

# CS188

# Scalable Internet Services

---

John Rothfels, 10/20/20

# Announcements

- You should be writing features with your project group
  - Load testing starts in 2 weeks.
- Follow [this guide](#) for getting help if you're stuck
- Use TA office hours for programming help

# Project

- Your project features are a means to an end! Our goal is to spend time **debugging scaling problems.**
- Your project does not need to be scalable (or even perfectly functional)!
  - In fact, the more unscalable it is (initially) the better.
- Your grade depends on your **analysis of the scaling problems**, and **what you do to try to fix them**, and your **analysis of the effects.**
  - The more scaling techniques you try to apply the better.
- Treat your project like an experiment. Change something, measure the effect. Change something else, measure the effect and see if it was better/worse than before. Etc.

# Grading

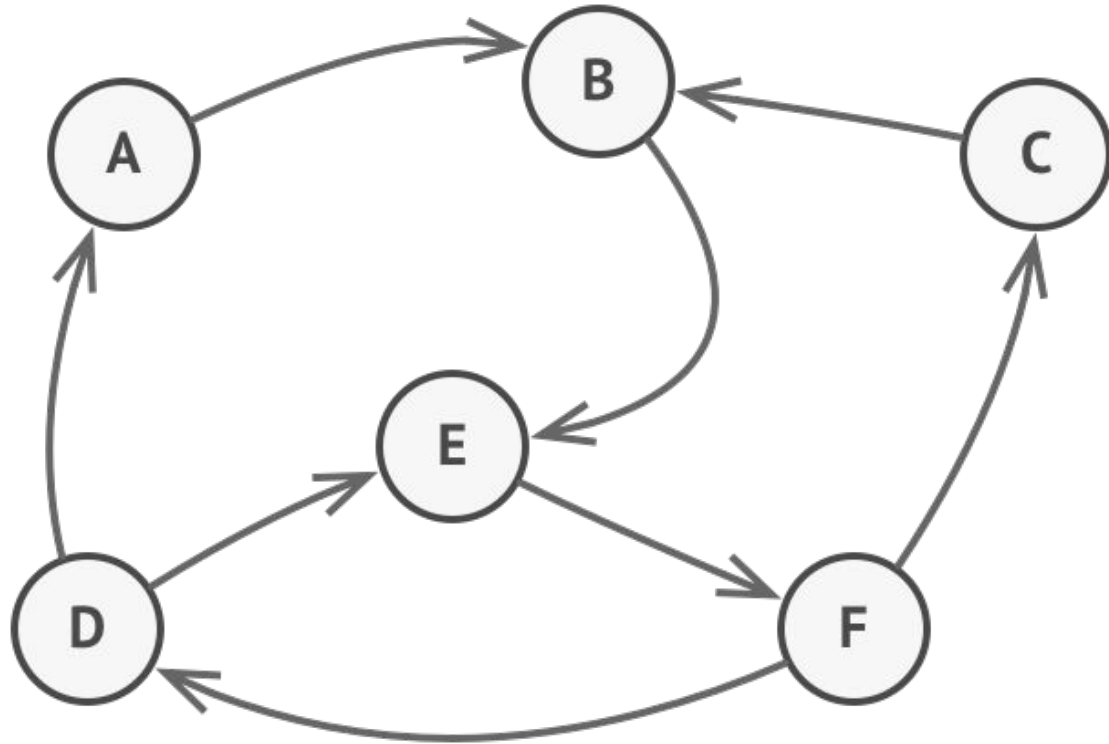
To get an **A** in the class, you will need:

- A project with functional features
- Accurate measurement & analysis of initial scaling problems
- Several techniques from the course attempt to fix those problems
- Accurate measurement & analysis of the techniques you apply
- Group paper/presentation & peer review

If you are on the edge of a letter grade, I will factor in:

- Participation in lecture, labs, Piazza
- Progress milestones from labs (GitHub commits to your project)

# Motivation



# State management

The most challenging scaling problems come from **managing the state of your application**. We will spend the next several lectures discussing state management:

- Relational databases & ORMs
- Non-relational databases
- Caching

# State management

! ? What is **application state**?

# State management

! ? What is **application state**?

State is the data required to run (or display) your application.

- *changes over time* (usually while your application is running)
- Often persisted (saved) to some storage medium (e.g. a file on a hard disk, a row in a database, a JSON in your browser local storage)
- changes due to user interaction or other business rules (e.g. background process)



# State management

! ? What does state look like in the browser?

The root route `/` of your application is requested:

- the browser passes a **cookie** (key/value pair) if you are logged in
  - the cookie is a reference to state living on the server (“I am user X”)
- the browser receives HTML with data
  - the data may be “static” (not stateful) or “dynamic”ally fetched from an API, database, ...
- the browser executes JavaScript on the page
  - code may fetch new data from your server, save stuff in data structures, re-draw HTML ...

# React

**React** is a JavaScript library that makes drawing HTML easier. How?

- **JSX**: “extended” JavaScript syntax that lets you write code that looks like HTML in JavaScript
- **MVC**: uses model view controller program design
- **Components**: simple abstraction for decomposing & organizing UI code and behavior

# React: JSX

You enter a special context called JSX (or TSX) when you start a `<tag>`.

You can put JavaScript (or TypeScript) in your JSX/TSX by wrapping `{ }`.

```
export function UserList(props: { users: { email: string }[] }) {  
  return (  
    <div>  
      {props.users.map(u => (  
        <div key={u.email}>{u.email}</div>  
      ))}  
    </div>  
  )  
}
```

 Putting lists into JSX requires giving each element a unique **key**.

# React: MVC

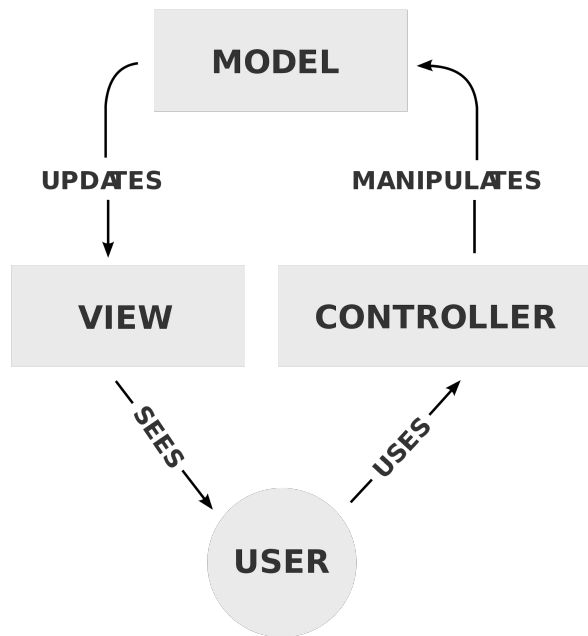
**Model-View-Controller** is a software design pattern commonly used for user interfaces.

## Pros:

- Simple
- Ubiquitous
- Separation of concerns

## Cons:

- Not all MVCs are created equal



# React: Components

React implements the MVC pattern with **components**. A component is JavaScript function that draws HTML.




props are received from a parent component and are **read only**

# React: Components

The beauty of React's component-based programming model is that it provides a simple abstraction:

```
view = function(props)
```

 A “pure” component is a pure function that draws the same HTML for a provided **props** no matter how many times you call it or when.

 How do React/components help you manage state?

# React: State

In React, components can have internal **state** that may change while the component is “mounted” (rendered / displayed):

```
function Counter() {  
  const [count, setCount] = React.useState(0)  
  return <button onClick={() => setCount(count + 1)}>{count}</button>  
}
```

**count** is the current state, initialized to 0

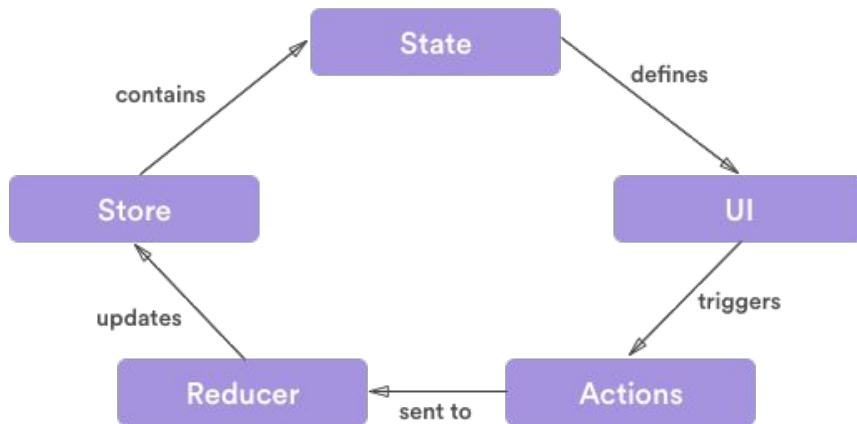
**setCount** is a function that updates state (causing the component to re-render)

# React: State

In React, components can have internal state that may change while the component is “mounted” (rendered / displayed).

**In React, there is no built-in solution for managing your entire application state. That is up to you.** You may have heard of libraries that help:

- redux / flux
- mobx
- recoil
- ...





# State management

!⚠️ What if our state doesn't live on the browser?

**Usually application state is spread across different computers.** Some application state will live in the user's browser while they are interacting with the app. Other application state will live on your server, or in your database.

# State management: databases

For durable data, databases help make guarantees about our ability to write data and make it easier to read (query) that data in the future.

 **There is no free lunch!** 

Durable writes are not easy.

- If requests A and B want to read and write the same data at the same time, how do we handle this?

# State management: databases

There are many database options to choose from. Broadly we can categorize them as:

- **Relational (SQL)**
  - MySQL
  - Postgresql
  - Oracle
  - MS SQL
- **Non-relational (NoSQL)**
  - Cassandra
  - MongoDB
  - Redis
  - ...

# State management: databases

## **Relational databases...**

- are a general-purpose persistence layer
- have more features
- limited ability to scale horizontally

## **Non-relational databases...**

- tend to be more specialized
- require more from the application layer
- are often better at scaling horizontally

# State management: databases

How we will approach database topics over the next few lectures:

- all about relational databases
- concurrency control
- SQL query analysis
- scaling options for relational databases
  - sharding
  - service-oriented architecture (SOA)
  - read replicas
- survey of NoSQL options

# State management: ORMs

**O**bject-relational **m**apping is a technique for converting data between incompatible type systems. 🤨

In object-oriented programming, we usually describe complex data in the form of objects. Objects can be pure data (e.g. JSON) or can be given behavior/methods (e.g. `class`).

In a relational database, we describe data in the form of tables (rows/columns).

# State management: ORMs

**O**bject-relational **m**apping is a technique for converting data between incompatible type systems. 🤨

In object-oriented programming, we usually describe complex data in the form of objects. Objects can be pure data (e.g. JSON) or can be given behavior/methods (e.g. `class`). **Objects can contain other objects.**

In a relational database, we describe data in the form of tables (rows/columns). **Columns contain scalar values.**

# State management: ORMs

ORMs let you express your tabular database data as objects in your programming language!

ORMs give you convenient syntax / methods for reading data (“entities”) out of the database and parsing it into your objects. Similar for writes.



# State management: ORMs

## Pros:

- less code to write (often significantly so)
- don't have to know/learn SQL (?)

## Cons:

- you don't know what it's doing under the hood
- complex queries not possible
- generated SQL often not as good/optimized as hand-written
- object-relational impedance mismatch (leaky abstraction)

# State management: ORMs

**Object-relational impedance mismatch** is a fancy way of saying objects and relational data don't play well together.

- **Granularity:** sometimes an object has more classes than corresponding tables
- **Subtypes/inheritance:** not represented in tabular data usually
- **Identity:** DB “sameness” based on primary key. Object “sameness” defined by programming language semantics
- **Associations:** DB references work differently than object references
- **Data navigation:** object field access is more efficient than DB table joins

# State management: ORMs

We will use an ORM for the class. It makes it faster to build features.

If you find when load testing your application that a particular ORM query is problematic, you can re-write that query in plain SQL, or using the TypeORM **query builder**.

# State management: server-side caching

We have a web server process that is repeatedly responding to HTTP requests from a variety of clients.

Responding to each request requires computation and I/O to be performed, and **this can be expensive.**

There are many parts of the responses that are similar, and many steps to creating a response that are repeated. **!?** What can you think of?

# State management: server-side caching

We have a web server process that is repeatedly responding to HTTP requests from a variety of clients.

Responding to each request requires computation and I/O to be performed, and **this can be expensive.**

There are many parts of the responses that are similar, and many steps to creating a response that are repeated. **!?** What can you think of?

- fetching/checking the user associated with the cookie
- rarely modified ORM objects
- any summarized data that is expensive to compute

# State management: server-side caching

! ? If we want to keep previously computed results around between requests, how should we do it? Where should we put it?

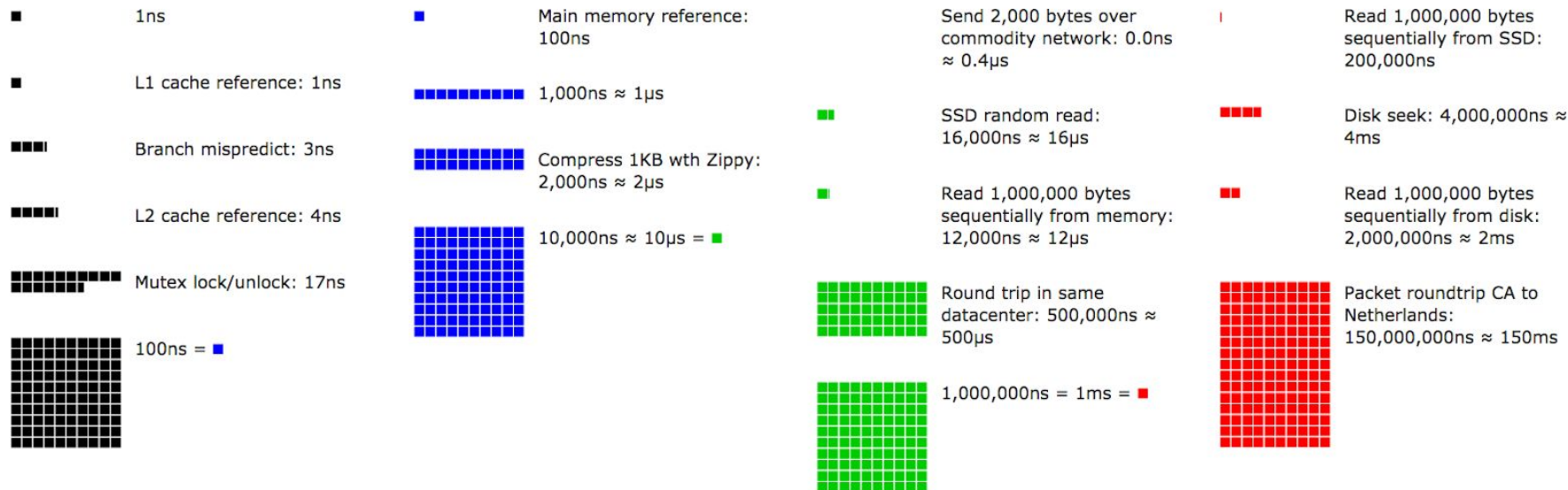
# State management: server-side caching

! ? If we want to keep previously computed results around between requests, how should we do it? Where should we put it?

- Keep it in memory between requests
- Store it on the filesystem
- Store it in memory on another machine

Each of these is reasonable for different use cases.

# State management: server-side caching



[source](#)



# State management: server-side caching

What can we conclude from these numbers?

- Storing in memory and reading later is fast
  - Random reads from memory will be  $0.1\mu\text{s}$ , reading 1MB will be  $12\mu\text{s}$
- Storing on disk is slow *without SSD*
  - Disk seek is  $4000\mu\text{s}$ , subsequent sequential read of 1MB is  $2000\mu\text{s}$
- Storing on disk *with SSD* is much more reasonable
  - Random read is  $16\mu\text{s}$ , sequential read of 1MB will be  $200\mu\text{s}$
- Storing on another machine is reasonable
  - Round trip within datacenter is  $500\mu\text{s}$ .

# State management: server-side caching

## Summary of latency:

- In memory: tens of  $\mu\text{s}$
- On SSD: hundreds of  $\mu\text{s}$
- On disk: thousands of  $\mu\text{s}$
- And if it's on a remote machine, add hundreds of  $\mu\text{s}$

## Conclusion:

- Always use SSD
- Memory > local SSD > remote (ish)

# State management: server-side caching

What effect on the cache hit rate does each of these designs have?

- In memory: cache per process
- On local SSD: cache per machine
- On (single) remote machine: cache per cluster

## Conclusion:

- In memory: highest performance, lowest hit rate
- On SSD: lower performance, higher hit rate
- On remote server: lowest performance, highest hit rate

 There is no silver bullet. Don't forget about cache invalidation! 

# State management: server-side caching

**Redis** and **Memcached** are commonly used implementations for a remote cache server.

- Keeps cache in memory
- Accepts TCP connections and returns lookup requests
- Distributed key/value store
  - Can scale horizontally

Memcached is designed more specifically for caching (e.g. has LRU eviction when it runs out of space). Redis is more general purpose and has cool features.

In this class, you can use Redis.