CS 4530: Fundamentals of Software Engineering
Module 10: Application Level Patterns

Adeel Bhutta, Joydeep Mitra and Mitch Wand

Khoury College of Computer Sciences

Learning Objectives for this Module

- By the end of this module you should be able to:
    - describe the basic ideas of the following architectures, with examples and pictures
        - anarchic
        - layered
        - pipeline
        - event-driven
        - microkernel
        - microservice
    - describe the main features of the following communication modalities:
        - procedure calls
        - HTTP and REST
        - Websockets

Three Scales of Design

The Structural Scale

- key questions: what are the pieces? how do they fit together to form a coherent whole?

The Interaction Scale

- key questions: how do the pieces interact? how are they related?

The Code Scale

- key question: how can I make the actual code easy to test, understand, and modify?

Design at larger scales

- Metaphor: building architecture
- How do the pieces fit together? Are there parts we can reuse?
- Will the result be structurally sound? earthquake-resistant? economical to build? easy to maintain?

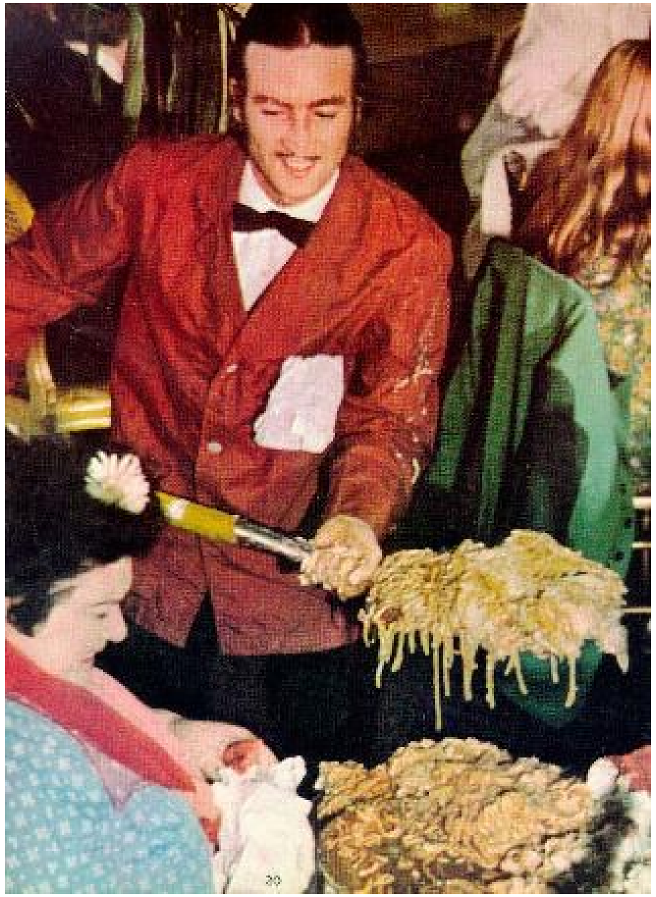
Goal: Create a high-level picture of the system

- Abstract details away into reusable components
- Allows for analysis of high-level design before implementation
- Enables exploration of design alternatives
- Reduce risks associated with building the software

Architecture #0: Anarchic

- A single app, with no particular organization
- Also known as: "spaghetti code"
- May still have useful interfaces for some degree of encapsulation and modularity.
  - but is there a method to the madness?

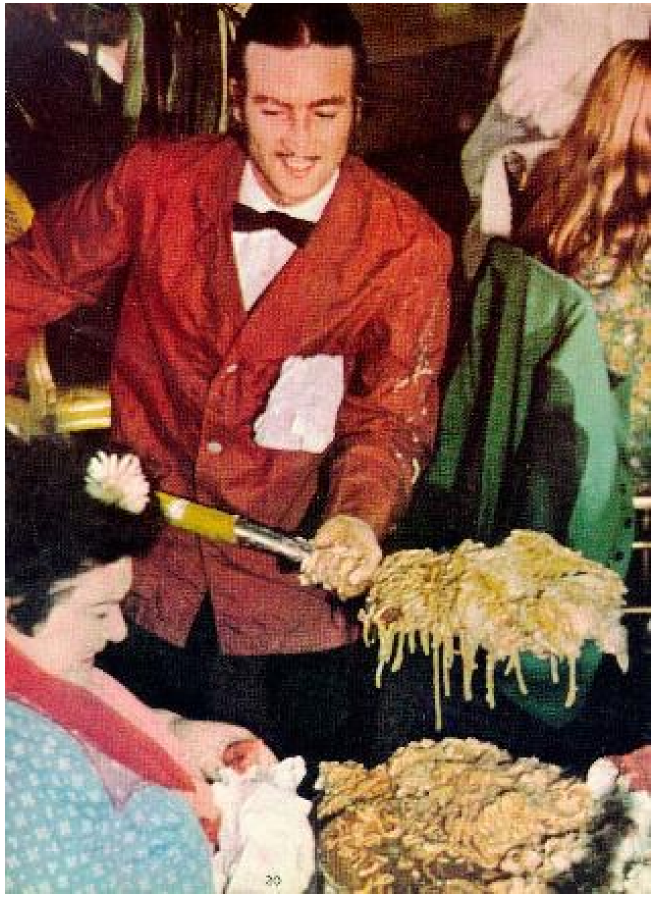
Shakespeare, *Hamlet*. The exact quote is: "Though this be madness, yet there is method in't" (Polonius, Act 2, Scene 2)



Brian Foote and Joe Yoder

Architecture #0: Anarchic

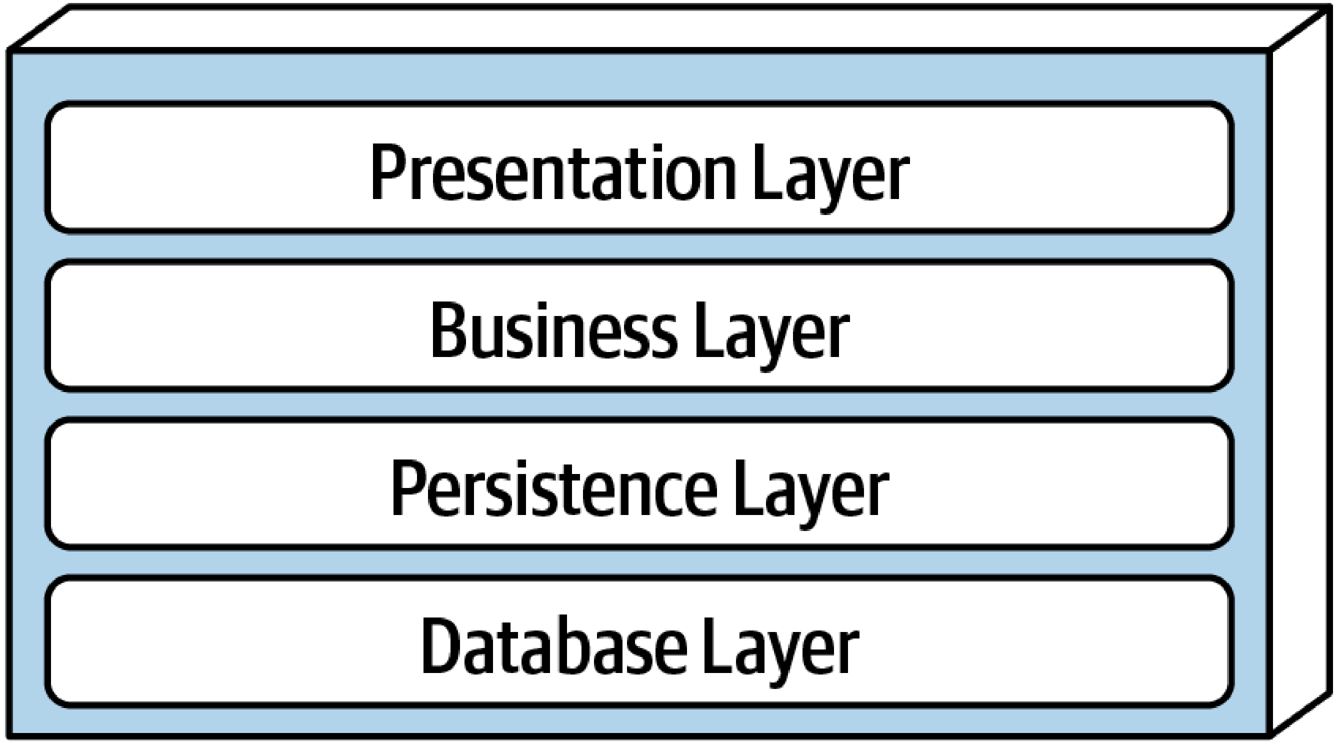
- OK for single-developer, short-lived projects
- But
  - what happens if you want to add a new developer
  - what happens if you need to come back to the code later?



Brian Foote and Joe Yoder

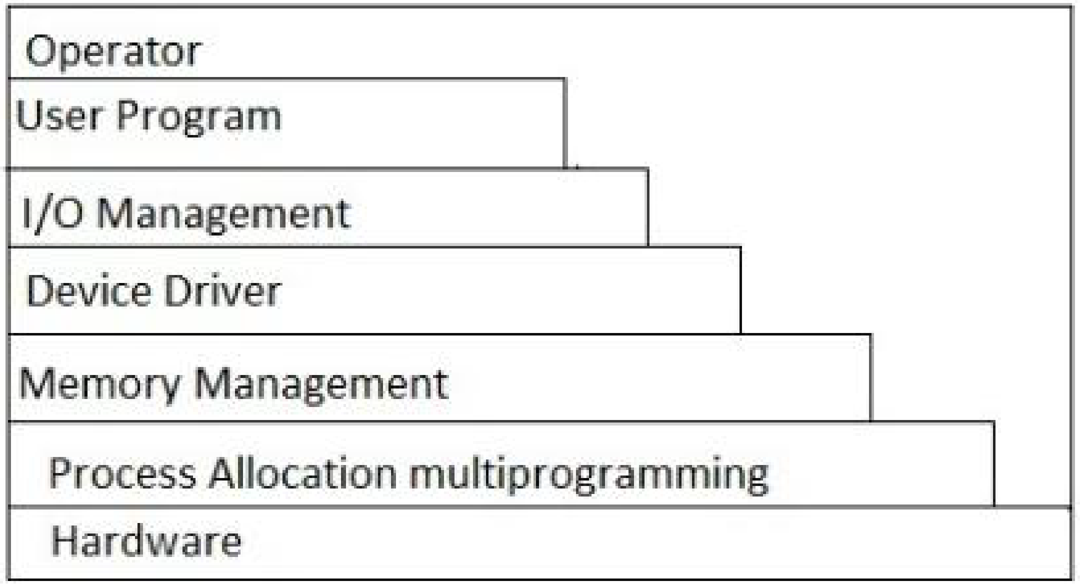
Architecture #1: Layered

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



Layered Architecture (contd)

- Typical organization for operating systems
- Layers communicate through procedure calls and callbacks ("up-calls")
- Well-defined interfaces are a must!



Operator

User Program

I/O Management

Device Driver

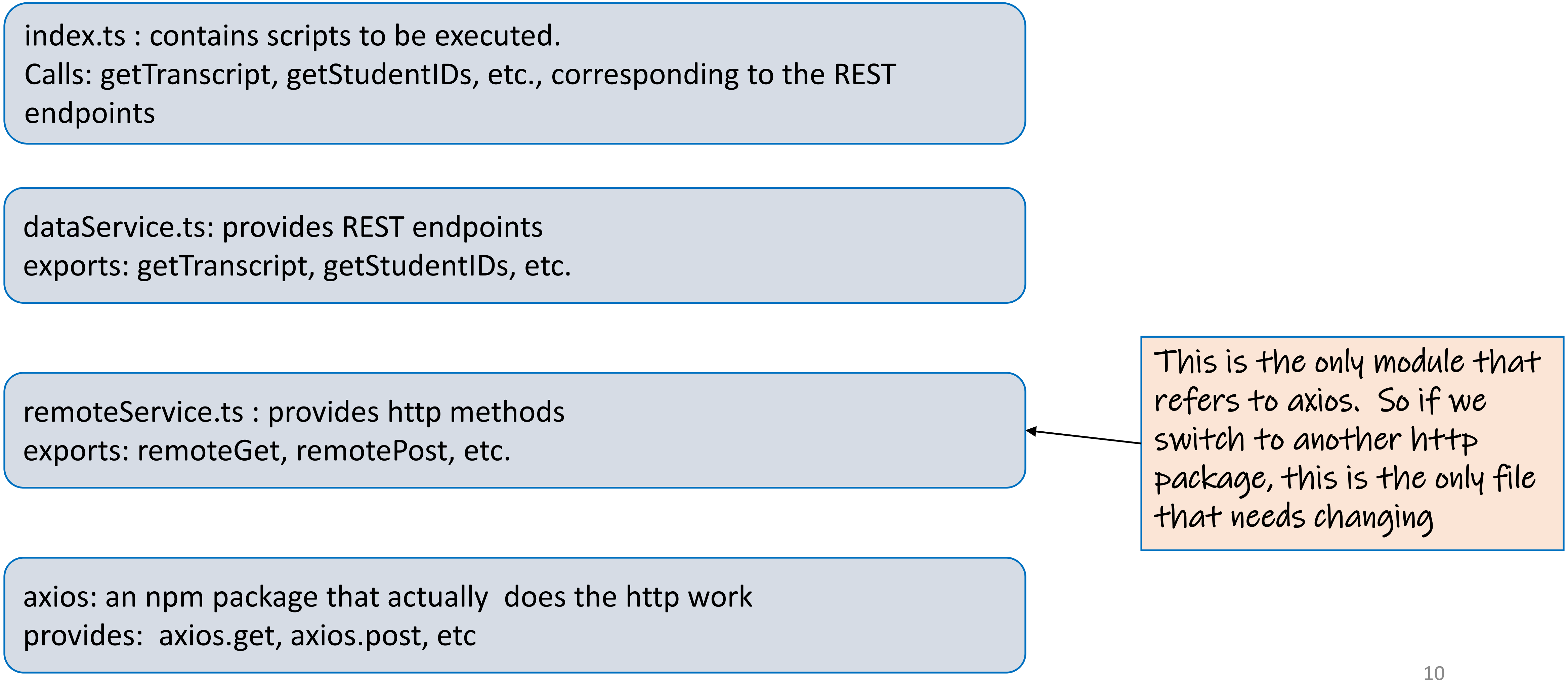
Memory Management

Process Allocation multiprogramming

Hardware

fig:- layered Architecture

Layers from a Spring '21 example



index.ts : contains scripts to be executed.
Calls: getTranscript, getStudentIDs, etc., corresponding to the REST endpoints

dataService.ts: provides REST endpoints
exports: getTranscript, getStudentIDs, etc.

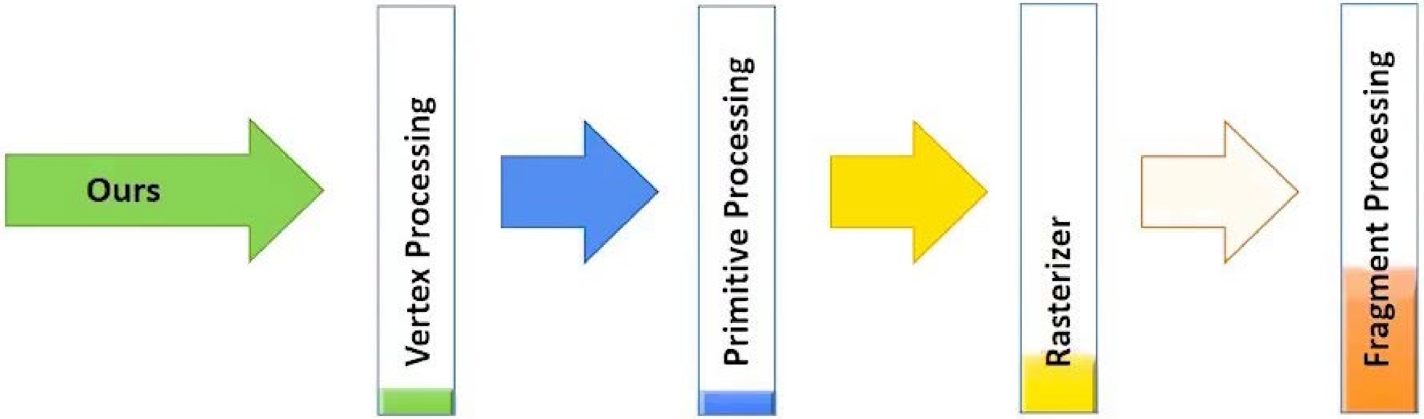
remoteService.ts : provides http methods
exports: remoteGet, remotePost, etc.

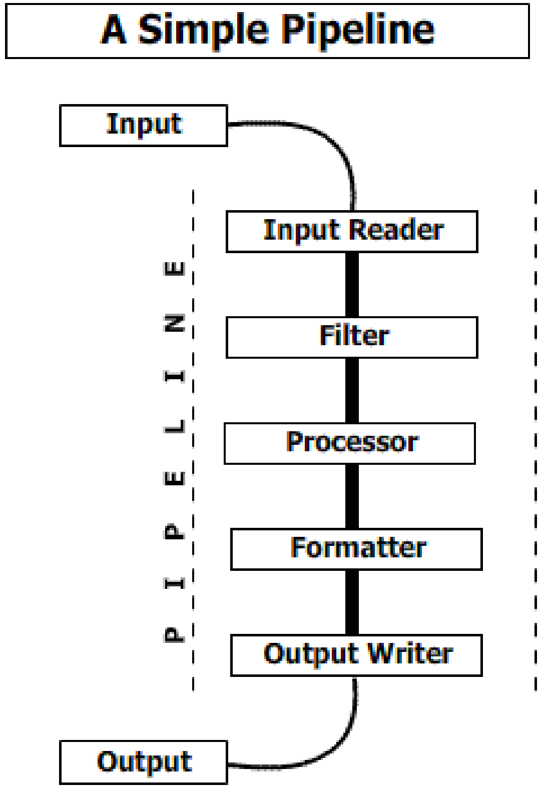
axios: an npm package that actually does the http work
provides: axios.get, axios.post, etc

This is the only module that refers to axios. So if we switch to another http package, this is the only file that needs changing

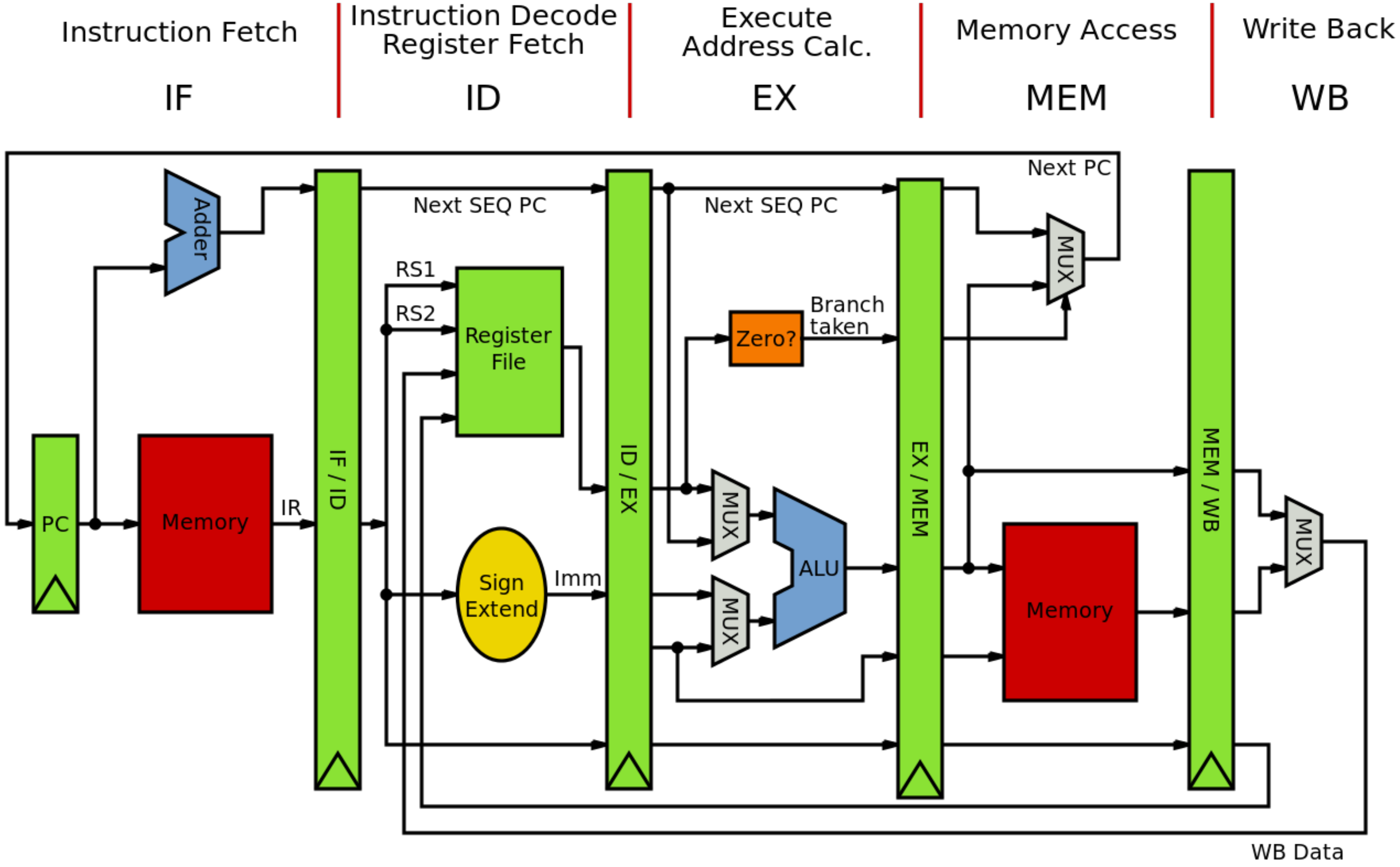
Architecture #2: Pipeline

- Good for complex straight-line processes, eg image processing





Also good for visualizing hardware



How do the stages communicate?

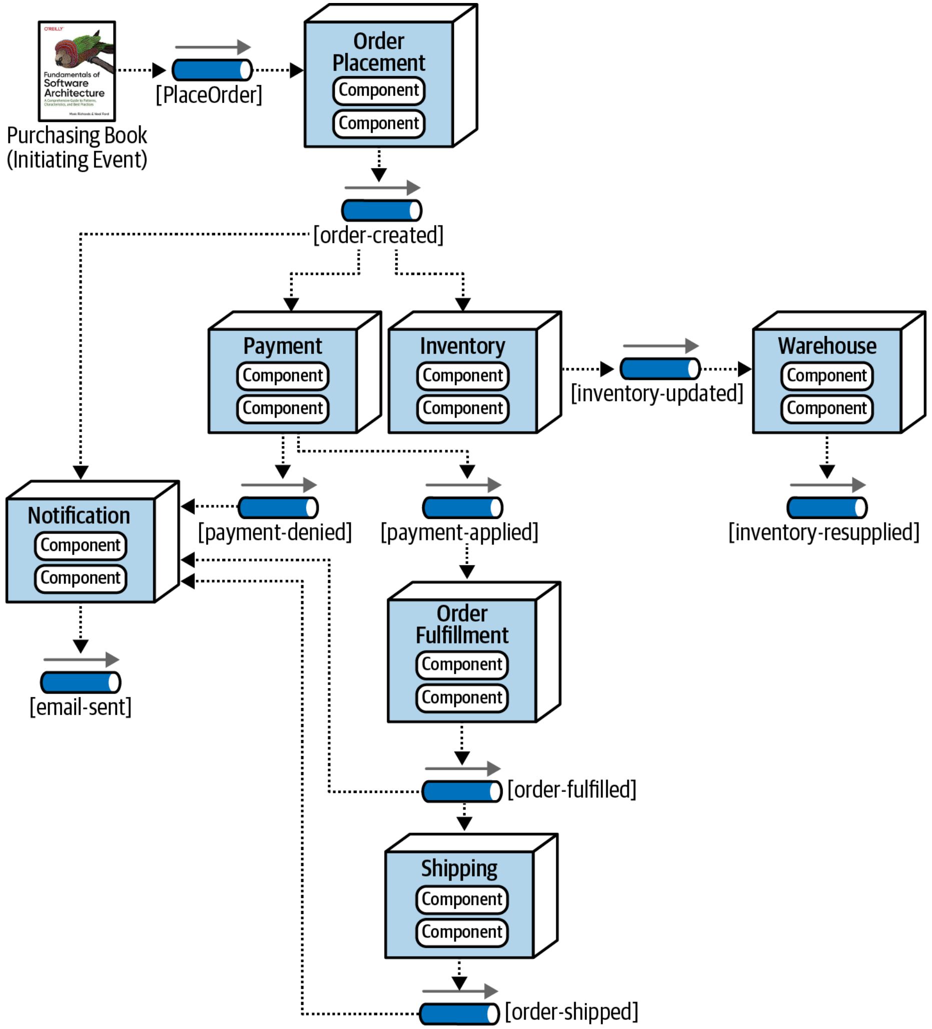
- That's the next-level decision
  - data-push (each stage invokes the next)
  - demand-pull (each stage demands data from its predecessor)
  - queues? buffers?
  - ??

In Express, each stage gets an object that represents the rest of the pipeline

```
app.use((req, res) => {
  res.status(404).json({
    error: 'Not Found',
    message:
      `Route ${req.method} ${req.originalUrl} not found`
  });
```

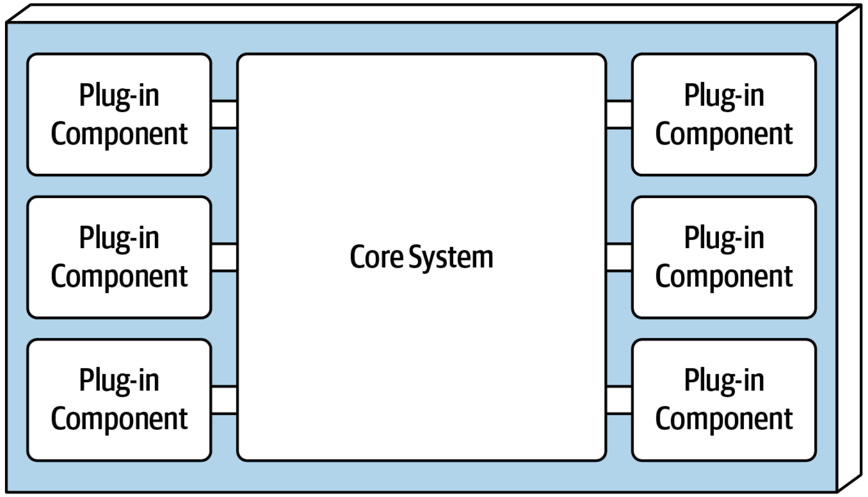
Architecture #3: Event-Driven Architecture

- 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 may be connected by asynchronous message queues.
- Or use the observer pattern, where each unit observes changes in its upstream units.
- Conditional flow



Architecture #4: Plugins ("microkernel")

- System consists of a small core (the "microkernel") for essential functions, and lots of hooks for adding other services
- Highly extensible
- Plug-ins can be designed by small, less-experienced teams– even by users!
- Connection methods may vary
  - often: core provides default behaviors that are overridable



Key Concepts for Plugin Architecture

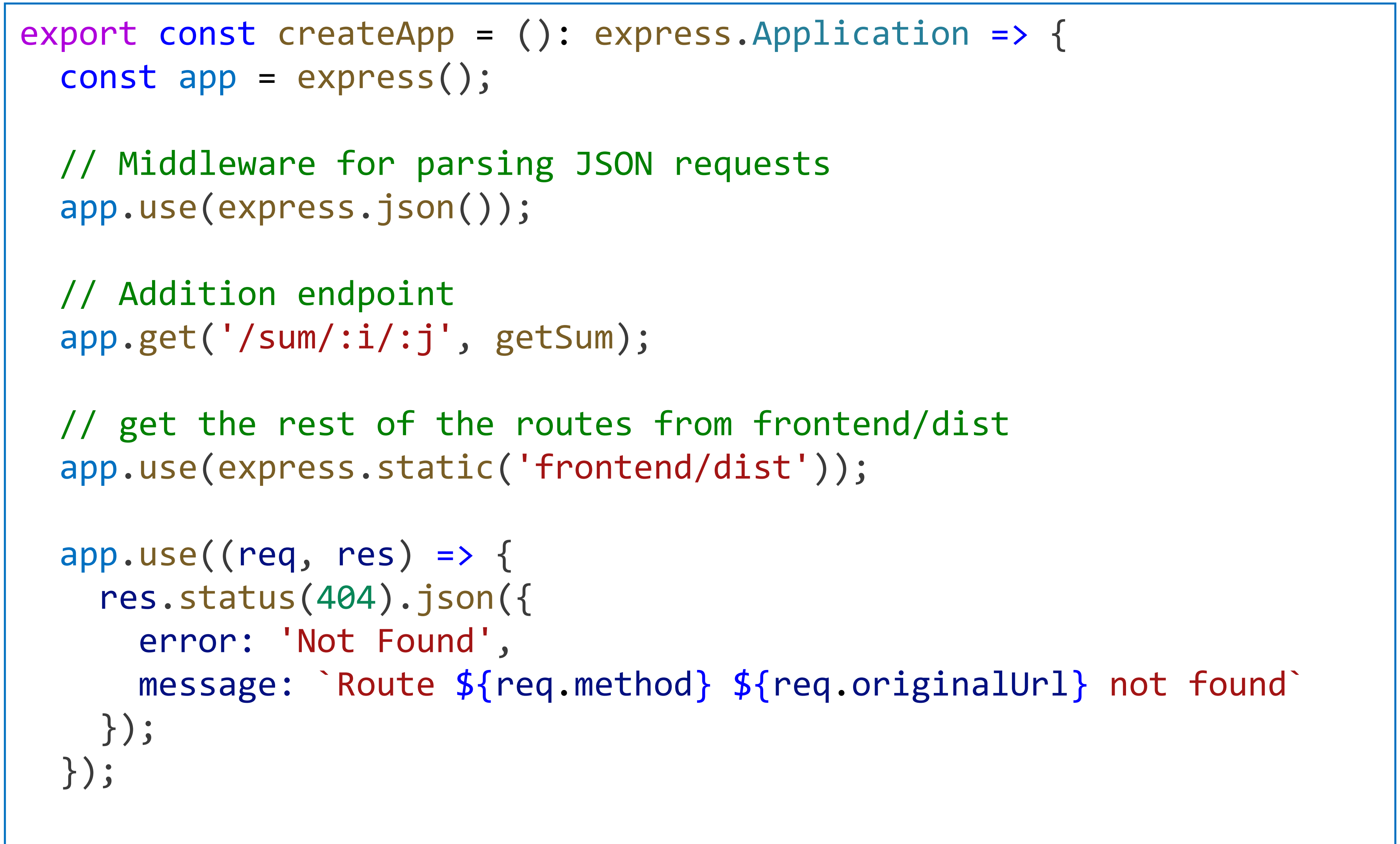
- **Activation Events**: when does your extension run?
- **Host API**: what procedures in the host app can your extension call?
- **Contribution Point**: what your extension contributes to the host (e.g. new commands, menus, pipeline stages, etc.)

Example 1: git hooks

- git provides a fixed set of activation events (files in .git/hooks)
- the user can extend git's default behavior by changing these files

```
$ cat .git/hooks/pre-merge-commit.sample
#!/bin/sh
#
# An example hook script to verify what is about to be committed.
# Called by "git merge" with no arguments.  The hook should
# exit with non-zero status after issuing an appropriate message to
# stderr if it wants to stop the merge commit.
#
# To enable this hook, rename this file to "pre-merge-commit".

. git-sh-setup
test -x "$GIT_DIR/hooks/pre-commit" &&
        exec "$GIT_DIR/hooks/pre-commit"
:
```

Example 2: express



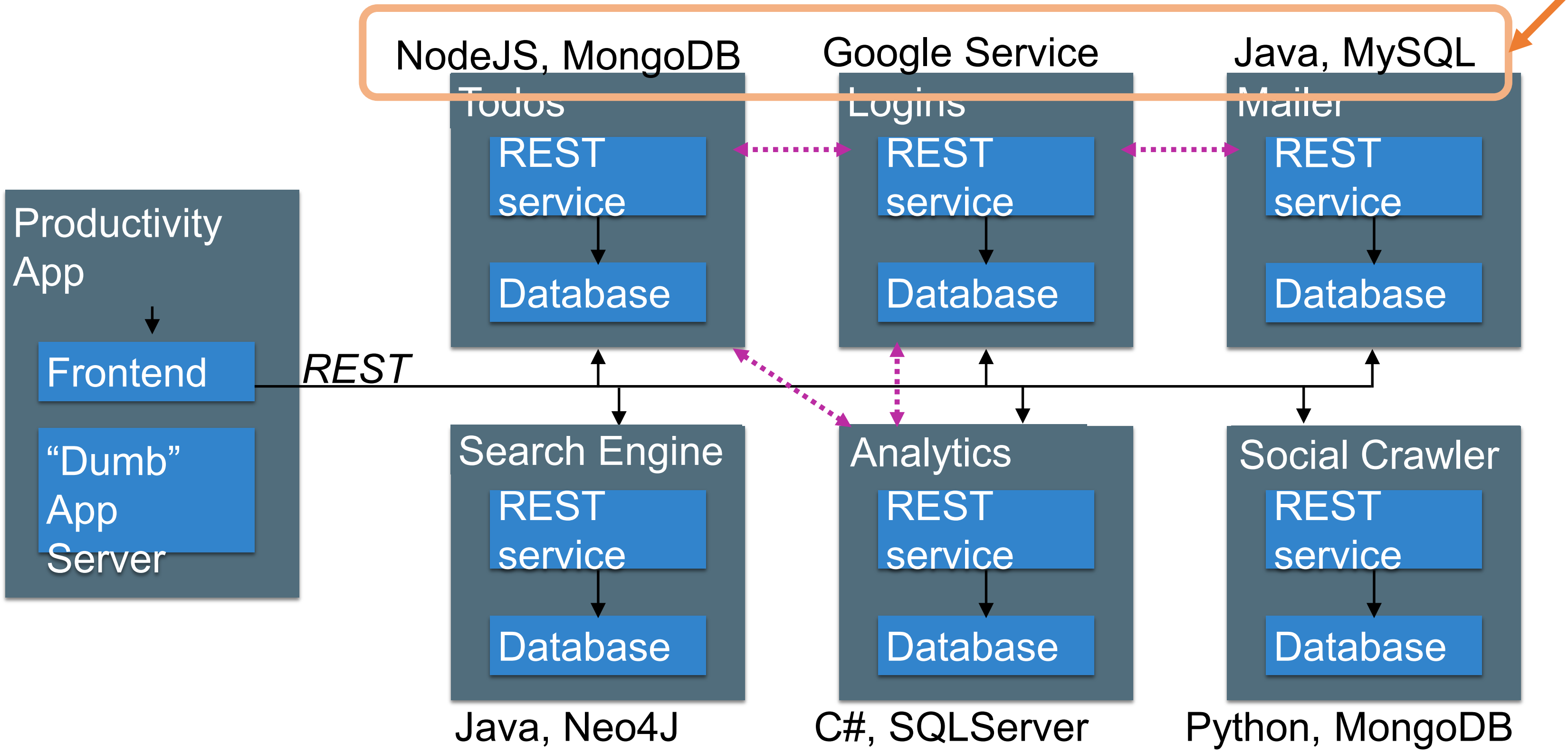
```
export const createApp = (): express.Application => {
  const app = express();

  // Middleware for parsing JSON requests
  app.use(express.json());

  // Addition endpoint
  app.get('/sum/:i/:j', getSum);

  // get the rest of the routes from frontend/dist
  app.use(express.static('frontend/dist'));

  app.use((req, res) => {
    res.status(404).json({
      error: 'Not Found',
      message: `Route ${req.method} ${req.originalUrl} not found`
    });
  });
```

Architecture #5: Microservices

- Overall task is divided into different components
- Each component is implemented independently
- Each component is
  - independently replaceable,
  - independently updatable
- Components can be built as libraries, but more usually as web services
  - Services communicate via HTTP, typically REST (see next lesson)

Microservices: Schematic Example

Different languages, different operating systems

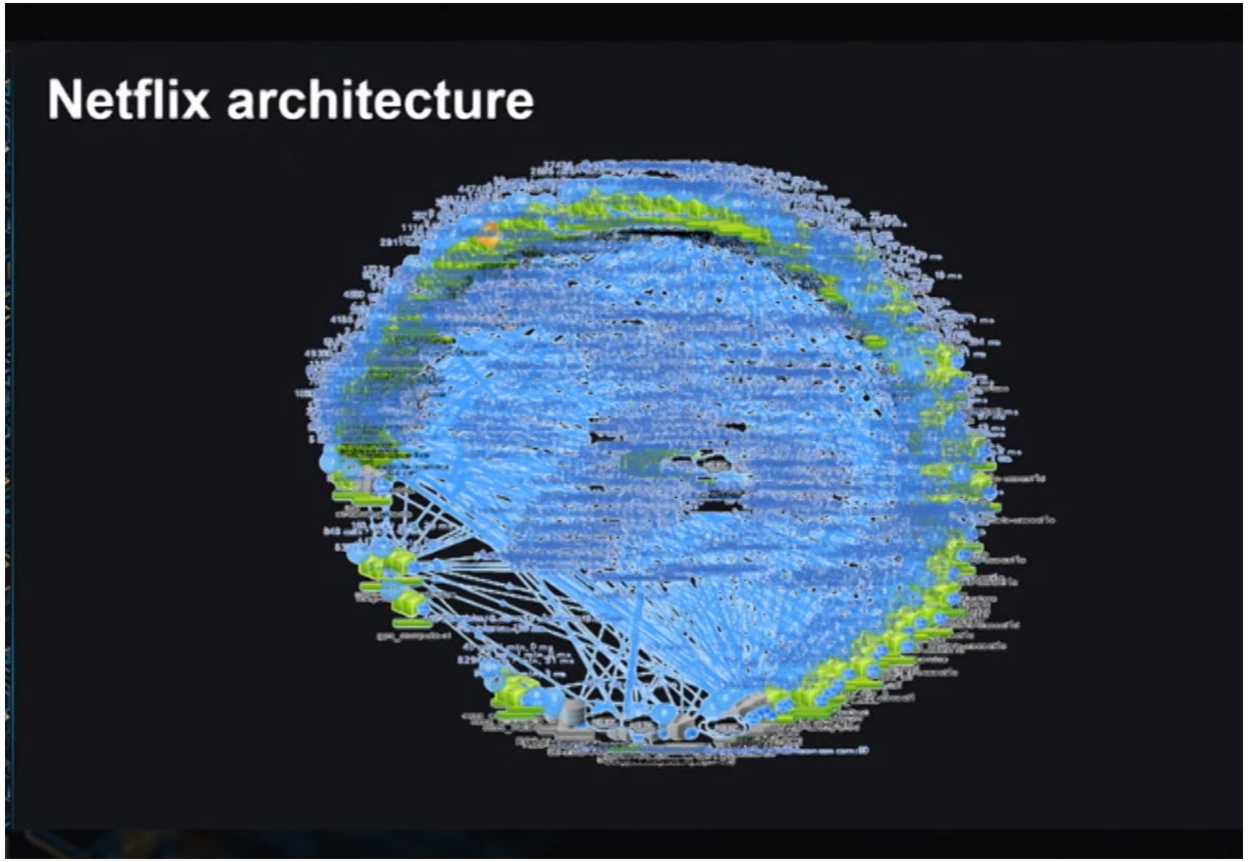


Microservice Advantages and Disadvantages

- Advantages
  - services may scale differently, so can be implemented on hardware appropriate for each (how much cpu, memory, disk, etc?). Ditto for software (OS, implementation language, etc.)
  - services are independent (yay for interfaces!) so can be developed and deployed independently
- Disadvantages
  - service discovery?
  - should services have some organization, or are they all equals?
  - overall system complexity

Microservices are (a) highly scalable and (b) trendy

- 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 (2017)

Lesson 10.2: Patterns of Communication

1. Procedure Calls (with callbacks)
2. HTTP
   - REST: a pattern for HTTP
3. Sockets

1. Procedure Calls

- Simplest
- Call + Return
- Call + Callback
- Only really works if both parties are in the same address space
- Best suited to layered architecture
- Less well-suited to pipeline (e.g. express)

2. HTTP

- Client-Server protocol
- Client sends a request, Server sends a response
- Can be used for Pull pattern
  - client requests data from server, server responds with data
  - "GET request"
- Can also be used for Push pattern
  - client sends local data to the server, server responds with acknowledgement
  - "POST request"

REST is a pattern for using HTTP

- Stands for "Representational State Transfer"
- Each request contains enough information that a different server could process it
- GET requests don't change server state
    - they are "idempotent"
- PUT requests are the ones that update the server state
    - not idempotent (eg "don't hit the PAY button more than once.")
- Uniform Interface - Standard way to specify interface

Uniform Interface: URIs are nouns

- In a RESTful system, the server is visualized as a store of named resources (nouns), each of which has some data associated with it.
- A URI is a name for such a resource.

Examples

- Examples:
  - `/cities/losangeles`
  - `/transcripts/00345/graduate` (student 00345 has several transcripts in the system; this is the graduate one)
- Non-examples:
  - `/getCity/losangeles`
  - `/getCitybyID/50654`
  - `/Cities.php?id=50654`

We prefer plural nouns for toplevel resources, as you see here.

Useful heuristic: if you were keeping this data in a bunch of files, what would the directory structure look like?
But you don't have to actually keep the data in that way.

Path parameters specify portions of the path to the resource

For example, your REST protocol might allow a path like

```
/transcripts/00345/graduate
```

In a REST protocol, this API might be described as

```
/transcripts/:studentid/graduate
```

`:studentid` is a path parameter, which is replaced by the value of the parameter

Query parameters allow named parameters

Examples:

- `/transcripts/graduate?lastname=covey&firstname=avery`

These are typically used to specify more flexible queries, or to embed information about the sender’s state, eg

- [`https://calendar.google.com/calendar/u/0/r/month/2023/2/1?tab=mc&pli=1`](https://calendar.google.com/calendar/u/0/r/month/2023/2/1?tab=mc&pli=1)

This URI combines path parameters for the month and date, and query parameters for the format (`tab` and `pli`).

You can also put parameters in the body.

- You can put additional parameters or information in the body, using any coding that you like. (We'll usually use JSON)
- You can also put parameters in the headers.
- Choose where to put parameters based on readability/copyability:
  - Path parameters provide a link to a resource
  - Query parameters modify how that resource is viewed/acted upon
  - Headers are transparent to users
  - Body parameters have unrestricted length

Uniform Interface:
Verbs are represented as http methods

- In REST, there are exactly four things you can do with a resource
- POST: requests that the server create a resource with a given value.
- GET: requests that the server respond with a representation of the resource
- (there are some others, but they are rarely used)

Example interface #1: a todo-list manager

- Resource: /todos
  - GET /todos - get list all of my todo items
  - POST /todos - create a new todo item (data in body; returns ID number of the new item)
- Resource: /todos/:todoItemID
  - :todoItemID is a path parameter
  - GET /todos/:todoItemID - fetch a single item by id
  - PUT /todos/:todoItemID - update a single item (new data in body)
  - DELETE /todos/:todoItemID - delete a single item

Example interface #2: the transcript database

```
POST /transcripts
 -- adds a new student to the database,
 -- returns an ID for this student.
 -- requires a body parameter 'name', url-encoded (eg name=avery)
 -- Multiple students may have the same name.
GET  /transcripts/:ID
 -- returns transcript for student with given ID.  Fails if no such student
DELETE /transcripts/:ID
 -- deletes transcript for student with the given ID, fails if no such student
POST /transcripts/:studentID/:courseNumber
 -- adds an entry in this student's transcript with given name and course.
 -- Requires a body parameter 'grade'.
 -- Fails if there is already an entry for this course in the student's transcript
GET  /transcripts/:studentID/:courseNumber
 -- returns the student's grade in the specified course.
 -- Fails if student or course is missing.
GET  /studentids?name=string
 -- returns list of IDs for student with the given name
```

Remember the heuristic: if you were keeping this data in a bunch of files, what would the directory structure look like?

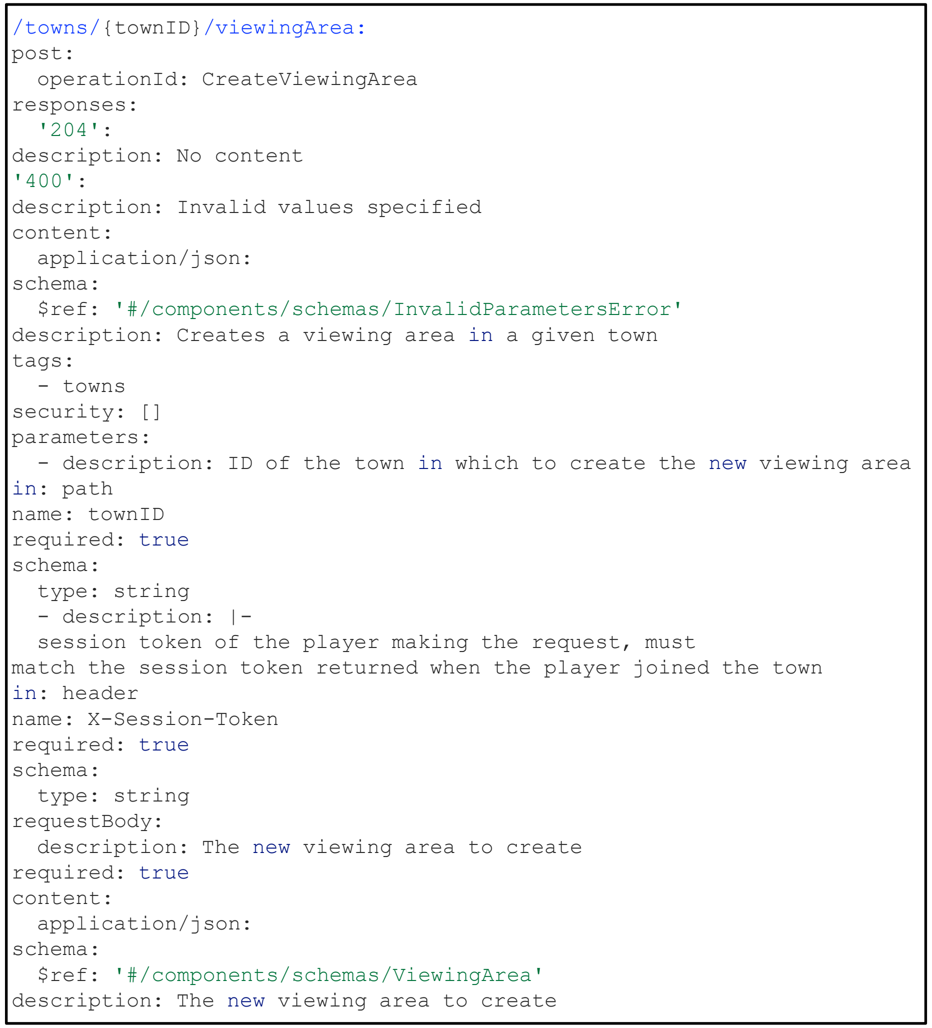
Didn't seem to fit the model, sorry

It would be better to have a machine-readable specification

- The specification of the transcript API on the last slide is RESTful, but is not machine-readable
- A machine-readable specification is useful for:
  - Automatically generating client and server boilerplate, documentation, examples
  - Tracking how an API evolves over time
  - Ensuring that there are no misunderstandings

OpenAPI is a machine-readable specification language for REST

- Uses YAML syntax
- Not really convenient for human use
- Better: use a tool!



```
/towns/{townID}/viewingArea:
post:
  operationId: CreateViewingArea
responses:
  '204':
description: No content
'400':
description: Invalid values specified
content:
  application/json:
schema:
  $ref: '#/components/schemas/InvalidParametersError'
description: Creates a viewing area in a given town
tags:
  - towns
security: []
parameters:
  - description: ID of the town in which to create the new viewing area
in: path
name: townID
required: true
schema:
  type: string
  - description: |-
  session token of the player making the request, must
match the session token returned when the player joined the town
in: header
name: X-Session-Token
required: true
schema:
  type: string
requestBody:
  description: The new viewing area to create
required: true
content:
  application/json:
schema:
  $ref: '#/components/schemas/ViewingArea'
description: The new viewing area to create
```

Tools for making these protocols machine-readable

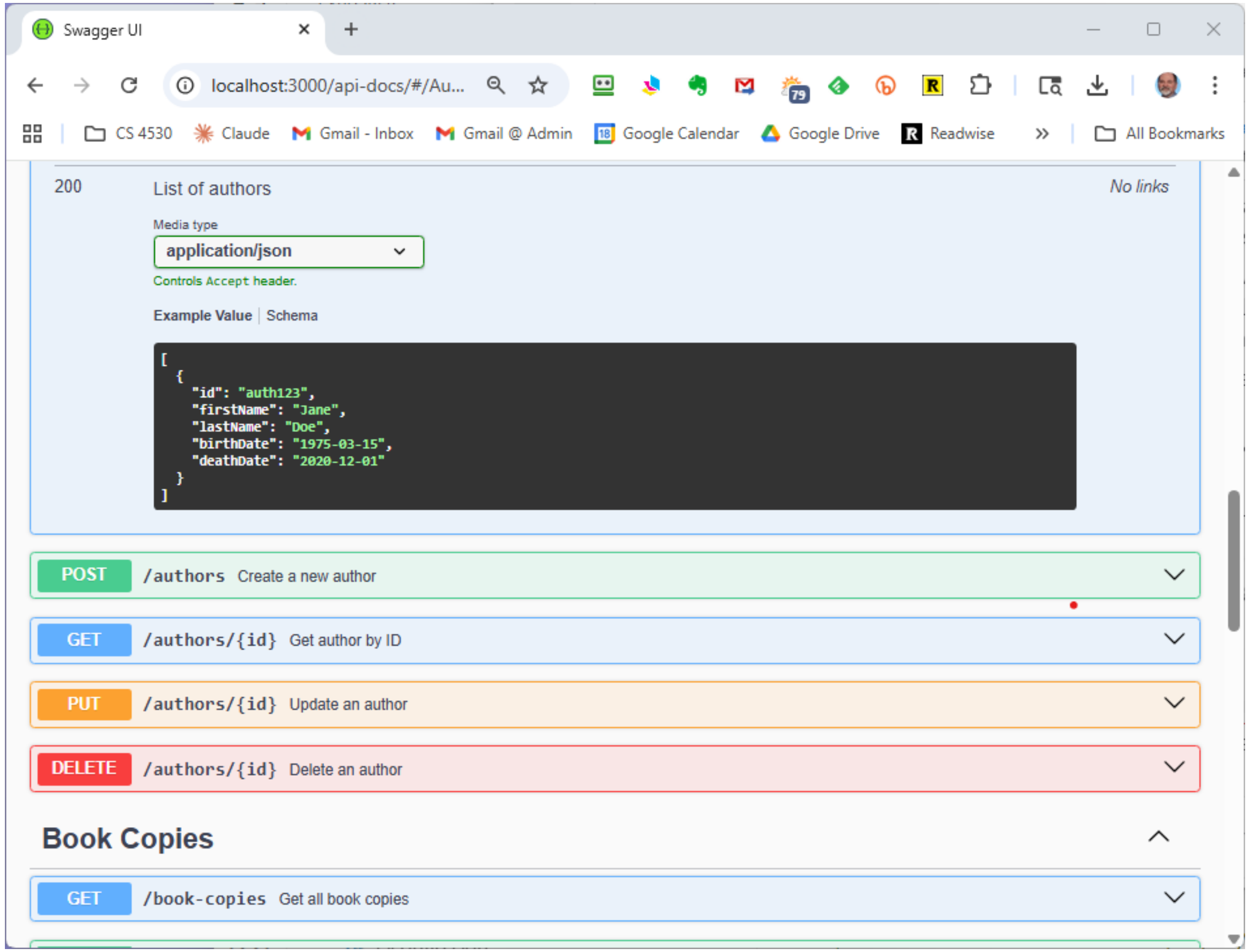
- TSOA
  - reads an annotated controller file
  - but only works with particular controller structures
- Swagger
  - human-annotated controller and route files
  - generates nice docs
  - but still requires human attention to ensure consistency, validation

Swagger example (in a routes file)

```
/**
 * @swagger
 * /authors/{id}:
 *   put:
 *     summary: Update an author
 *     tags: [Authors]
 *     parameters:
 *       - in: path
 *         name: id
 *         required: true
 *         schema:
 *           type: string

 ======= many lines omitted ======

 *     responses:
 *       200:
 *         description: Author updated successfully
 *         content:
 *           application/json:
 *             schema:
 *               $ref: '#/components/schemas/Author'
 *       404:
 *         description: Author not found
 */
router.put('/:id', updateAuthorById);
```

Detailed description of the request, in YAML, human-written

Swagger-generated documentation



200 List of authors *No links*

Media type: application/json

Controls Accept header.

Example Value | Schema

```
[
  {
    "id": "auth123",
    "firstName": "Jane",
    "lastName": "Doe",
    "birthDate": "1975-03-15",
    "deathDate": "2020-12-01"
  }
]
```

POST `/authors` Create a new author

GET `/authors/{id}` Get author by ID

PUT `/authors/{id}` Update an author

DELETE `/authors/{id}` Delete an author

Book Copies

GET `/book-copies` Get all book copies

Can also run queries right from this page!

But we'd like to do better

- No guarantee the human-written descriptions are accurate!
  - Correct extraction of data from a request?
  - Automatic validation?
- There are tools for this, too
  - swagger-codegen
  - OpenAPI Generator
  - …and others

3. Websockets

- Server-Client
    - We saw this earlier in Module 05.
    - Client talks only to server
    - Server can talk to a single client or to subsets of the clients
    - Either side can initiate a conversation
    - Allows for more complex protocols
        - like we saw in Module05
    - NOT query-response
        - though a particular protocol may do some of this
    - But generally less scalable than HTTP.

Review

- You should now be able to:
  - describe the basic ideas of the following architectures, with examples and pictures
    - anarchic
    - layered
    - pipeline
    - event-driven
    - microkernel
    - microservice
  - describe the main features of the following communication modalities:
    - procedure calls
    - HTTP and REST
    - Websockets