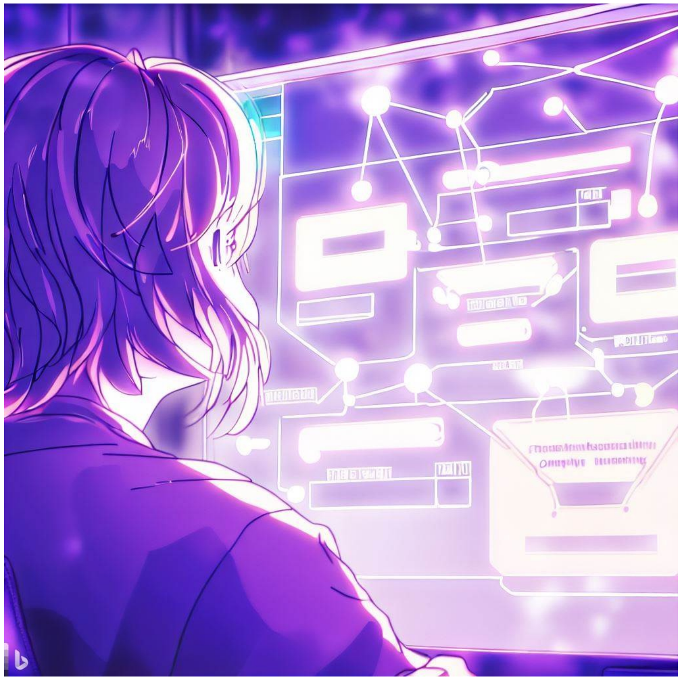


CSE 331

App and Data Design

Kevin Zatloukal

Component Modularity

Component Modularity

- Poor design to put all the app in one Component
  - it works, but is lacks some properties of high quality
  - better to break it into smaller pieces (modular)

- Two ways to the UI into separate components:

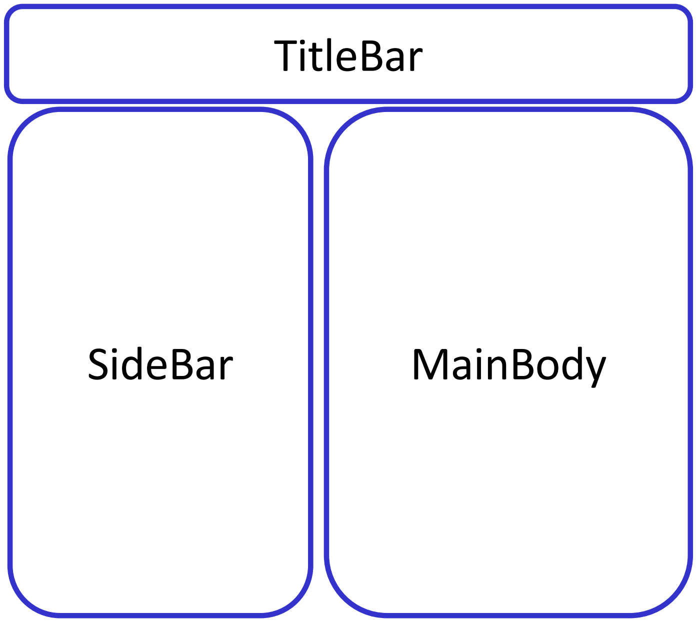
  1. Separate parts that are next to each other

  2. Separate parts on the screen at different times

Component Modularity

- **Separate parts that are next to each other**

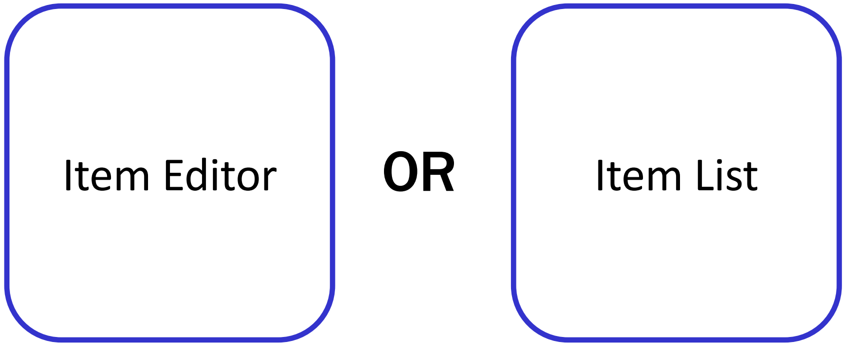
```
class App extends Component<..> {
  render = (): JSX.Element {
    return (<div>
        <TitleBar title={“My App”}/>
        <SideBar/>
        <MainBody/>
      </div>);
  };
}
```



Component Modularity

- Separate parts on the screen at different times

- App is always on the screen
  - App chooses which child component to display



  - sometimes it has an Editor child and sometimes not

Component Modularity

- **Separate parts on the screen at different times**

```
type AppState = {editing: boolean};

class App extends Component<{}, AppState> {
  …
  render = (): JSX.Element {
    if (this.state.editing) {
      return <ItemEditor item={this.state.item}/>;
    } else {
      return <ItemList/>;
    }
  };
  …
}
```

Example: Quarter Picker

Writing a Full Stack App

Steps to Writing a Full Stack App

- Assume we know what the app should look like
  - all different interactions are describe to us

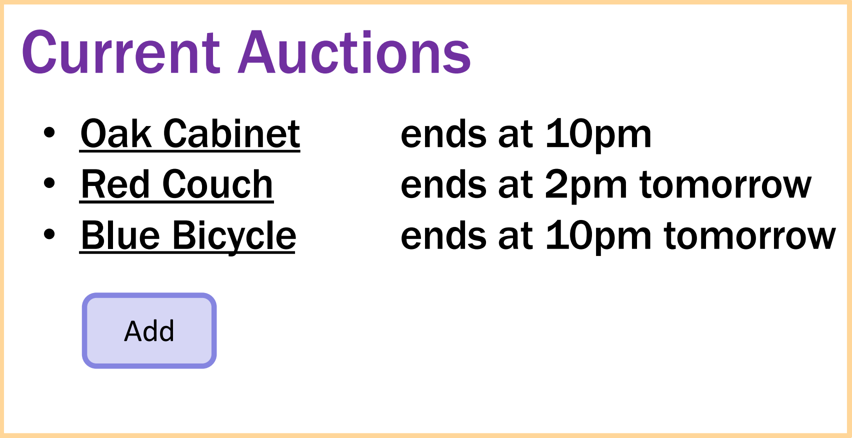
- Then we can write it in the following order:

  1. Write the server
     - official store of the data (client state is ephemeral)
     - only provide the operations needed by the client
  2. Write the client UI with local data
     - no client/server interaction at the start
  3. Connect the client to the server
     - use fetch to update data on the server

could swap these

Example: Auction Site

- Initial page shows user a list of auctions
  - can also add their own



Current Auctions

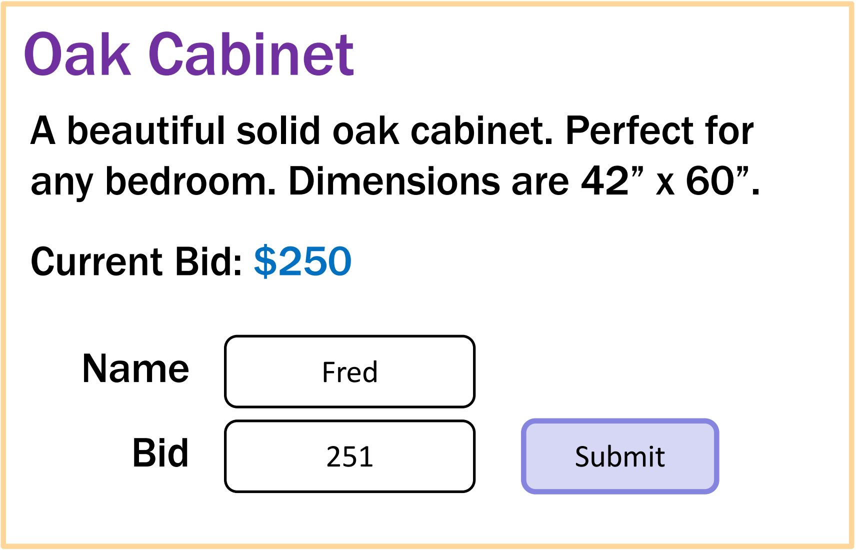
• Oak Cabinet	ends at 10pm
• Red Couch	ends at 2pm tomorrow
• Blue Bicycle	ends at 10pm tomorrow

can click on item name

can click on Add

Example: Auction Site

- Clicking on an item shows the full details
  - allows user to bid



Oak Cabinet

A beautiful solid oak cabinet. Perfect for any bedroom. Dimensions are 42” x 60”.

Current Bid: $250

Name: Fred

Bid: 251

click Submit to bid

Show an error if the user:

- does not enter a name
- enters a non-number bid
- enters a bid smaller than the current bid

Example: Auction Site

- Clicking on an item shows the full details
  - allows user to bid

Oak Cabinet

A beautiful solid oak cabinet. Perfect for any bedroom. Dimensions are 42” x 60”.

Final Bid: $250

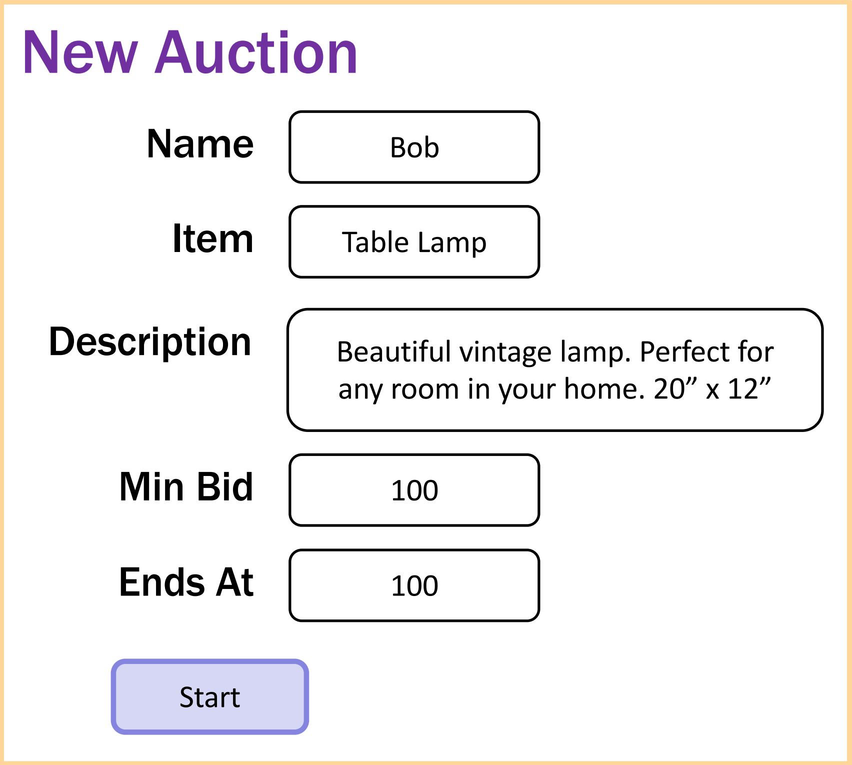
Won By: Alice

Don’t let users bid if the auction is over.

Instead, show who won the auction.

Example: Auction Site

- Click on Add allows the user to start a new auction
  - user provides the full details of the



New Auction

Name	Bob
Item	Table Lamp
Description	Beautiful vintage lamp. Perfect for any room in your home. 20” x 12”
Min Bid	100
Ends At	100

click Start to start auction

Writing the Server

App and Data Design

- Most applications are centered on data
  - present data to the user
  - allow them to manipulate the data and see the result

- App design is first and foremost *data* design
  - first step is to decide what data to store
  - then think about how to display it, change it, etc.

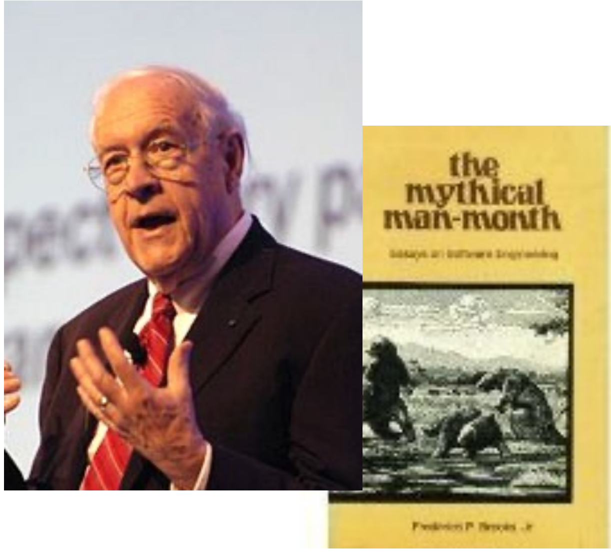
Data Before Code

Bad programmers worry about the code. Good programmers worry about data structures and their relationships.

-- Linus Torvalds

Show me your flowcharts and conceal your tables, and I shall continue to be mystified. Show me your tables, and I won't usually need your flowcharts; they'll be obvious.

-- Fred Brooks



Entities & Operations

- **Figure out what data to store by asking ourselves**

  1. **What entities do we need to keep track of?**
     - become records or ADTs stored in the server
     - what information do we need about each one?

  2. **What operations do we need to perform on them?**
     - become routes in the server
     - typical operations
       - list all the entities
       - find entities with certain properties
       - add an entity, remove an entity
       - change an entity in some way (depends on the type of entity)

Example: To-Do List

1. **What entities do we need to keep track of?**
   - items on the To Do List
   - each one has a name and when it was completed

     {name: string, completedAt: Date}

2. **What operations do we need to perform on them?**
   - supported operations:
     - list all the items
     - add an item
     - mark an item completed (change)
     - no way to remove (happens automatically)

Example: Auction Site

1. What entities do we need to keep track of?

Example: Auction Site

1. **What entities do we need to keep track of?**

Auctions

- what information do we need about each one?

Example: Auction Site

1. What entities do we need to keep track of?

Auctions

- what information do we need about each one?

name of owner	string
name of item	string
description of item	string
minimum bid	number
ending time	Date

got these from the Add Auction UI

what else?

Example: Auction Site

1. What entities do we need to keep track of?

Auctions

- what information do we need about each one?

name of owner	string
name of item	string
description of item	string
minimum bid	number
ending time	Date
highest bidder	string
highest bid	number

```
type Auction = {itemName: string, … };
```

Example: Auction Site

2. What operations do we need to perform on them?

Example: Auction Site

2. What operations do we need to perform on them?

- list all auctions
- add an auction
- bid on an auction (change)

got these from the UI

could also do

- get auction by name

Server Data Structures

- **Start with the simplest data structure**
  - **list or array of records**
    - one record per entity

- **Start concrete, not abstract**
  - **ADTs introduce abstraction & complexity**
  - **wait until the data structures are tricky to make an ADT**
    - may not be necessary at all!
    - wait until you know it is too slow to change data structures

Server Data Structures

- **Start with the simplest data structure**
  - **list or array of records**

    one record per entity

- **One option worth considering is using a `Map`**
  - **TypeScript has `Map<K, V>` just like Java**

    key methods are `get` and `set` (see MDN for more)

  - **using a `Map` can be easier**

    no need to write a loop to find something!

Examples

- How should we store items in a To Do List app?

list or array of items

- How should we store items in our Auction app?

list or array of auctions

or map from item name to record

```
Map<string, Auction>
```

Testing the Server

- Write unit tests for each route / function
- Can also test manually
  - type the URL into the browser and see the response
    - works only for GET requests
    - for POST, either change to GET temporarily or use a tool (e.g., curl)

More Realistic Data

- In practice, data is more complex
  - many kinds of entities
  - complex relationships between them
  - complex invariants

- Useful tools for modeling these
  - e.g., entity-relationship diagrams
  - see 344 for more on that

More Realistic Servers

- In practice, can’t store user data on one machine
  - machines break, hard drives fail, etc.

- Sharing state between servers is very complex
  - requires even more sophisticated invariants
  - see 452 for more on this

- Most apps use existing software for this
  - relational or non-relational database of some kind
  - see 344 for more on that

- App logic in server becomes purely functional!

Writing the Client

Design on the Client Side

- Design the server by thinking about entities & ops

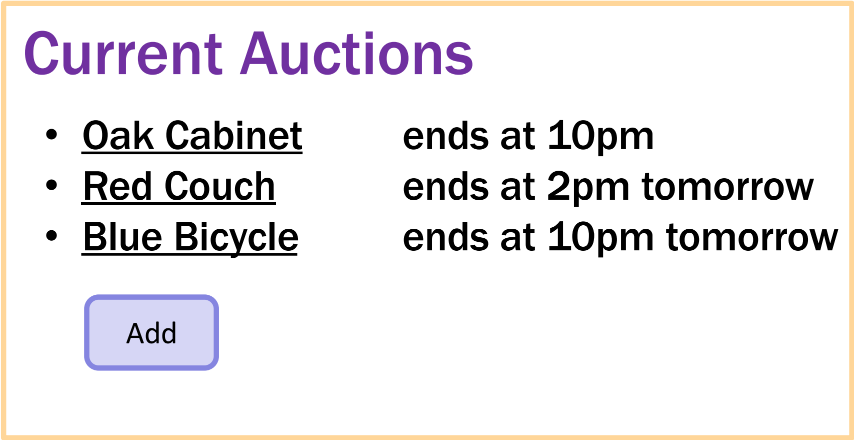
- Designing the client is different
  - component state is **tightly coupled** with UI on the screen
  - must store state to render exactly what you see

- Design the client by thinking about what you see
  - what component do you need to show that UI
  - different "pages" require different components

    also need a parent component that decides which one to show

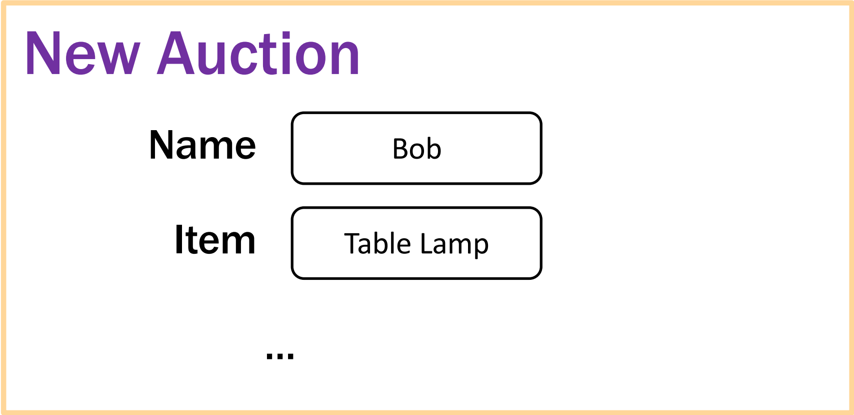
Example: Auction UI

- Auction site had three different “pages”



Current Auctions

Oak Cabinet	ends at 10pm
Red Couch	ends at 2pm tomorrow
Blue Bicycle	ends at 10pm tomorrow

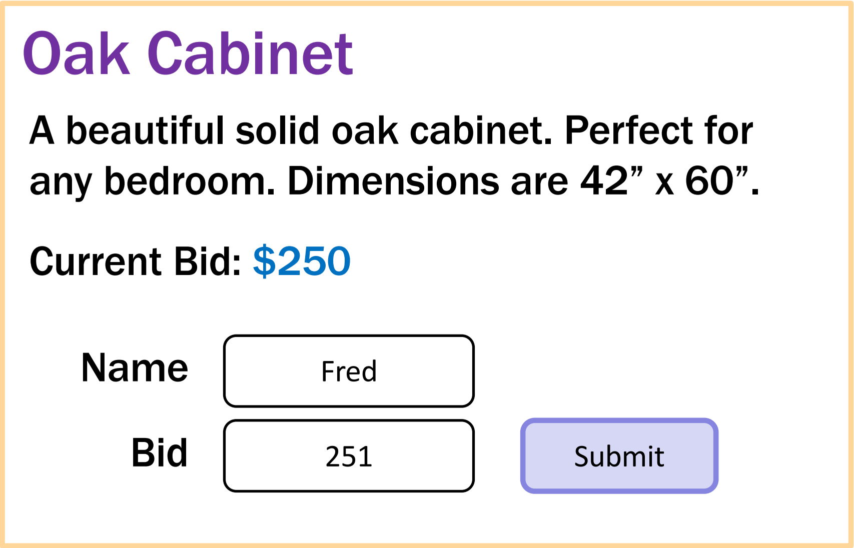


New Auction

Name: Bob

Item: Table Lamp

...



Oak Cabinet

A beautiful solid oak cabinet. Perfect for any bedroom. Dimensions are 42” x 60”.

Current Bid: $250

Name: Fred

Bid: 251

Example: Auction UI

- Auction site had three different “pages”
- Need four different components:
  - Auction List: shows all the auctions (and Add button)
  - Auction Details: shows details on the auction (w Bid button)
  - New Auction: lets the user describe a new auction

what else?

Example: Auction UI

- Auction site had three different “pages”

- Need four different components:
  - Auction List: shows all the auctions (and Add button)
  - Auction Details: shows details on the auction (w Bid button)
  - New Auction: lets the user describe a new auction
  - App: decides which of these pages to show

Example: Auction UI

- `AuctionList.tsx`
  - **state stores the full list of auctions**

    fetch this from the server when created
  - **“Add” goes back to the New Auction page**

    onAdd prop tells the App to switch to the auction list
  - **clicking on an auction goes to the Auction Details page**

    onShow prop tells the App to switch to the details of that auction

Example: Auction UI

- `AuctionDetails.tsx`
  - **state stores the details of the auction**
  - **render shows the result of the auction or UI to bid**
  - **"Submit" bid button makes a /bid request to the server**

    display an error or success message upon completion
  - **"Back" button goes back to the Auction List page**

    onBack prop tells the App to switch back to the auction list UI

Example: Auction UI

- `NewAuction.tsx`
  - **state stores all the data shown on the page**

    name, item, description, min bid, ends at
  - **“Start” button makes /new request to server**

    display an error or success message upon completion
  - **“Back” button goes back to the Auction List page**

    onBack prop tells the App to switch back to the auction list UI

Example: Auction UI

- `App.tsx`
  - state says which page to be showing

```
type Page = “list” | “add” |
            {kind: “details”, itemName: string};

type AppState = {page: Page};

class App extends Component<{}, AppState> { … }
```

  - `Page` is an inductive data type of the “enum” variety

Example: Auction UI

- App.tsx
  - render shows the appropriate UI

```
render = (): JSX.Element => {
  if (this.state.page === “list”) {
    return <AuctionList onAdd={this.handleAdd}
                        onShow={this.handleShow}/>;
  } else if (this.state.page === “add”) {
    return <NewAuction onBack={this.handleBack}/>;
  } else {
    return <AuctionDetails
                itemName={this.state.page.itemName}
                onBack={this.handleBack}/>;
  }
};
```

Example: Auction UI

- `App.tsx`
  - **event handlers change what is shown**

```
handleAdd = (): void => {
  this.setState({page: “add”});
};

handleBack = (): void => {
  this.setState({page: “list”});
};

handleShow = (itemName: string): void => {
  this.setState({page: {kind: “details”,
                        itemName: itemName}});
};
```