Microservices

SWE 432, Fall 2017

Design and Implementation of Software for the Web

Today

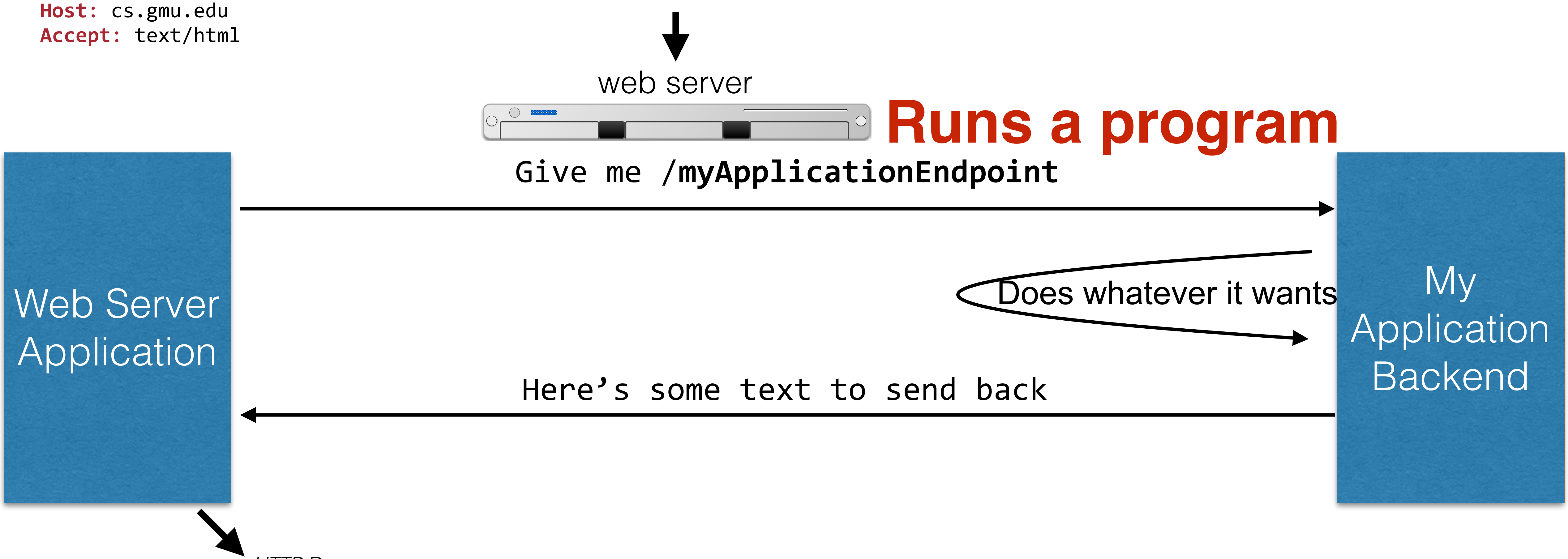
- How is a being a microservice different than simply being RESTful?
- What are the advantages of a microservice backend architecture over a monolithic architecture?

- Next time: what additional infrastructure is required to realize these advantages?

The “good” old days of backends

HTTP Request

```
GET /myApplicationEndpoint HTTP/1.1
Host: cs.gmu.edu
Accept: text/html
```



HTTP Response

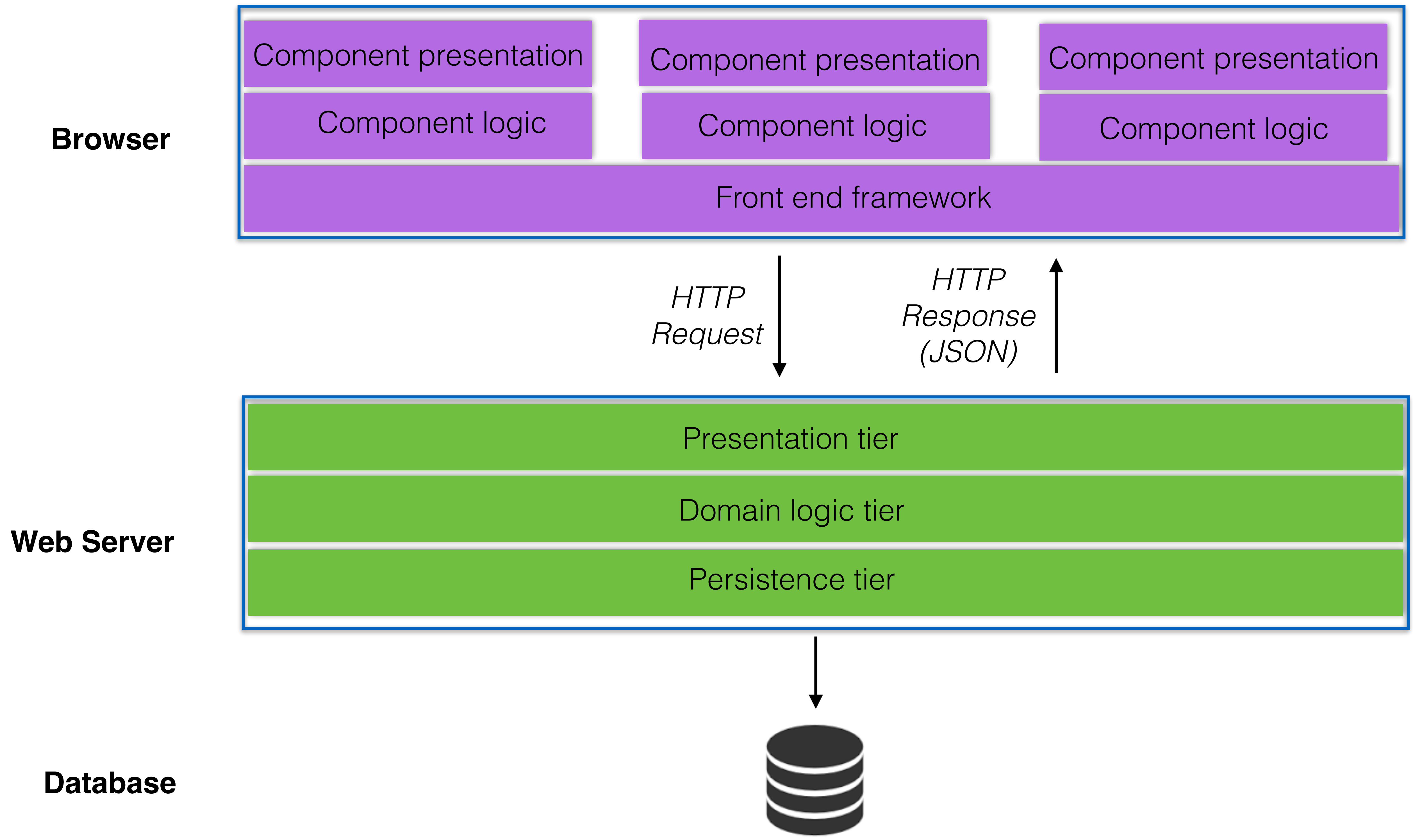
```
HTTP/1.1 200 OK
Content-Type: text/html; charset=UTF-8

<html><head>...
```

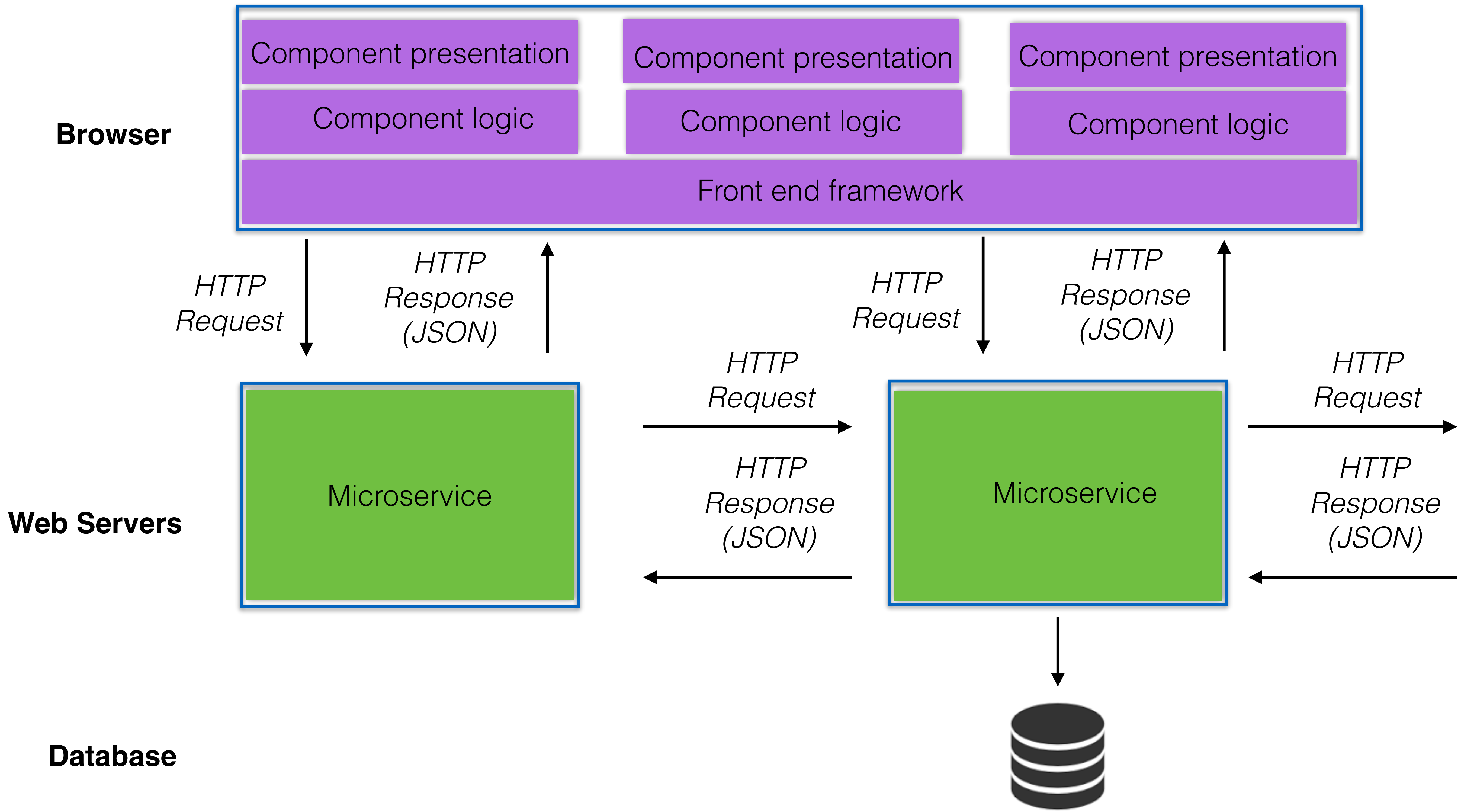
History of Backend Development

- In the beginning, you wrote whatever you wanted using whatever language you wanted and whatever framework you wanted
- Then… PHP and ASP
  - Languages “designed” for writing backends
  - Encouraged spaghetti code
  - A lot of the web was built on this
- A whole lot of other languages were also springing up in the 90’s…
  - Ruby, Python, JSP

Monolothic backend



Microservices backend



RESTful APIs

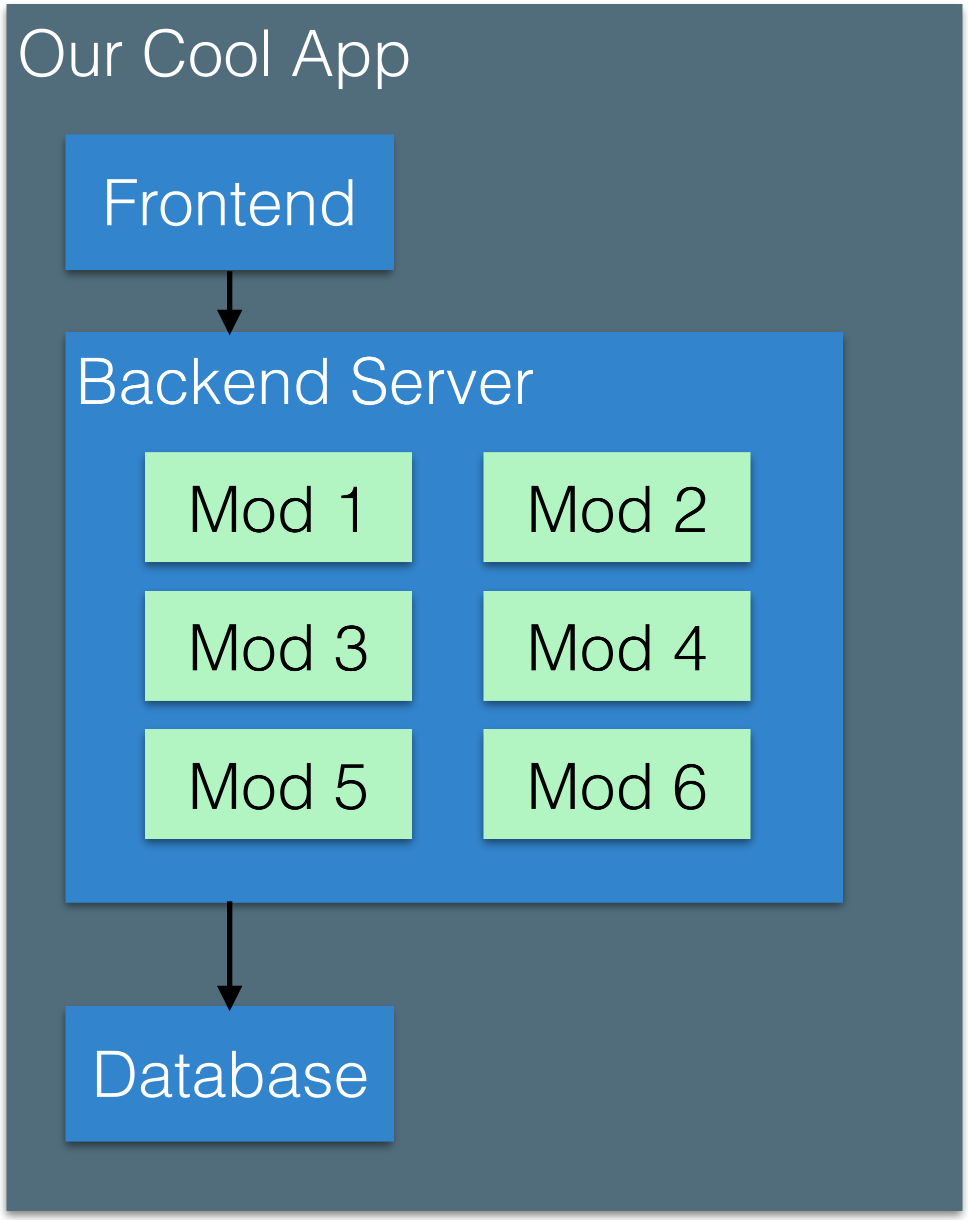
- Recall guidelines for RESTful APIs from Lecture 6: Handling HTTP Requests
- Support scaling
  - Use HTTP actions to support intermediaries (e.g., caches)
- Support change
  - Leave anything out of URI that might change
  - Ensure any URI changes are backwards compatible
- Support reuse
  - Design URIs around resources that are expressive abstractions that support a range of client interactions
  - Resources are nouns; use HTTP actions to signal verbs

Challenges building a RESTful monolith

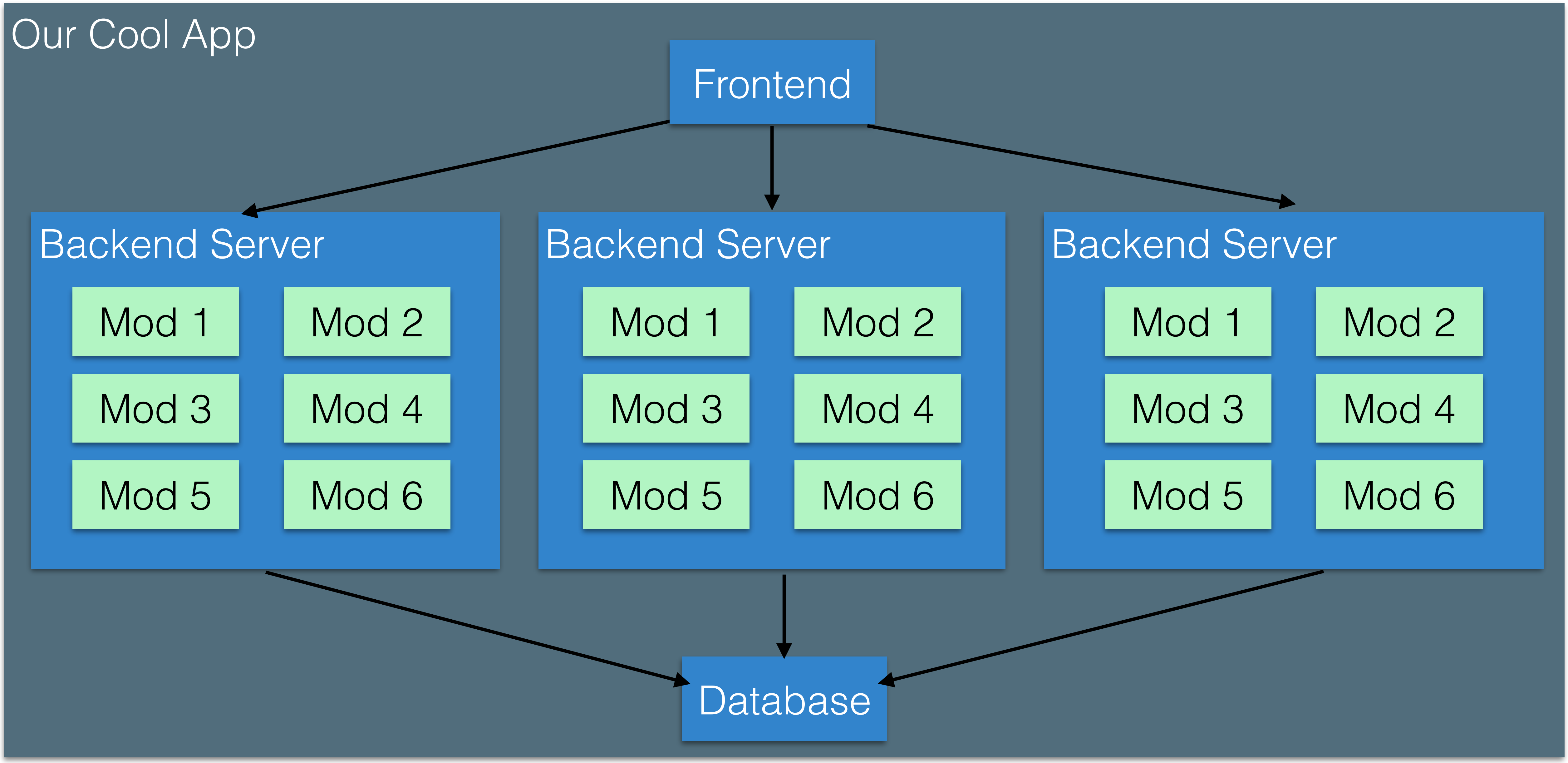
Microservices vs. Monoliths

- Advantages of microservices over monoliths include
  - Support for scaling
    - Scale vertically rather than horizontally
  - Support for change
    - Support hot deployment of updates
  - Support for reuse
    - Use same web service in multiple apps
    - Swap out internally developed web service for externally developed web service
  - Support for separate team development
    - Pick boundaries that match team responsibilities
  - Support for failure

Support for scaling



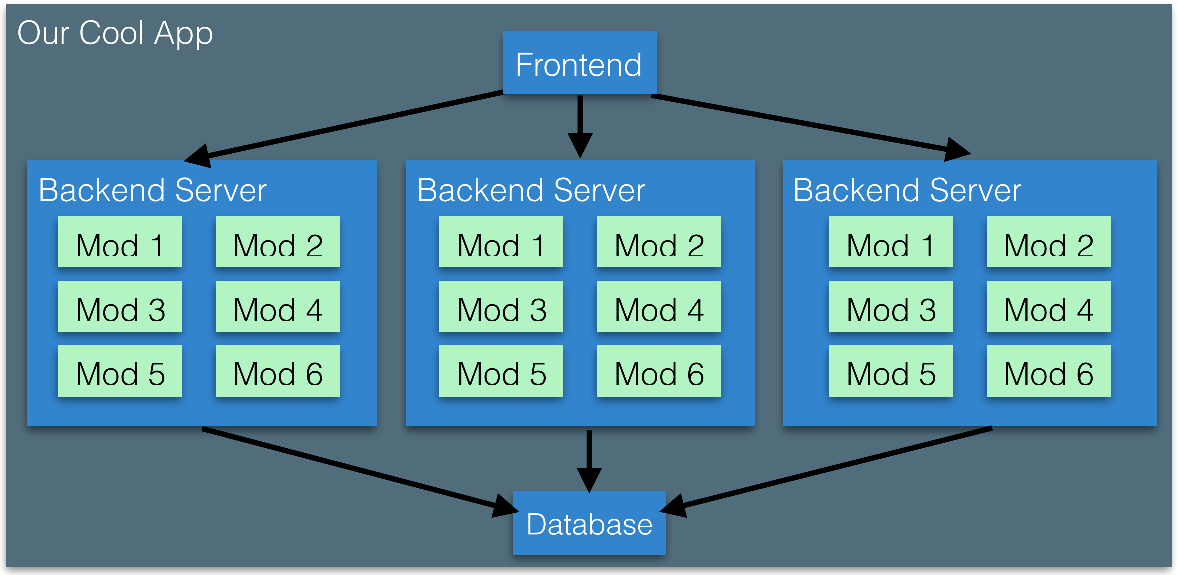
Now how do we scale it?



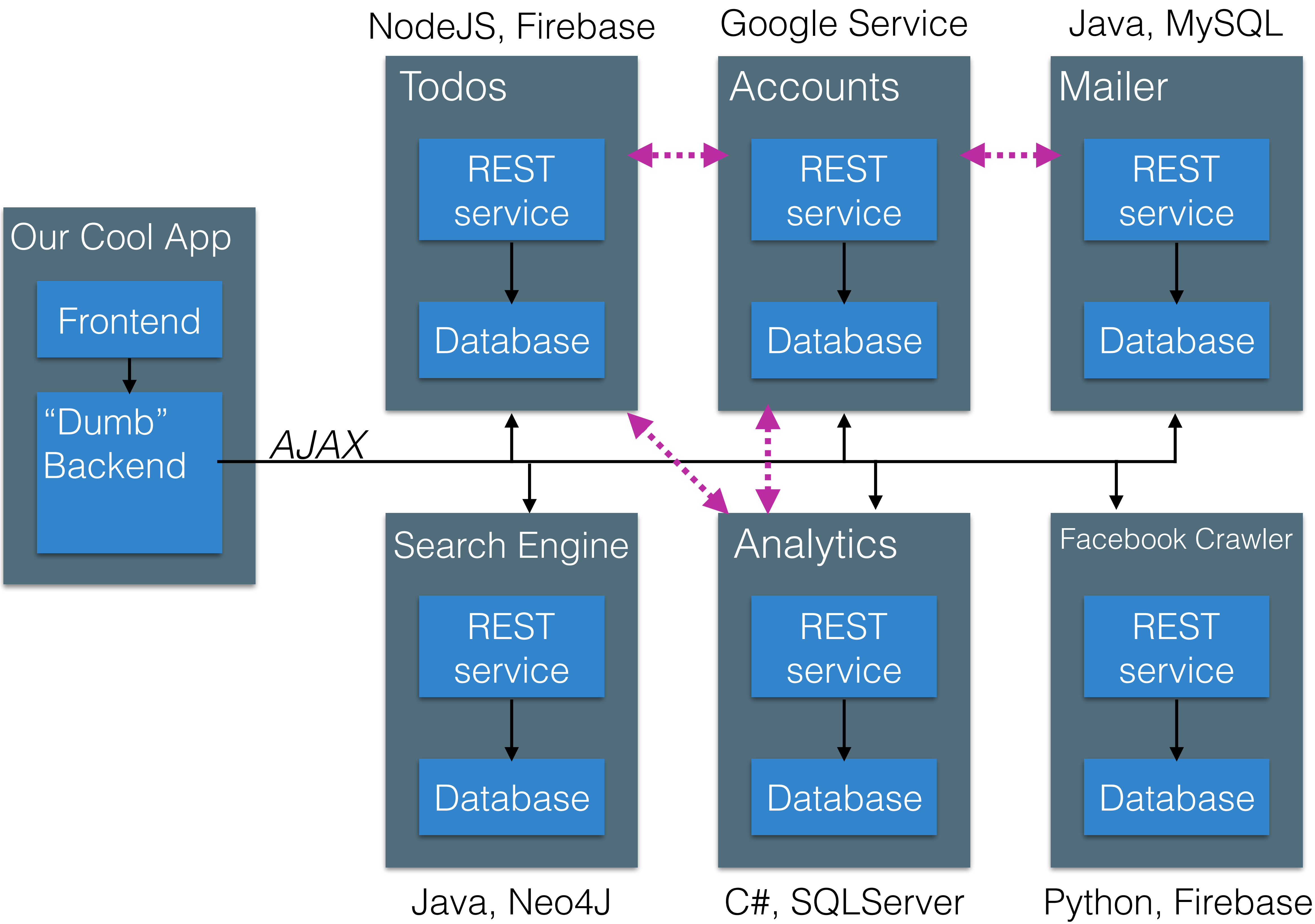
We run multiple copies of the backend, each with each of the modules

What's wrong with this picture?

- This is called the "monolithic" app
- If we need 100 servers…
- Each server will have to run EACH module
- What if we need more of some modules than others?



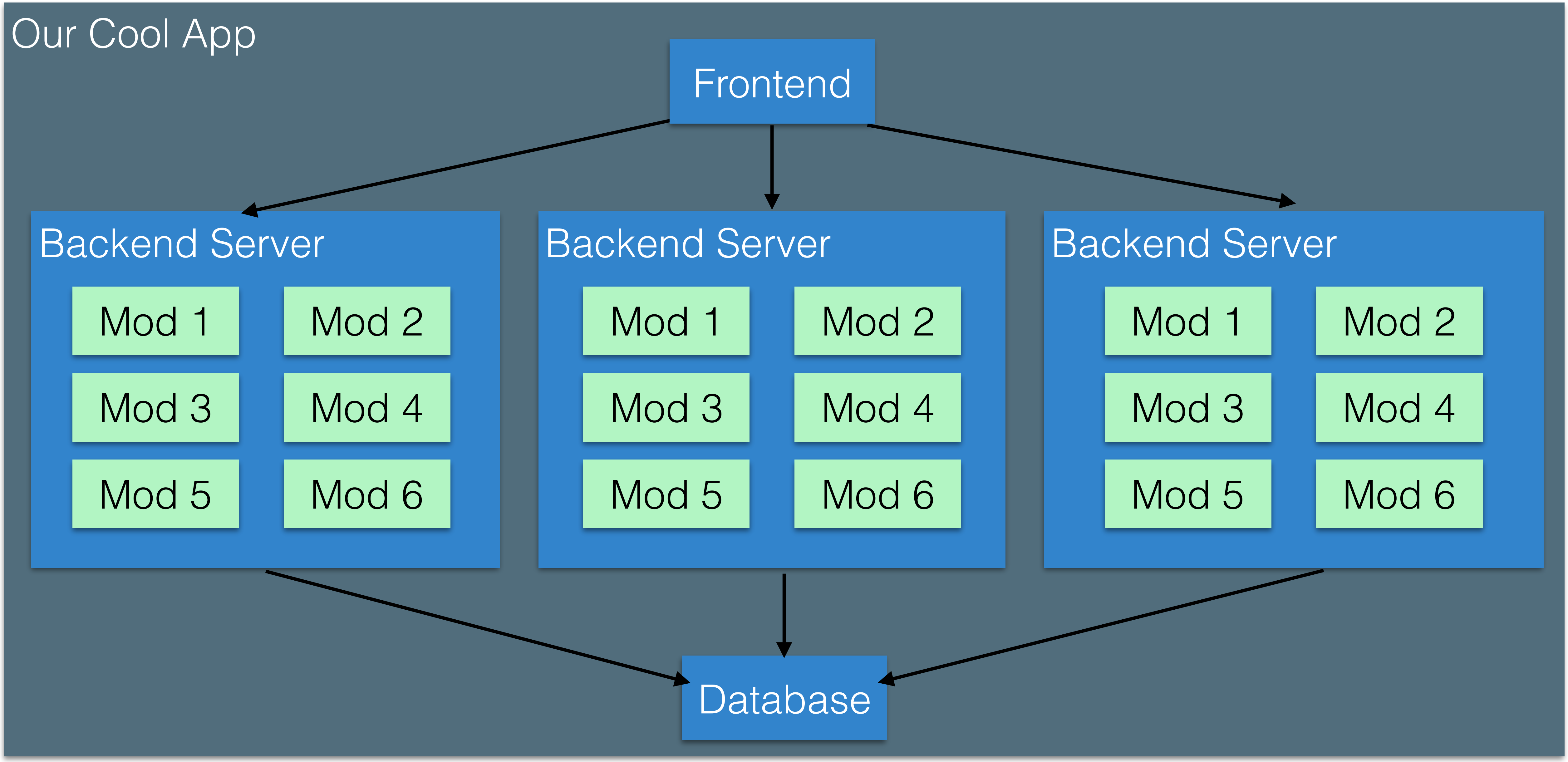
Microservices



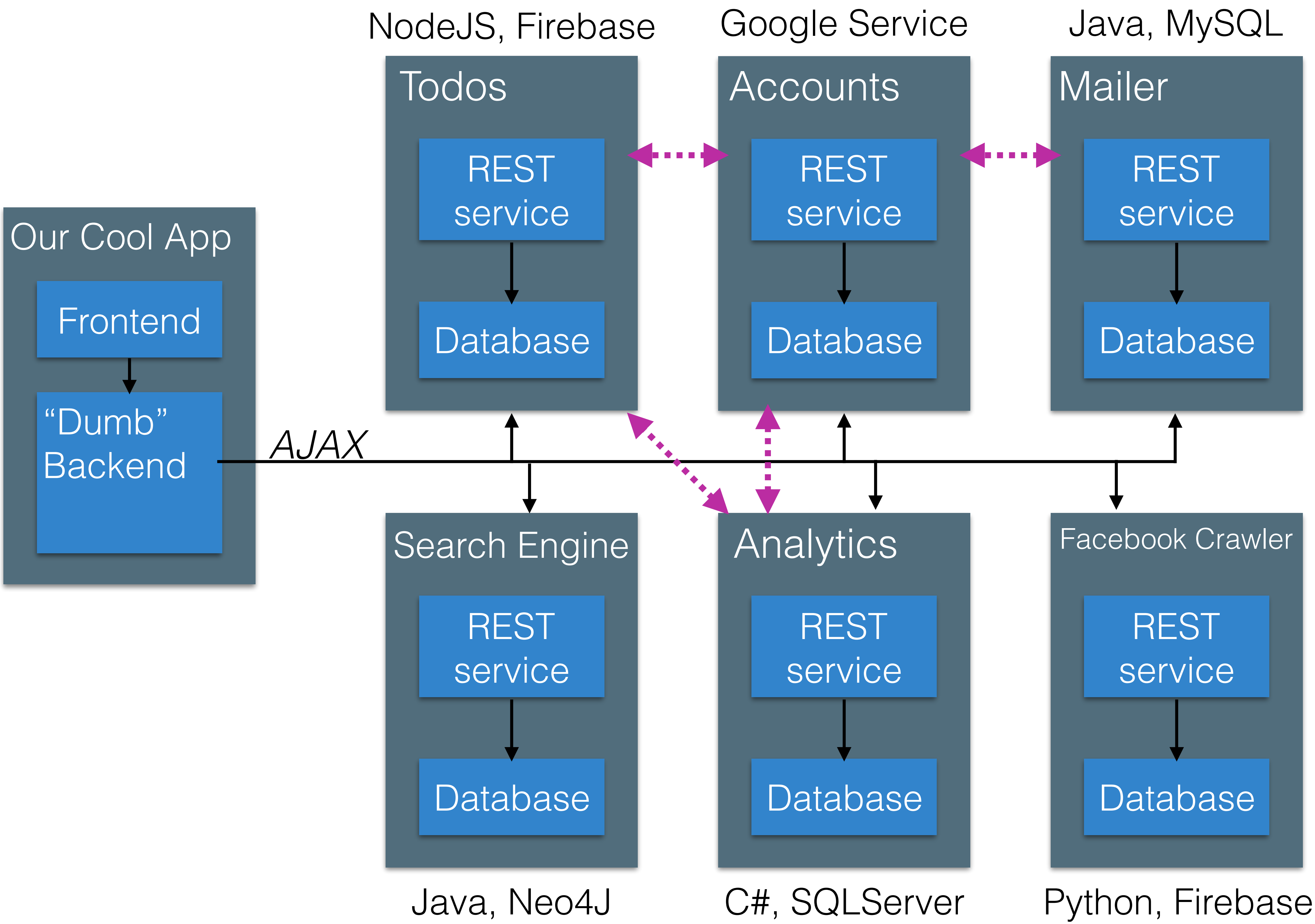
Support for change: hot swapping

- In a large organization (e.g., Facebook, Amazon, AirBnb), will constantly have new features being finished and rolled out to production
- Traditional model: releases
  - Finish next version of software, test, release as a unit once every year or two
- Web enables frequent updates
  - Could update every night or even every hour
- But.... if updating every hour, really do not want website to be down

Support for change in a monolith



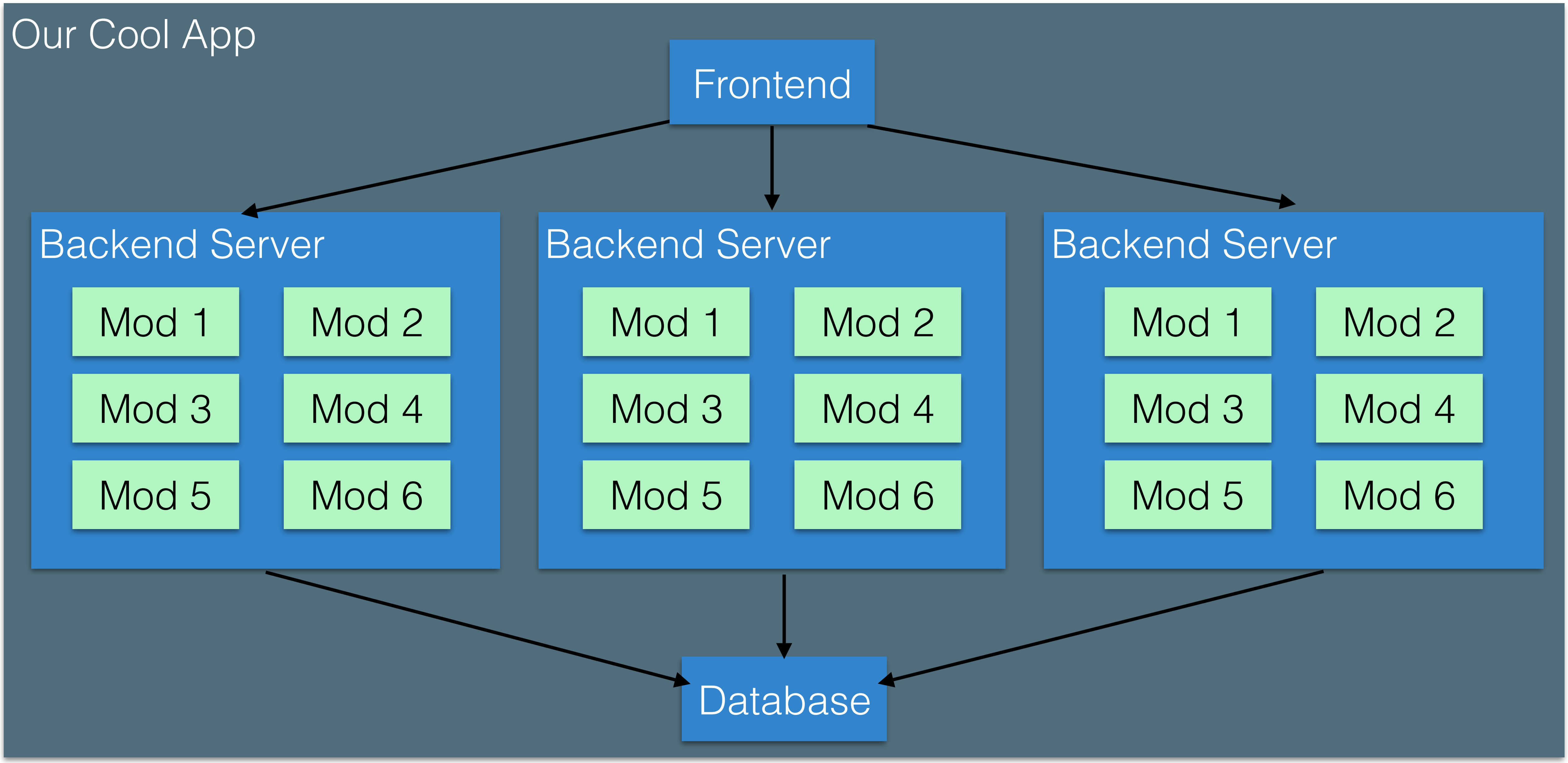
Microservices



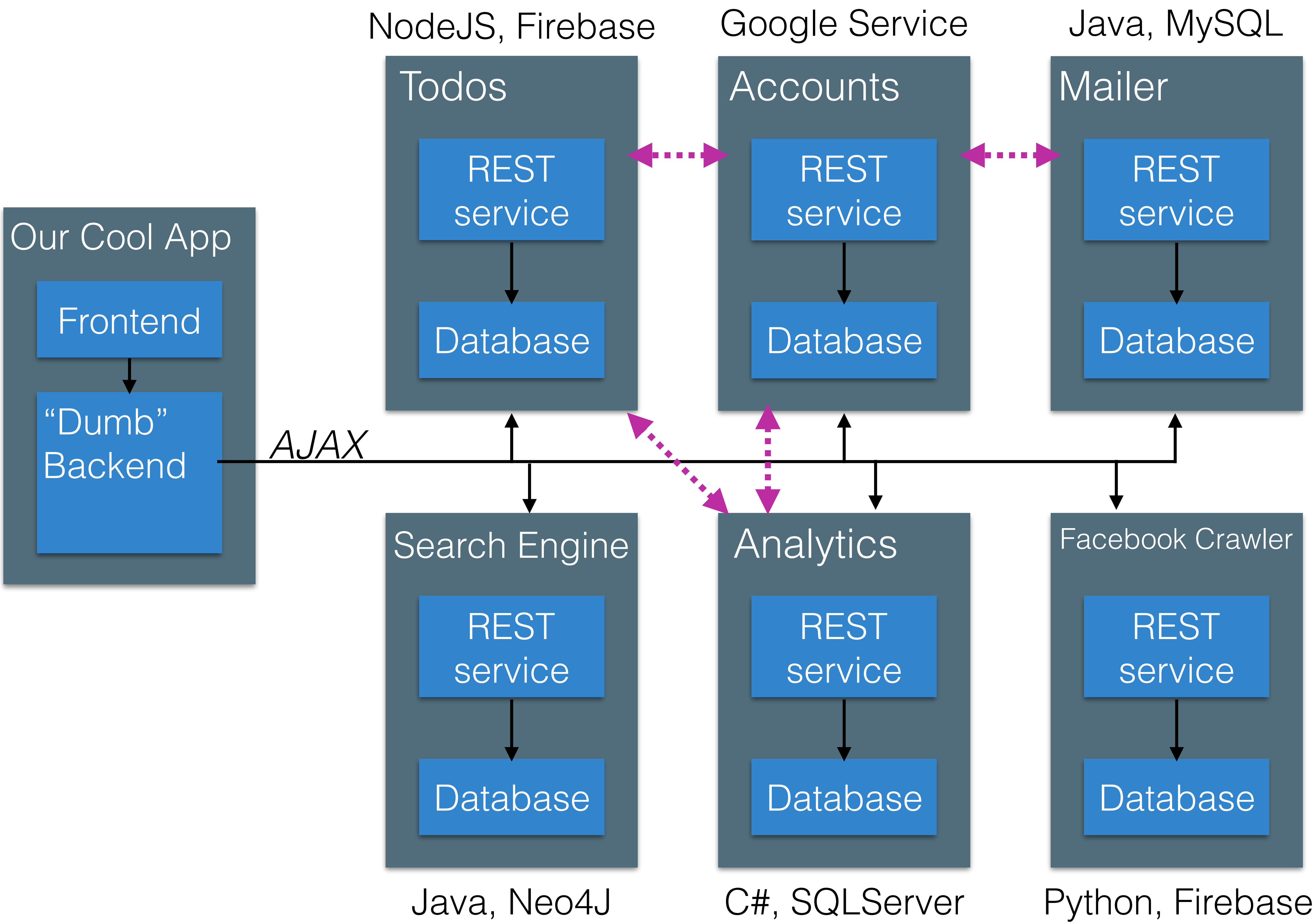
Support for reuse

- In a large organization (e.g., Facebook, Amazon, AirBnb), may have many internal products that all depend on a similar core service (e.g., user account storage, serving static assets)
- Would like to
  - be able to build functionality once, reuse in many place
  - swap out an old implementation for a new implementation with a new technology or implementation
  - swap out an internal service for a similar external service

Support for reuse in a monolith



Microservices



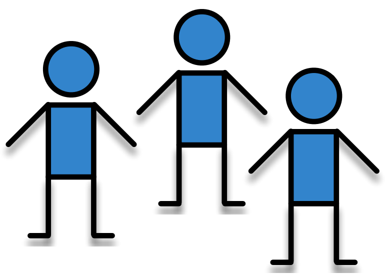
Conway's Law

- The structure of an organization mirrors the structure of a product.

- Building a car.
  - Have a team for tires
  - Have a team for drivetrain
  - Have a team for seating
  - Have a team for paint
  - Have a team for ...

- Could pick a product structure and design **team** around it.
- Or could pick a desired team structure and design **product** around it.

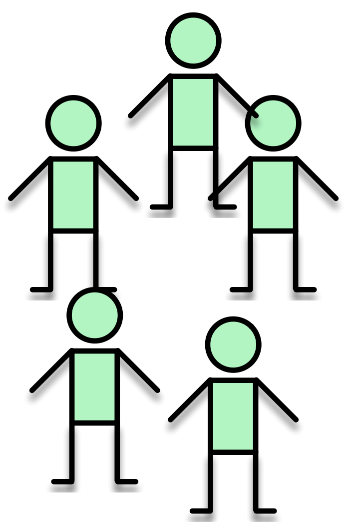
Organization in a monolith



Frontend

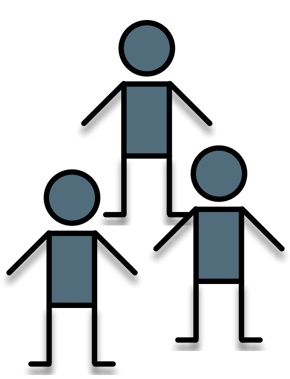
Orders, shipping, catalog

Classic teams:
1 team per "tier"



Backend

Orders, shipping, catalog



Database

Orders, shipping, catalog

Organization around business capabilities in microservices

Example: Amazon

Teams can focus on one business task
And be responsible directly to users

“Full Stack”

“2 pizza teams”

How big is a microservice?

- Metaphor: Building a stereo system
- Components are independently replaceable
- Components are independently updatable
- This means that they can be also independently developed, tested, etc
- Components can be built as:
  - Library (e.g. module)
  - Service (e.g. web service)

Goals of microservices

- Add them independently
- Upgrade the independently
- Reuse them independently
- Develop them independently

- ==> Have ZERO coupling between microservices, aside from their shared interface

Exercise: Design a restaurant review site

- In groups of 2 or 3, build diagram depicting a set of microservices, their connections, and a list of important endpoints

- Requirements
  - Restaurant owners can create restaurant pages, add links to website, add food keywords, update address and business info
  - Restaurant reviewers can post reviews of a restaurant, see reviews they've written, comment on other reviews.
  - All users can search for a restaurant based on its food keywords and address.
  - Users have accounts, with profile information and settings.

Design for Failure

- Each of the many microservices might **fail**
  - Services might have bugs
  - Services might be slow to respond
  - Entire servers might go down
    - If I have 60,000 hard disks, 3 fail a day
  - The more microservices there are, the higher the likelihood at least one is currently failing
- Key: design every service assuming that at some point, everything it depends on might disappear - must fail “gracefully”
- Netflix simulates this constantly with “ChaosMonkey”

Support for failure

- Goal: Support graceful degradation with service failures
- Design for idempotency
  - Should be able to retry requests without introducing bad data
- Design for data locality
  - Transactions across microservices are hard to manage
- Design for eventual consistency

Design for idempotency

- Want to design APIs so that executing an action multiple times leads to same resulting state
- Prefer state changes on existing entity rather than creating new entities

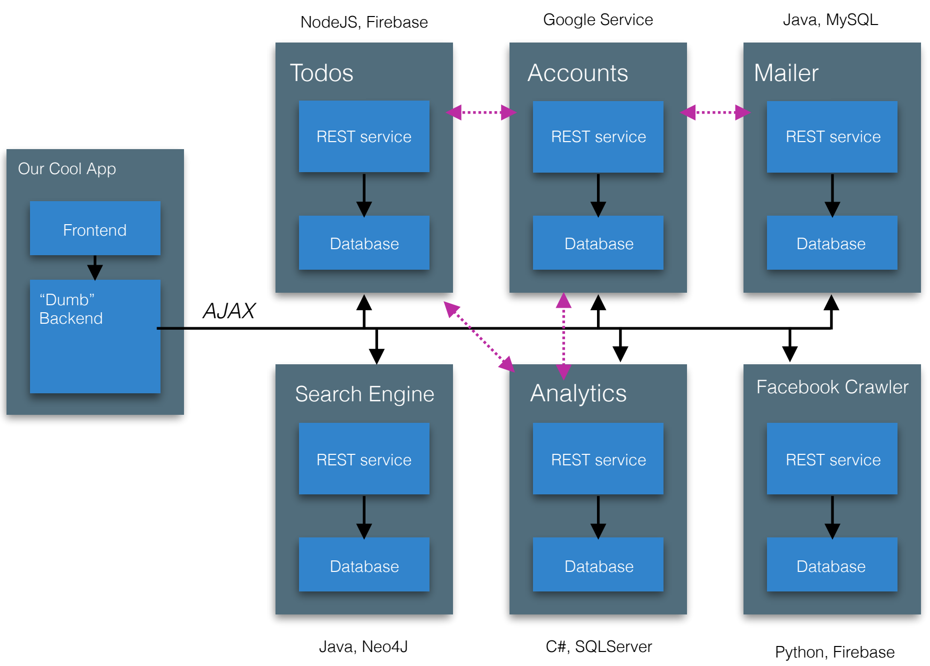
Design for data locality

- If datastore server fails or is slow, do not want entire site to go down.
- Decentralizes implementation decisions.
- Allows each service to manage data in the way that makes the most sense for that service
- Also performance benefit: caching data locally in microservices enables faster response

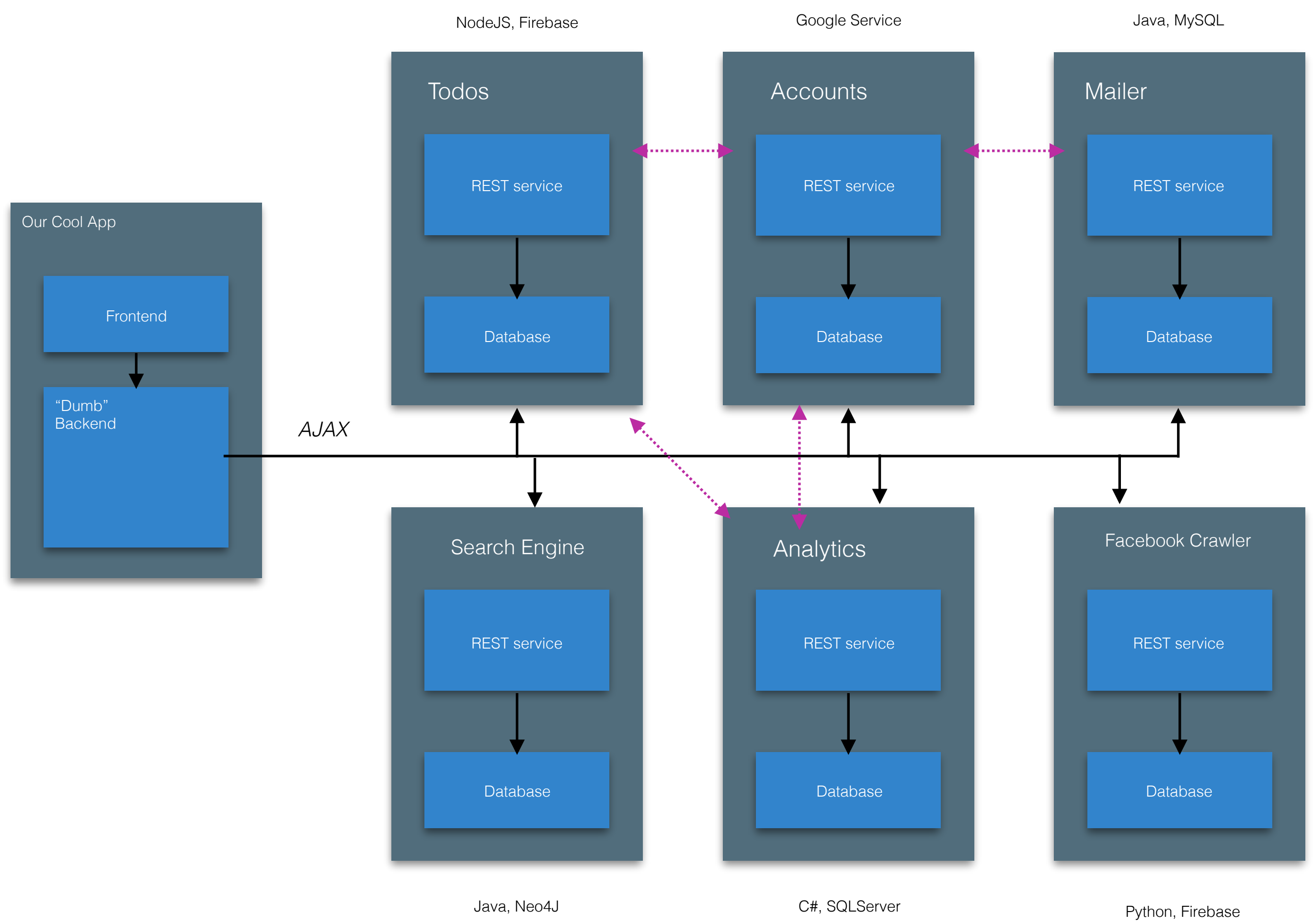
- Rule: Services exchange data ONLY through their exposed APIs - NO shared databases

Consistency

- One of our rules was “no shared database”
- But surely some state will be shared
- Updates are sent via HTTP request
- No guarantee that those updates occur immediately
- Instead, guarantee that they occur ***eventually***
- Can force some ordering, but that’s expensive



Maintaining Consistency



- Core problem: different services may respond to requests at **different times**.
  - What if a request results in change to resource in one service, but other service has not yet processed corresponding request?
  - May end up with different states in different resources.
  - Logic needs to be written to correctly handle such situations.

Eventual Consistency: Example

Reading for next time

- Fundamentals of DevOps:
  - https://blogs.oracle.com/developers/getting-started-with-microservices-part-four