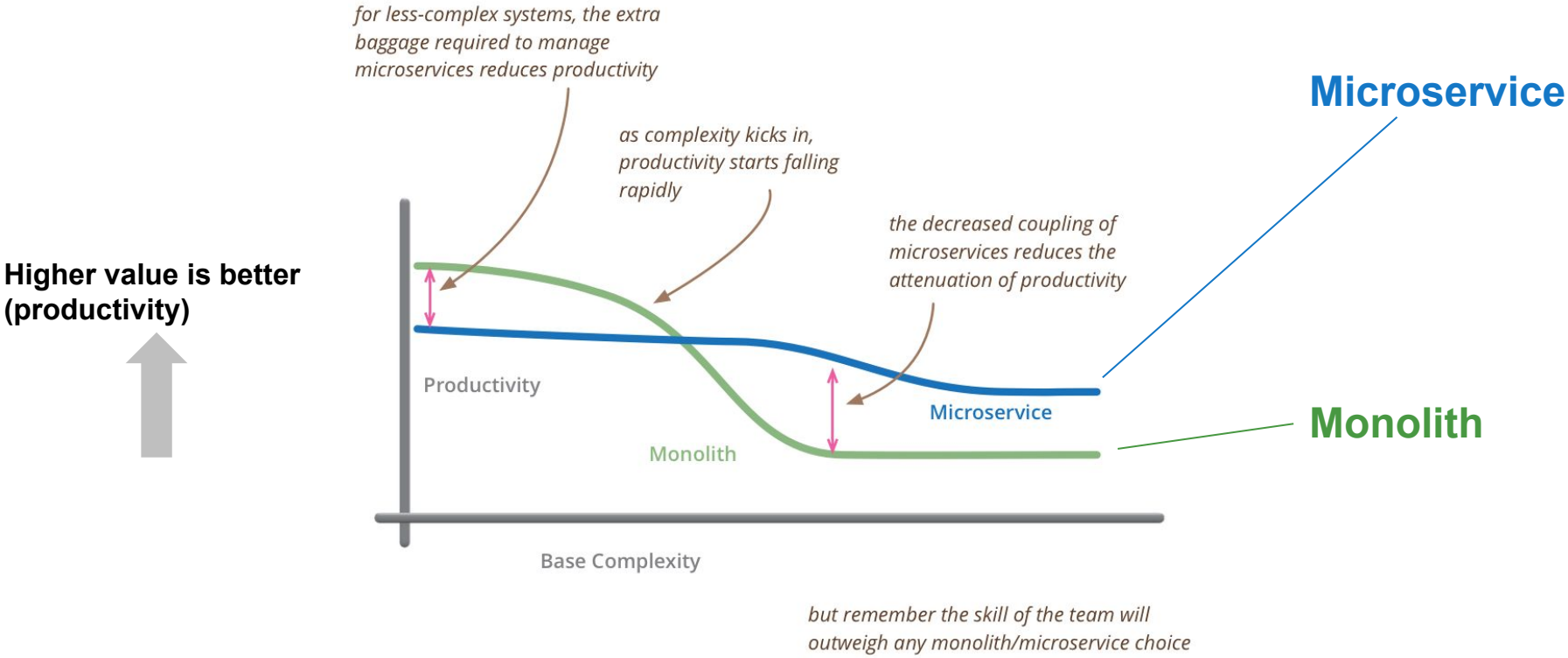
- Microservices are
  - Highly scalable
  - Trendy
- For example - microservices at Netflix:
  - 100s of microservices
  - 1000s of daily production changes
  - 10,000s of instances
  - BUT… only 10s of operations engineers



https://medium.com/refraction-tech-everything/how-netflix-works-the-hugely-simplified-complex-stuff-that-happens-every-time-you-hit-play-3a40c9be254b

Monoliths vs. microservices

New Uzbekistan University



Martin Fowler's Microservices Guide - https://martinfowler.com/microservices/

HTTP basics

What is HTTP?

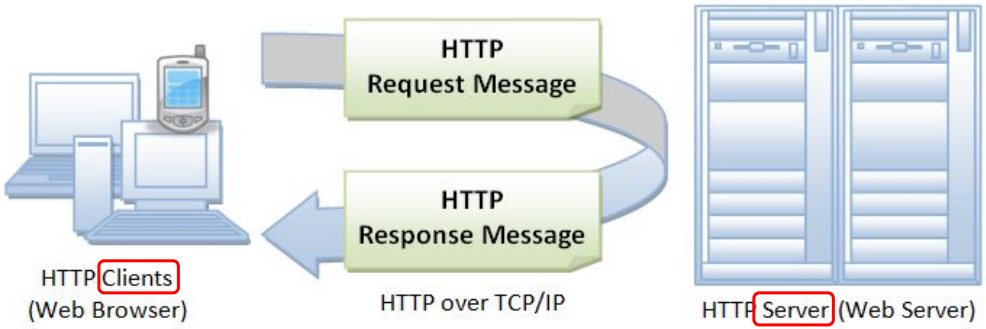
New Uzbekistan University

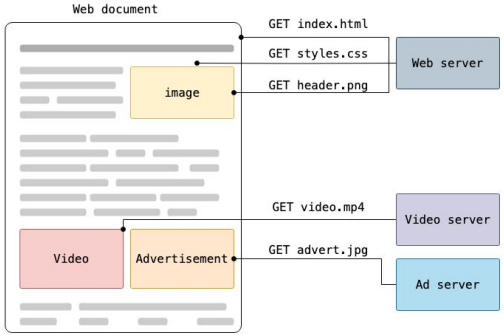
- HTTP stands for "HyperText Transfer Protocol"
- Originally developed (1989-1991) by Tim Berners-Lee as a protocol for transmitting web pages.
  - HTML is a hypertext markup language and HTTP is a network protocol for transferring the hypertext across the network.
- Over time HTTP has evolved into a general-purpose character-based protocol for communicating on the Web.

HTTP is asymmetric

New Uzbekistan University

- It distinguishes between client and server.
- The client initiates a request, and the server replies by sending a response.





HTTP is stateless

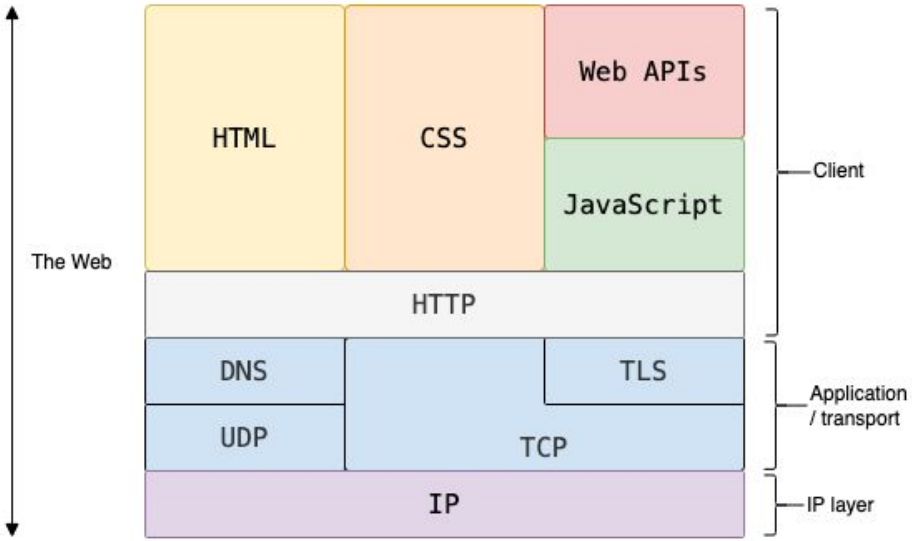
New Uzbekistan University

- Each request / response pair is independent.
- If the client intends the request to be part of a session, then the request must include all of the data needed to allow the server to resume the session at that point.
  - We'll see later how this data can be included in the request.
- We say "HTTP is stateless but not sessionless"

HTTP is an application layer protocol

New Uzbekistan University

- HTTP is an application layer protocol sitting on top of TCP/IP layer
- Data are sent over TCP, or over a TLS-encrypted TCP connection
- Used for fetching hypertext documents, images, videos and other data.
- HTTP can also be used to fetch parts of documents to update a webpage on demand.

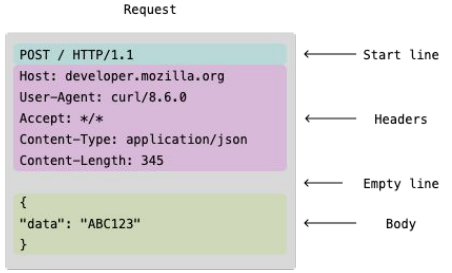


https://developer.mozilla.org/en-us/docs/Web/HTTP/Overview

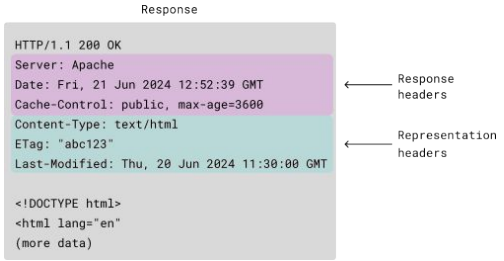
HTTP client-server communication

New Uzbekistan University

- HTTP request (client → server)
  - Client's HTTP request includes: start-line, headers, and body.
  - Headers define the kind of data (HTML, JSON, image, video, etc.) the client is sends and/or expects to receive in response to the request.
  - Headers may also include many other kinds of metadata, and the set of possible headers has grown tremendously over the years.



- HTTP response (client ← server)
  - Server's HTTP response also includes: start-line, headers, and body.
  - Headers indicate whether the request was successful or not, the formats in which the reply data is transmitted, etc.



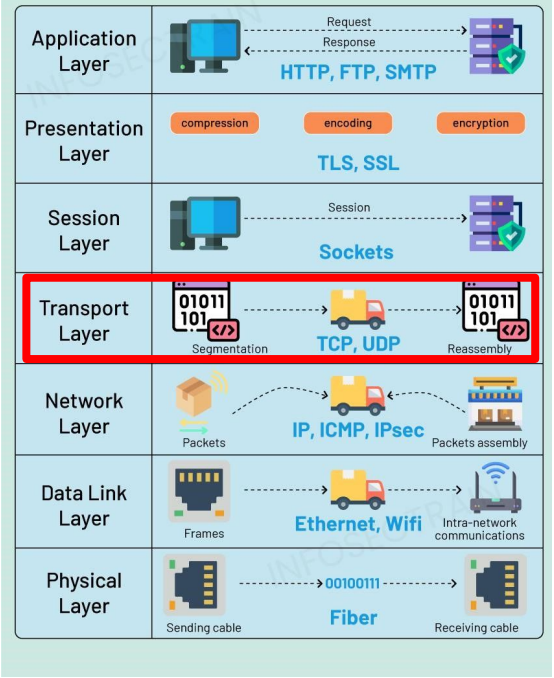
- More technical description of HTTP messages: https://developer.mozilla.org/en-US/docs/Web/HTTP/Guides/Messages

Flow of HTTP messages

New Uzbekistan University

Step 1: Client opens a TCP connection

- This is the connection over which data will flow
- Client can create a fresh connection or re-use an existing one.

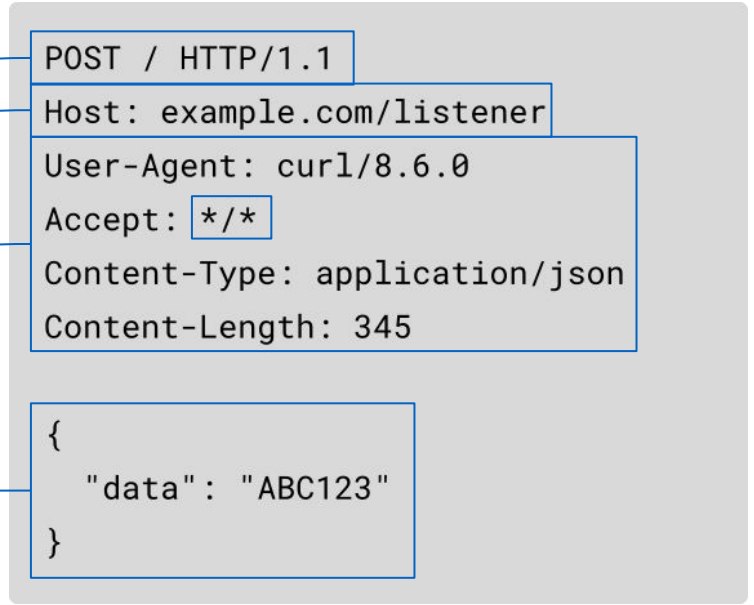


Flow of HTTP messages

New Uzbekistan University

Step 2: Client sends an HTTP request

- Request method (verbs)
- Version
- Path (URL)
- Headers
- Body (optional)
  - Separated from the headers by a blank line



HTTP methods (verbs)

New Uzbekistan University

- Client's each request comes with a method.
- There are just a few of these. Here are the ones that you should know (often called CRUD operations):
  - GET: requests a representation of a resource
  - POST: requests the server create a new resource
  - PUT: requests the server to replace a resource
  - DELETE: requests the server to delete a resource
- These methods often map to the CRUD operations in database
  - Acronym refers to the four main operations for data management: creating, reading, updating, and deletion

Flow of HTTP messages

New Uzbekistan University

Step 3: Server interprets the request

- “example.com” identifies the host (the server's location)
- the rest of the request is the path here “/listener”
  - this might be a path in the server's file system
  - or it could be anything at all…
  - it's entirely up to the server to interpret the path

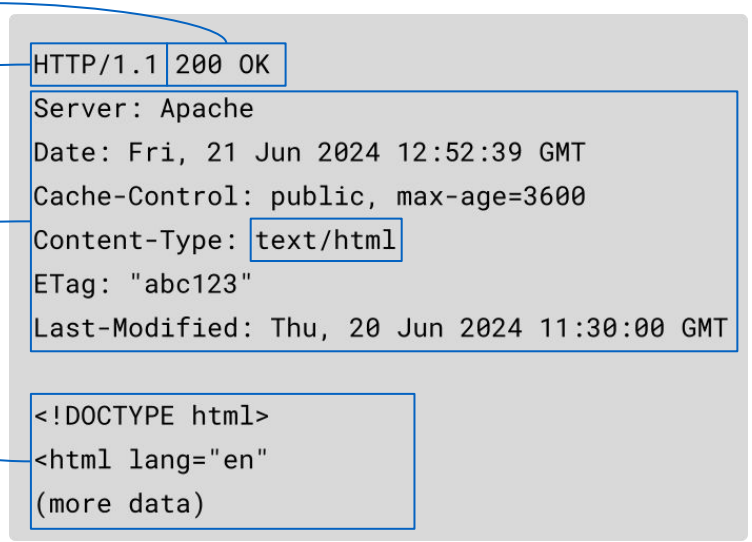
```
POST / HTTP/1.1
Host: example.com/listener
User-Agent: curl/8.6.0
Accept: */*
Content-Type: application/json
Content-Length: 345

{
  "data": "ABC123"
}
```

Flow of HTTP messages

New Uzbekistan University

Step 4: Server sends a response

- Status code & message
- Version
- Some headers
- And maybe a response body



Flow of HTTP messages

New Uzbekistan University

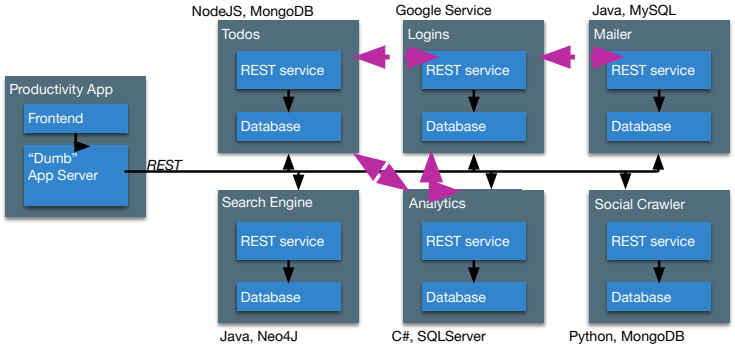
Step 5: Client closes or reuses the connection

REST protocols

Your app relies on other apps for services

New Uzbekistan University

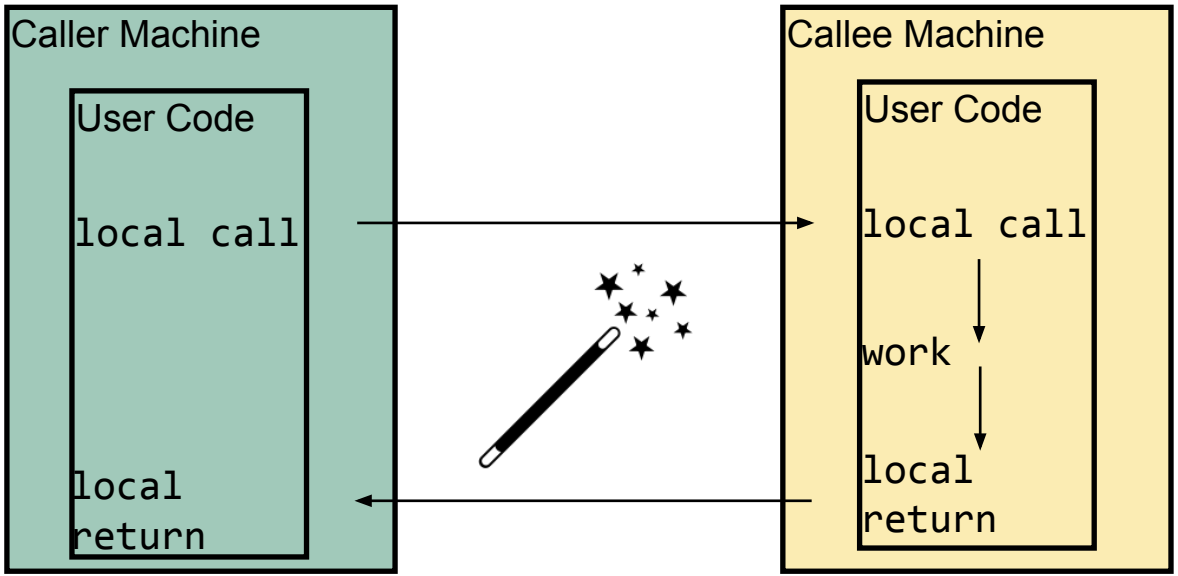
- Authentication
  - Login with Google / Apple / Facebook
  - SMS auth with Firebase Auth
- Sending / receiving email
  - SendGrid, MailGun, MailChimp
- Telephony, text messaging, video chat
  - Twilio APIs
- AI integration
  - OpenAI APIs



What we'd like

New Uzbekistan University

- A magic abstraction: remote procedure call (RPC)



Obstacles to magic RPC

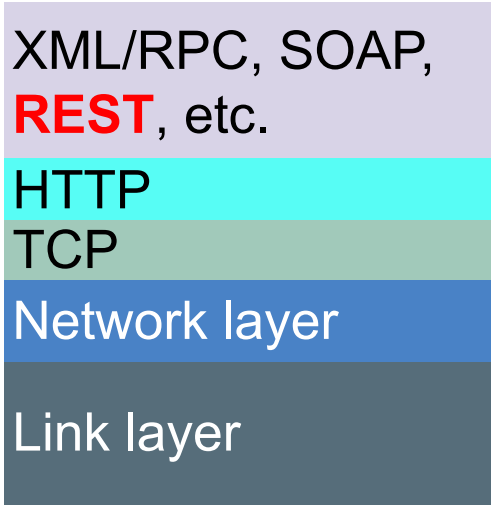
New Uzbekistan University

- Transmission delays (latency)
- Can the client do something useful in the meantime?
  - Asynchrony
  - "mask latency with multiprocessing" → complexity
- Client / server mismatch
  - Different languages
  - Different data representations
  - Wire-transmission formats
  - More complexity

Solution - just use the web!

New Uzbekistan University

- Implement your protocol via HTTP.
- Of course, then you have to define your protocol.
- You'll want to define it in some standard metalanguage, so client and server can agree on its meaning.
- But that means the client-human and server-human have to agree on a standard metalanguage.
- Lots of choices: XML/RPC, SOAP, WSDL, or ...



Recap slide 41: client-server comm.

New Uzbekistan University

Slide 41 says:

Step 3: Server interprets the request

- "example.com" identifies the host (the server's location)
- the rest of the request is the path here "/listener"
  - this might be a path in the server's file system
  - or it could be anything at all…
  - it's entirely up to the server to interpret the path

```
POST / HTTP/1.1
Host: example.com/listener
User-Agent: curl/8.6.0
Accept: */*
Content-Type: application/json
Content-Length: 345

{
  "data": "ABC123"
}
```

- That means the client can ask the server to **do things** other than retrieve files.
- Just there has to be an **agreement** (a protocol) between **client** and **server** about **how these tasks are to be described**.
- Need a general framework to help us design such protocols.
- We will talk about one such philosophy, called **REST**.

REST: Representational State Transfer

New Uzbekistan University

- Defined by Roy Fielding in his 2000 Ph.D. dissertation (UCI)

> "Throughout the HTTP standardization process, I was called on to defend the design choices of the Web. That is an extremely difficult thing to do... I had comments from well over 500 developers, many of whom were distinguished engineers with decades of experience. That process honed my model down to a **core set of principles, properties, and constraints** that are now **called REST**."

- REST is not just a transport protocol… nor a protocol definition language… **REST is a SW design philosophy**.
- Interfaces that follow REST principles are called **RESTful**.

REST principles

New Uzbekistan University

Single server

- Client calls server, server responds. That's it.
- Separation of concerns: client doesn't worry about data, server doesn't worry about UI.
- Server may pass request on to other machines, but that's not visible to the client.

Stateless

- No session state in the server.
- Each client request must contain all the information the server needs to process the request.

Uniform interface

- Associate URIs with resources.

Uniform cacheability

- Requests must classify themselves as cacheable or not.

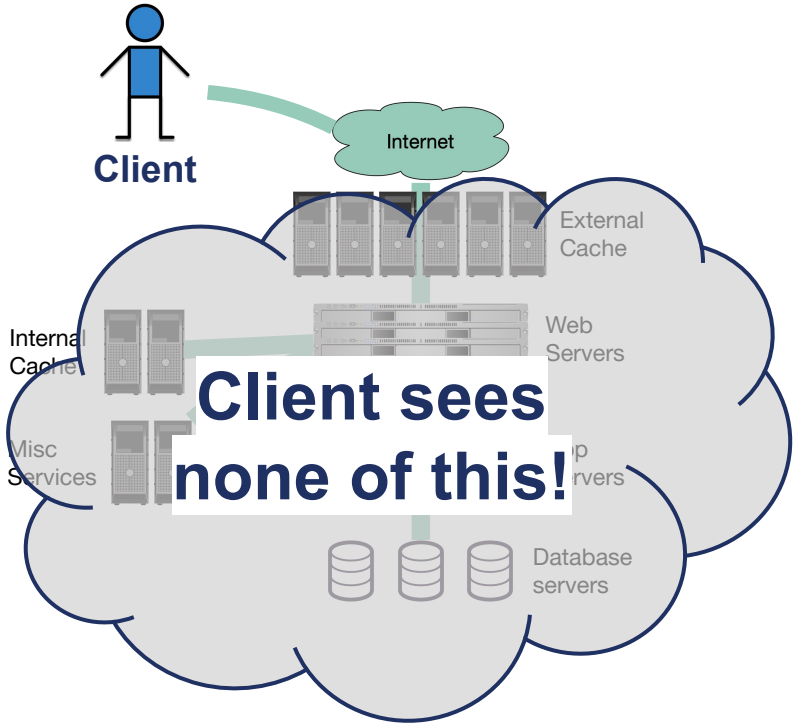
REST principles: single server

New Uzbekistan University

- Server is abstracted as a single box
- Client calls the server, server doesn’t call the client
- Enables separation of concerns:
  - Client doesn’t worry about how the server does its business
  - Server doesn't worry about UI

REST principles: single server

New Uzbekistan University

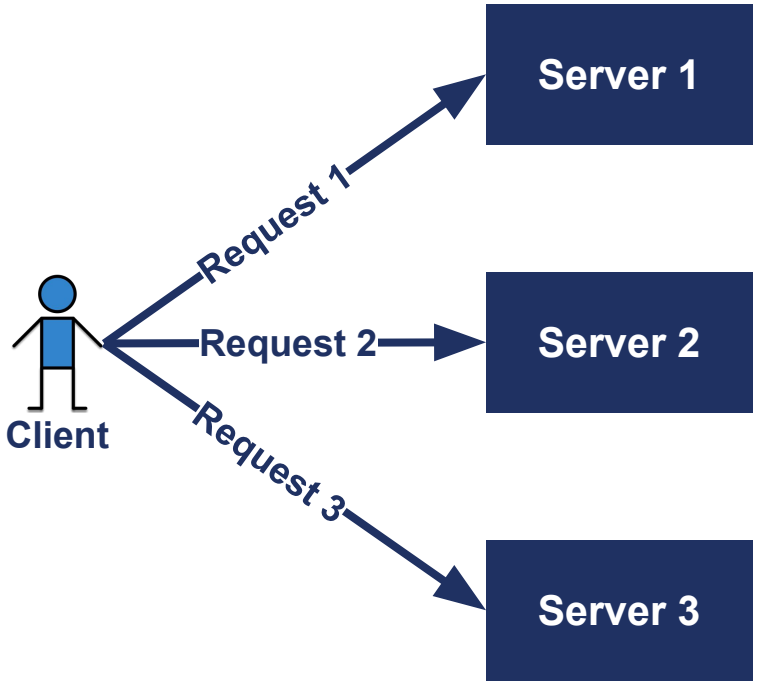


- Enables a flexible SW design.
- Different servers can have different responsibilities.
- Client sees just a single server.

REST principles: stateless

New Uzbekistan University

- Each client request contains all information necessary to service the request.
- The client doesn't have to write a sequence of requests to get their work done.
- So requests can be farmed out to different servers.



REST principle: uniform interface

New Uzbekistan University

- URIs should hierarchically identify nouns describing resources that exist
- Actions that can be taken with resources are specified by the HTTP methods (verbs)
- So, in summary: nouns for resources, verbs for actions.

Uniform interface: the nouns

New Uzbekistan University

- Nouns are represented as **URIs** (uniform resource identifiers)
- In a RESTful system, the server is visualized as a store of resources (**nouns**), each of which has some data associated with it.
- URIs represent these resources. Examples:
  - `/cities/losangeles`
  - `/transcripts/220000/undergraduate`
    (Student ID 0000 has several transcripts in the system; this is the undergraduate one)
- Bad naming examples that fail REST principles
  - `/getCity/losangeles`
  - `/getCitybyID/50654`
  - `/Cities.php?id=50654`

Uniform interface: the verbs

New Uzbekistan University

- Verbs are represented as **HTTP methods**
- In REST, there are mainly four things you do with a resource
  1. GET: requests the server to respond with a representation of the resource
  2. POST: requests the server to create a resource
     (there are several ways in which the value for the new resource can be transmitted)
  3. PUT: requests the server to replace the value of the resource by the given value
  4. DELETE: requests the server to delete the resource

But there's more available: PATCH (like PUT, but only partial modifications), HEAD (like GET, but only metadata), and OPTIONS (what method is allowed on URL?)

Request parameters

New Uzbekistan University

There are (at least) 3 ways to associate parameters with a request:

1. **Path parameters**
   - Specify portions of the path to the resource.
   - For example, your REST protocol might allow a path like:
     `/transcripts/220000/undergraduate`

2. **Query parameters**
   - These are part of the URI and are typically used as search items.
   - For example, your REST protocol might allow a path like:
     `/transcripts/undergraduate?lastname=Burxonov&firstname=Sardor`

3. **Request body** – fully customizeable

Example interface: a todo app

New Uzbekistan University

- Resource: `/todos`
  - `GET /todos` - get list all of my todo items
  - `POST /todos` - create a new todo item (data in body)

- Resource: `/todos/:todoItemID`

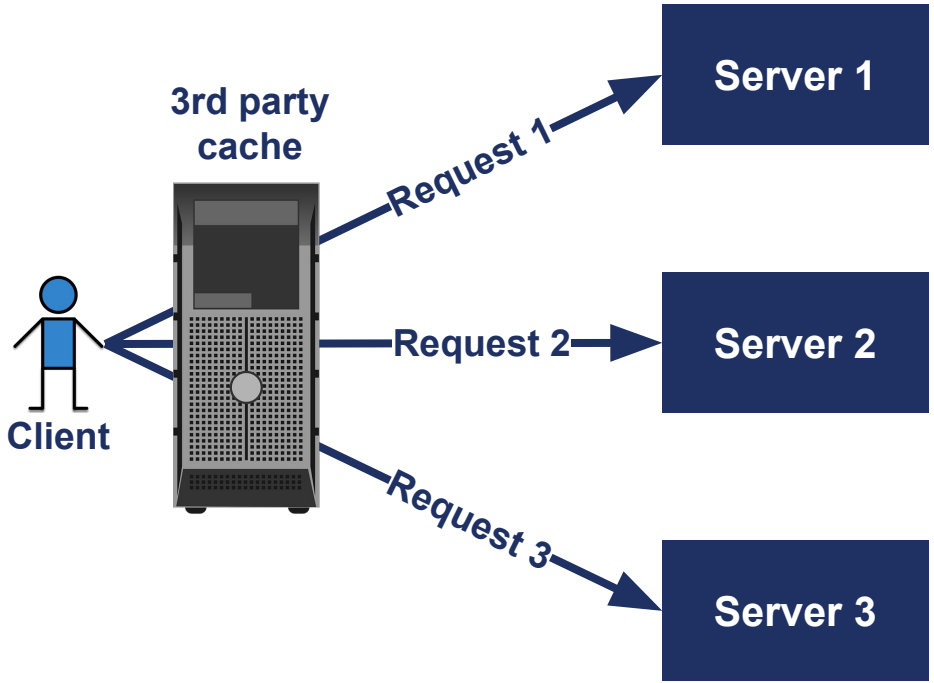
a path parameter

  - `GET /todos/:todoItemID` - fetch a single todo item by id
  - `PUT /todos/:todoItemID` - update a single todo item (data in body)
  - `DELETE /todos/:todoItemID` - delete a single todo item

REST principle: uniform cacheability

New Uzbekistan University

- Requests and responses are clearly classified as cacheable or not.
- Enables use of generic caches that don't know anything about the structure of what they cache - just what can be cached.




Redo
Undrln
Q & A
Find
J
C
V
B
N