

# Principles of Software Construction: Objects, Design, and Concurrency

## Events Everywhere

**Christian Kästner**   Vincent Hellendoorn





# Outline

- Revisiting Immutability
- Model-View-Controller
- Event-Based Programming, Reactive Programming
- ReactJS UI

# Revisiting Immutability

# Reading Quiz: Immutability

<https://bit.ly/214q1026>



# Recall: Why Immutability?

# Recall: Ensuring Immutability

- Don't provide any mutators
- Ensure that no methods may be overridden
- Make all fields final
- Make all fields private
- Ensure security of any mutable components

# Immutable?

```
class Stack {
  readonly #inner: any[]
  constructor (inner: any[]) {
    this.#inner=inner
  }
  push(o: any): Stack {
    const newInner = this.#inner.slice()
    newInner.push(o)
    return new Stack(newInner)
  }
  peek(): any {
    return this.#inner[this.#inner.length-1]
  }
  getInner(): any[] {
    return this.#inner
  }
}
```



# Immutable?

Inner mutable state  
(List in Java)

Create copy of  
mutable object  
(new ArrayList(old)  
in Java)

Return new  
immutable object


```
class Stack {
  readonly #inner: any[]
  constructor (inner: any[]) {
    this.#inner=inner
  }
  push(o: any): Stack {
    const newInner = this.#inner.slice()
    newInner.push(o)
    return new Stack(newInner)
  }
  peek(): any {
    return this.#inner[this.#inner.length-1]
  }
  getInner(): any[] {
    return this.#inner
  }
}
```

# Aliasing is what makes Mutable State risky

Many variables may point to same object

Any reference to the object can modify the object, effect seen by all other users

x, y, and z all point to  
the same mutable  
array



```
const x = [ 1, 2, 3 ]
const y = x
function foo(z: number[]): void { /*...*/ }
foo(y)
```

# Immutable?

Inner mutable state  
(List in Java)

Create copy of  
mutable object  
(new ArrayList(old)  
in Java)

Return new  
immutable object

Leak mutable state  
Accept mutable state

```
class Stack {
  readonly #inner: any[]
  constructor (inner: any[]) {
    this.#inner = inner
  }
  push(o: any): Stack {
    const newInner = this.#inner.slice()
    newInner.push(o)
    return new Stack(newInner)
  }
  peek(): any {
    return this.#inner[this.#inner.length-1]
  }
  getInner(): any[] {
    return this.#inner
  }
}
```

# Fixed

```
class Stack {
  readonly #inner: any[]
  constructor (inner: any[]) {
    this.#inner=inner.slice()
  }
  push(o: any): Stack {
    const newInner = this.#inner.slice()
    newInner.push(o)
    return new Stack(newInner)
  }
  peek(): any {
    return this.#inner[this.#inner.length-1]
  }
  getInner(): any[] {
    return this.#inner.slice()
    // Java: return new ArrayList(inner)
  }
}
```

# Recall: Ensuring Immutability

- Don't provide any mutators
- Ensure that no methods may be overridden
- Make all fields final
- Make all fields private
- **Ensure security of any mutable components**

# Writing Immutable Data Structures

Any “set” operation returns a new copy of an object  
(can point to old object to save memory, e.g. linked lists)

Final fields of immutable objects are save (e.g., strings, numbers)

Fields of mutable objects must be protected  
(encapsulation, making copies)

Careful with mutable constructor/method arguments (make copies)

Easy to make mistakes when mixing mutable and immutable data structures, only academic tools for checking

# Trend toward immutable data structures

Immutable data structures common in functional programming

Many recent languages and libraries embrace immutability

Scala, Rust, stream, React, Java Records

Simplifies building concurrent and distributed systems

Requires some practice when used to imperative programming with mutable state, but will become natural

# Circular references & Caching

Immutable data structures often from a directed acyclic graph

Cycles challenging

Cycles often useful for performance (caching)

```
class TreeNode {
  readonly #parent: TreeNode
  readonly #children: TreeNode[]
  constructor(parent: TreeNode,
              children: TreeNode[]) {
    this.#parent = parent
    this.#children = children
  }
  addChild(child: TreeNode) {
    const newChildren = this.#children.slice()
    //const newChild = child.setParent(this) ??
    newChildren.push(child)
    const newNode = new TreeNode(this.#parent,
                                  newChildren)
    //child.setParent(newNode) ??
    return newNode
  }
}
```



# Design Discussion

## Design for Understandability / Maintainability

- Immutable objects are easy to reason about, they won't change
- Mutable objects have more complicated contracts, function and client both can modify state
- Do not need to think about corner cases of concurrent modification

## Design for Reuse

- Easy to reuse even in concurrent settings

# Java 16 Records

Records are (shallowly) immutable

No setters

But also no defensive copying of mutable fields

# Reactive Programming

# Reactive Programming

Programming strategy or patterns, where programs react to data

Embraces concurrency, focuses on data flows

Takes event-based programming to an extreme

Decouples programs around data

# Useful analogy: Spreadsheets

Cells contain data or formulas

Formula cells are computed automatically whenever input data changes

	A	B
1		0
2	1	=A2+B1
3	2	3
4	3	6
5		

# Implementing Spreadsheet-Like Computations?

# Implementing Spreadsheet-Like Computations?

```
x = 3
y = 5
z = x + y
print(z) // prints 8
x = 5
print(z) // expect 10, prints 8
```

in imperative computations,  
no update when inputs change

# Implementing Spreadsheet-Like Computations?

```
x = 3
y = 5
z = () => x + y
print(z()) // prints 8
x = 5
print(z()) // prints 10
```

computation performed on demand (pull)  
caching possible

Does not easily work in Java, since Java requires variables in closure to be final. Need object with mutable internal state



# Implementing Spreadsheet-Like Computations?

```
x = new Cell(3)
y = new Cell(5)
z = new DerivedCell(x, y, (a,b)=>a+b)
print(z.get()) // prints 8
x.set(5)
print(z.get()) // prints 10
```

Cell implements observer pattern,  
informs observers of changes (push)

DerivedCell listens to changes from Cell,  
updates internal state on changes,  
informs own observers of changes

# Complications

Single change in cell can trigger many computations (push)

Possibly put in queue, compute asynchronously

Perform some computations lazily when needed

Cyclic dependencies can result in infinite loops

Detect, special ways to handle

Observers can hinder garbage collection

	A	B
1		0
2	1	1
3	2	3
4	3	6
5	#REF!	=B4+3+A5
6		

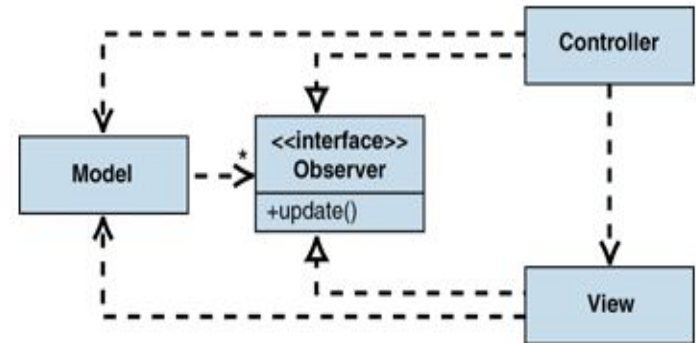
# Reactive Programming and GUIs

Store state in observable cells, possibly derived

Have GUI update automatically on state changes

Have buttons perform state changes on cells

Mirrors active model-view-controller pattern, discussed later  
(model is observable cell)



# From Pull to Push

Instead of clients to look for state (pull)

observers react to state changes with actions (push)

Commonly observables indicate that something has changed, triggering observers to get updated state (push-pull)

# Beyond Spreadsheet Cells

	SINGLE	MULTIPLE
Pull	Function	Iterator
Push	Promise	Observable

<https://rxjs.dev/guide/observable>

# Reactive Programming Libraries

RxJava, RxJS, many others

Provide Stream-like interfaces for event handling, with many convenience functions (similar to promises)

Observables typically allow pushing multiple values in sequence

Cells can be implemented by considering only the latest value of observables

# Previous Example with RxJava

```
PublishSubject<Integer> x = PublishSubject.create();  
PublishSubject<Integer> y = PublishSubject.create();  
Observable<Integer> z = Observable.combineLatest(x, y,  
(a,b)->a+b);  
z.subscribe(System.out::println);  
x.onNext(3);  
y.onNext(5);  
x.onNext(5);
```

# Chaining Computations along Data

```
awk '{print $7}' < /var/log/nginx/access.log |  
sort |  
uniq -c |  
sort -r -n |  
head -n 5 > out
```

Multiple programs executed in sequence each read lines and produce lines;  
can start reading lines before previous program is finished



# Streams / Reactive Programming / Events

Instead of calling methods in sequence,  
set up pipelines for data processing

Let data control the execution

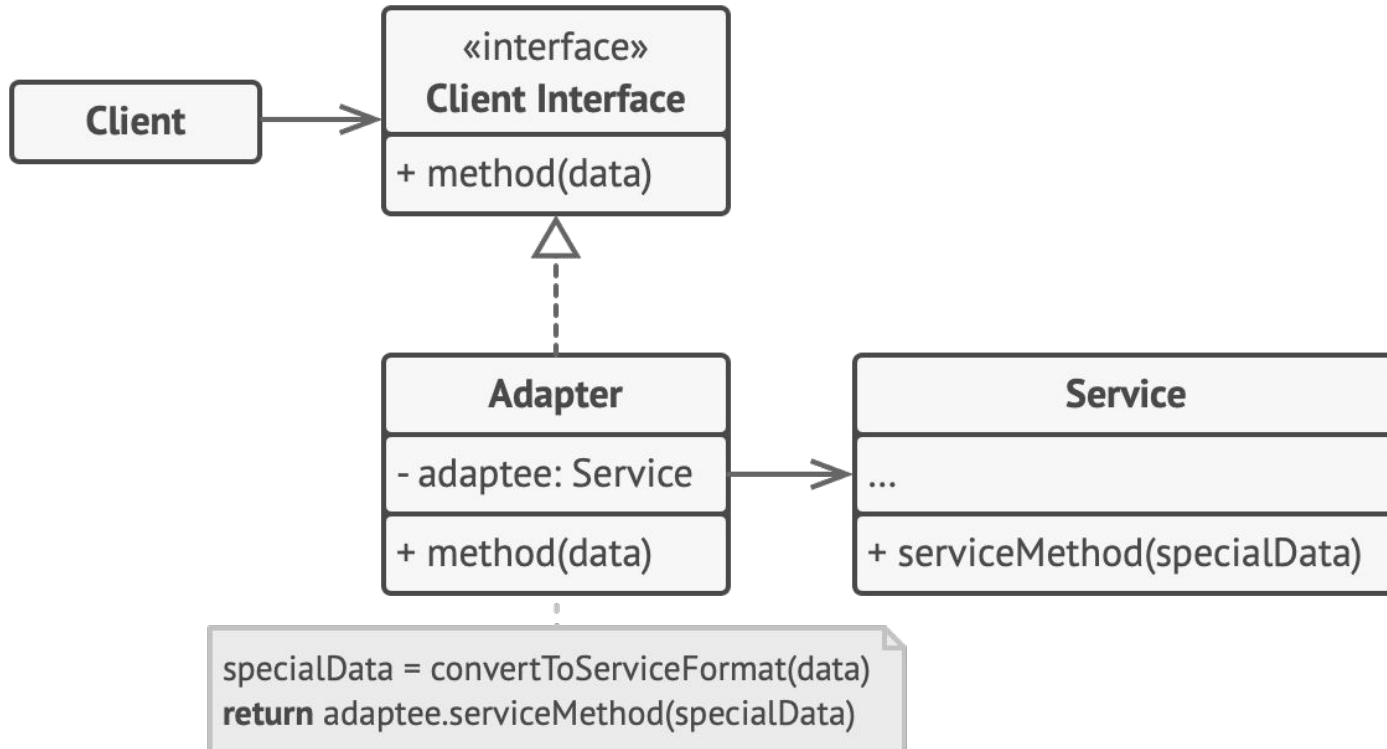
```
var lines = IOHelper.readLinesFromFile(file);  
var linesObs = Observable.fromIterable(lines);  
linesObs.  
    map(Parser::getURLColumn).  
    groupBy(...).  
    sorted(comparator).  
    subscribe(IOHelper.writeFile(outFile));
```

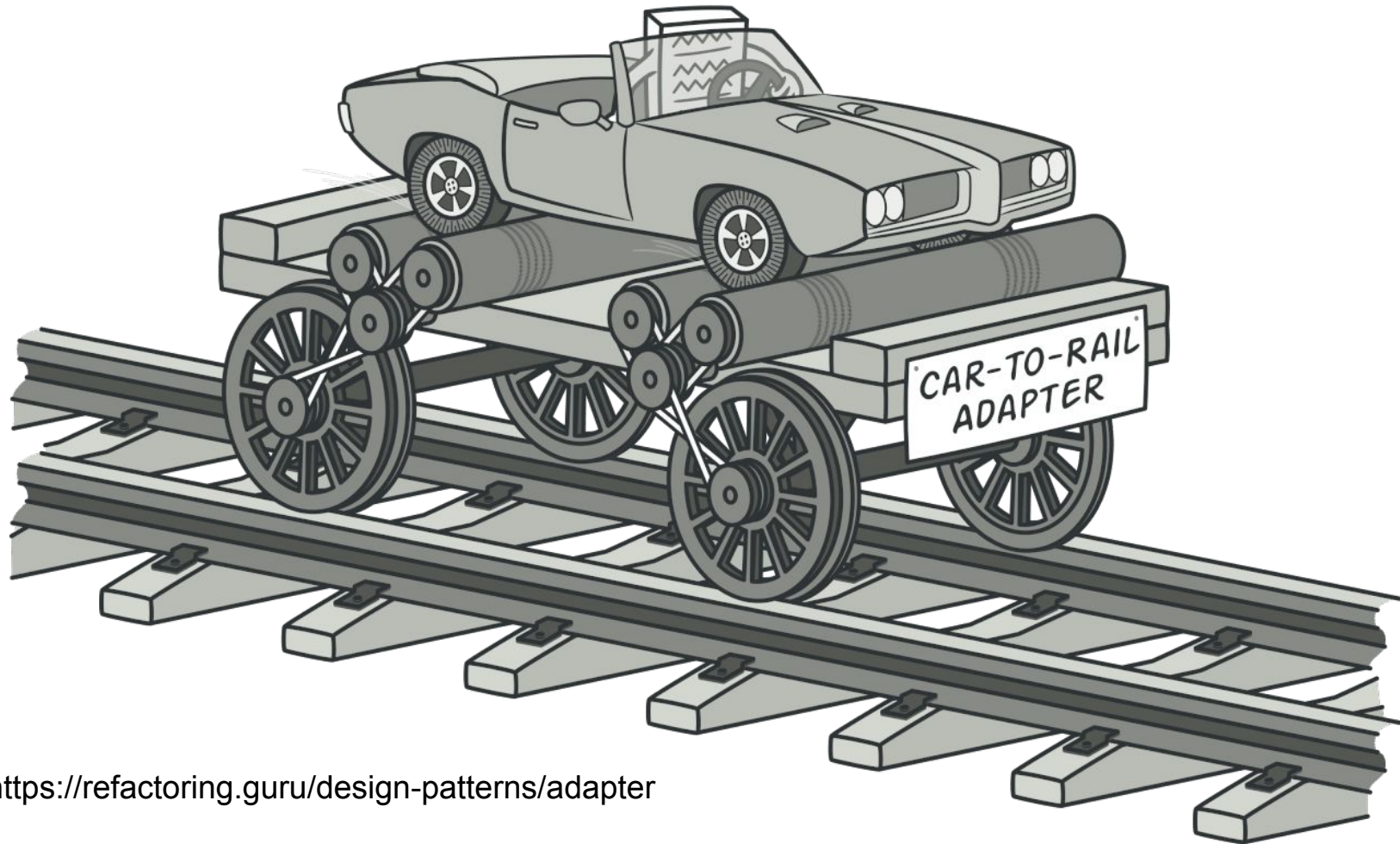
# Many more Features in Reactive Programming Libraries

Backpressure (see last lecture)

# Aside: The Adapter Pattern

# The *Adapter* Design Pattern



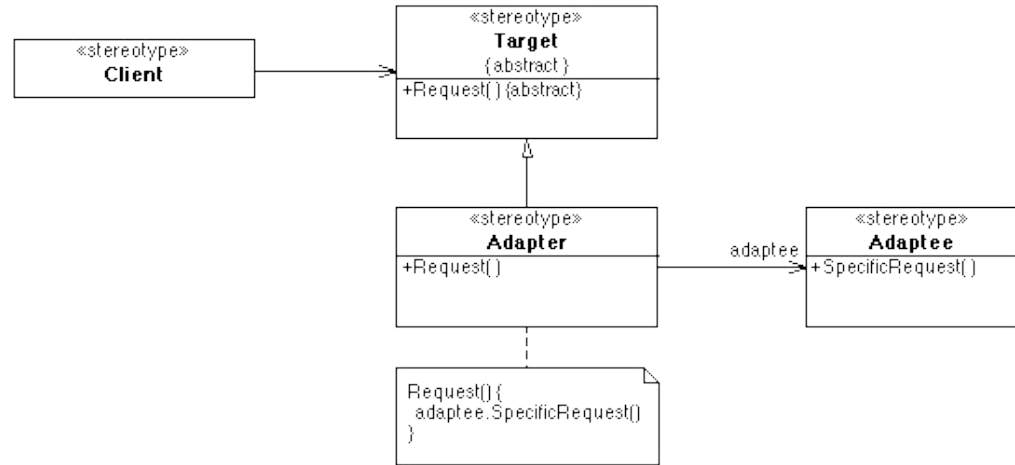


<https://refactoring.guru/design-patterns/adapter>

# The *Adapter* Design Pattern

## Applicability

- You want to use an existing class, and its interface does not match the one you need
- You want to create a reusable class that cooperates with unrelated classes that don't necessarily have compatible interfaces
- You need to use several subclasses, but it's impractical to adapt their interface by subclassing each one



## Consequences

- Exposes the functionality of an object in another form
- Unifies the interfaces of multiple incompatible adaptee objects
- Lets a single adapter work with multiple adaptees in a hierarchy
- -> **Low coupling, high cohesion**

# Adapters for Collections/Streams/Observables

```
var lines = IOHelper.readLineFromFile(file);  
var linesObs = Observable.fromIterable(lines);  
linesObs.  
    map(Parser::getURLColumn).  
    groupBy(...).  
    sorted(comparator).  
    subscribe(IOHelper.writeToFile(outFile));
```

Any others?

# Façade/Controller vs. Adapter

- Motivation
  - Façade: simplify the interface
  - Adapter: match an existing interface
- Adapter: interface is given
  - Not typically true in Façade
- Adapter: polymorphic
  - Dispatch dynamically to multiple implementations
  - Façade: typically choose the implementation statically



# Core vs GUI

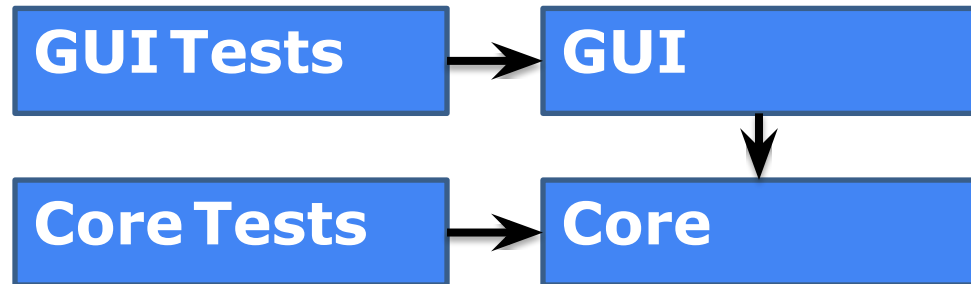
## Backend vs Frontend

# Recall: Core implementation vs. GUI

- Core implementation: application logic
  - Computing some result, updating data
- GUI
  - Graphical representation of data
  - Source of user interactions
- Design guideline: *avoid coupling the GUI with core application*
  - Multiple UIs with single core implementation
  - Test core without UI

# Recall: Separating application core and GUI

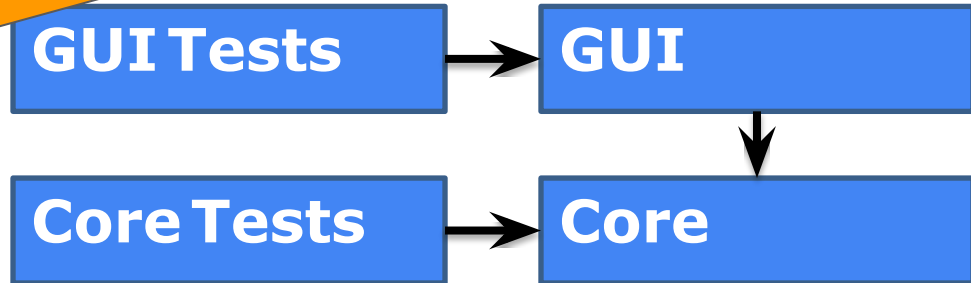
- Reduce coupling: do not allow core to depend on UI
- Create and test the core without a GUI
  - Use the Observer pattern to communicate information from the core (Model) to the GUI (View)



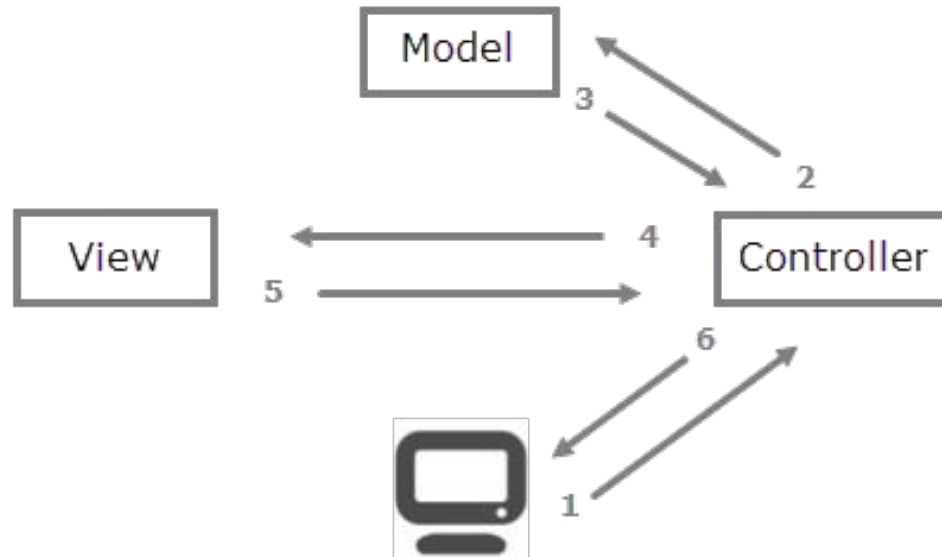
# Recall: Separating application core and GUI

- Reduce coupling: do not allow core to depend on UI
- Create and test the core without a GUI
  - Use the Observer pattern to separate the GUI from the core (Model) to the GUI

What design goals does this further?

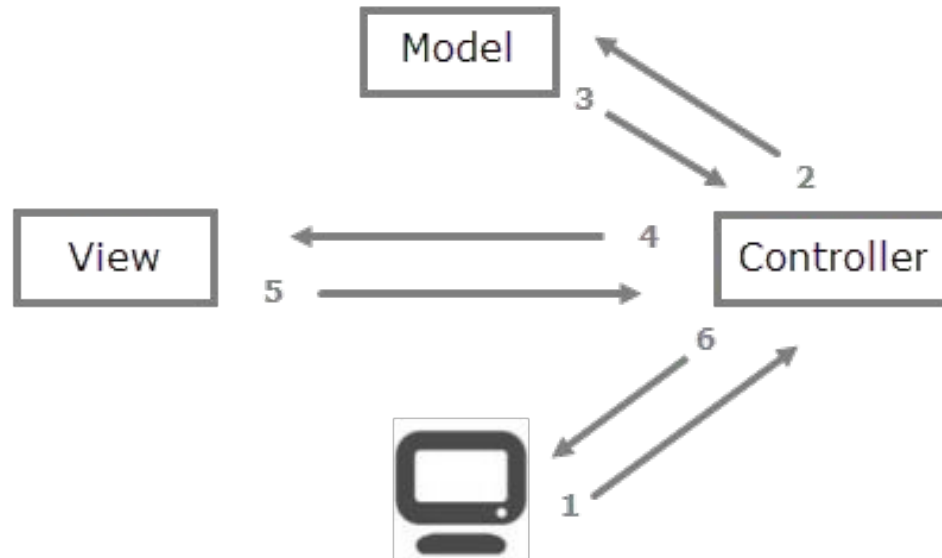


# Model View Controller



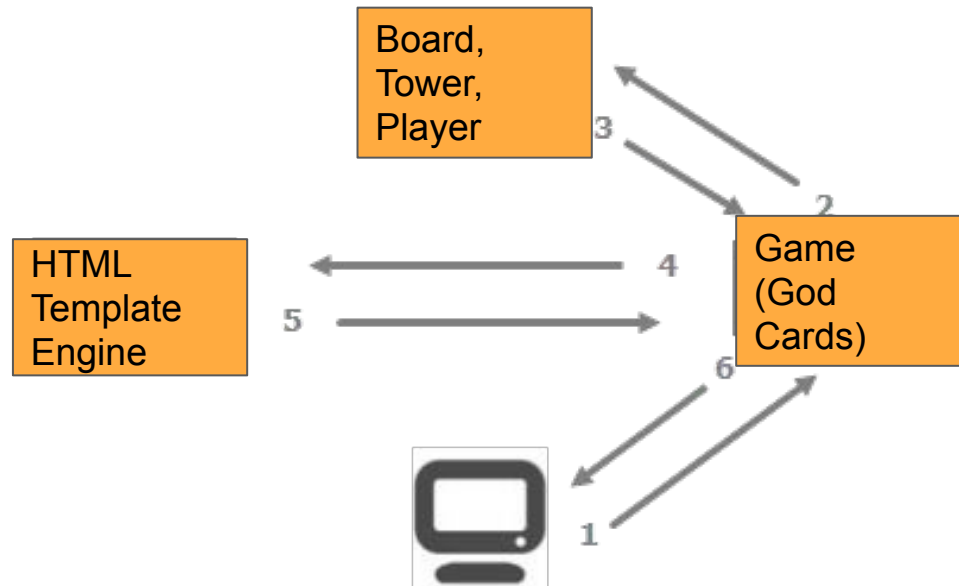
<https://overiq.com/django-1-10/mvc-pattern-and-django/>

# Model View Controller in Santorini?



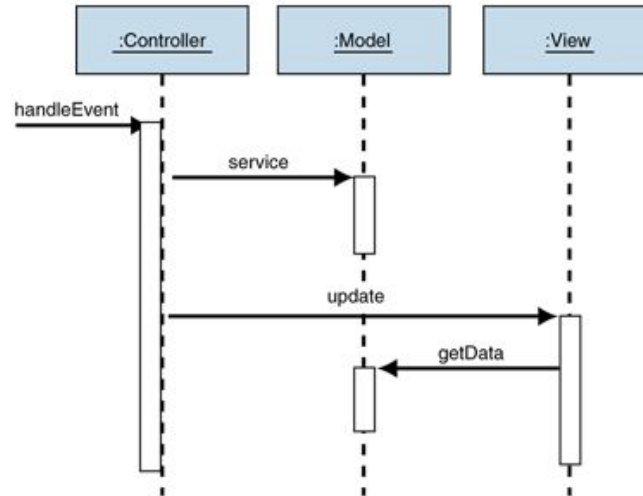
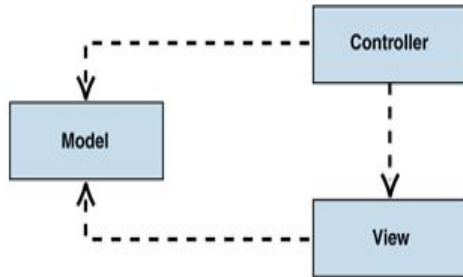
<https://overiq.com/django-1-10/mvc-pattern-and-django/>

# Model View Controller in Santorini?



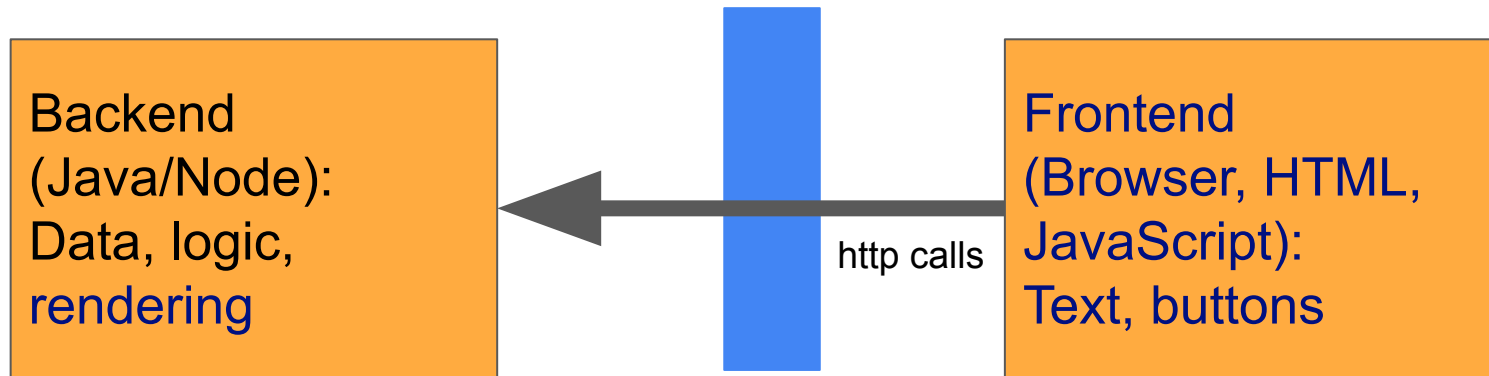
<https://overiq.com/django-1-10/mvc-pattern-and-django/>

# Model View Controller Dependencies





# Client-Server Programming forces Frontend-Backend Separation

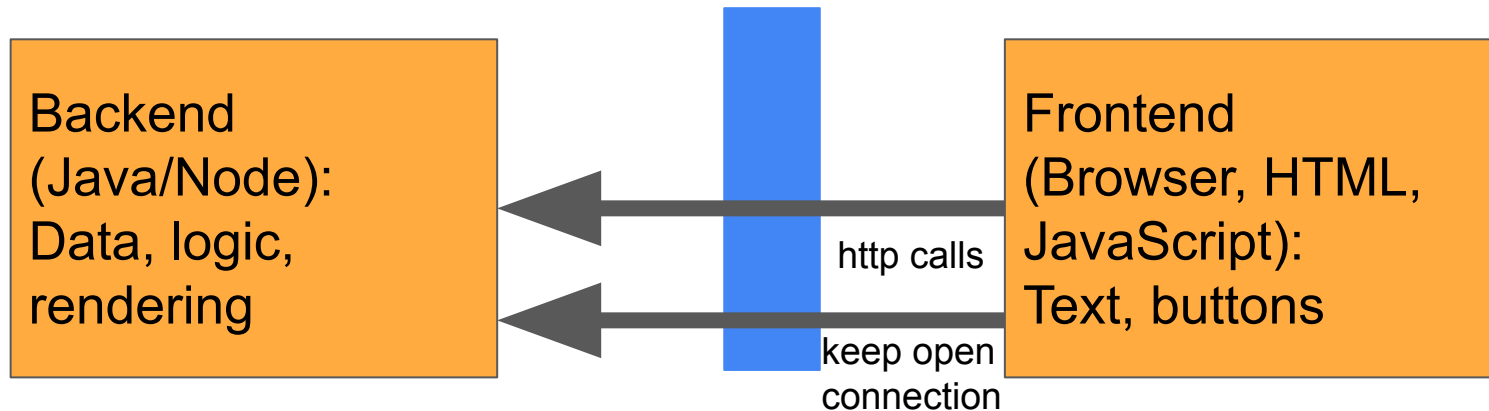


Browser can call web server, but not the other way around

Browser needs to *pull* for updates

Browser can request entirely no page or just additional content (ajax, REST api calls, ...)

# Client-Server Programming forces Frontend-Backend Separation



Trick to let backend push information to frontend: Keep http request open, append to page (compare to stream)

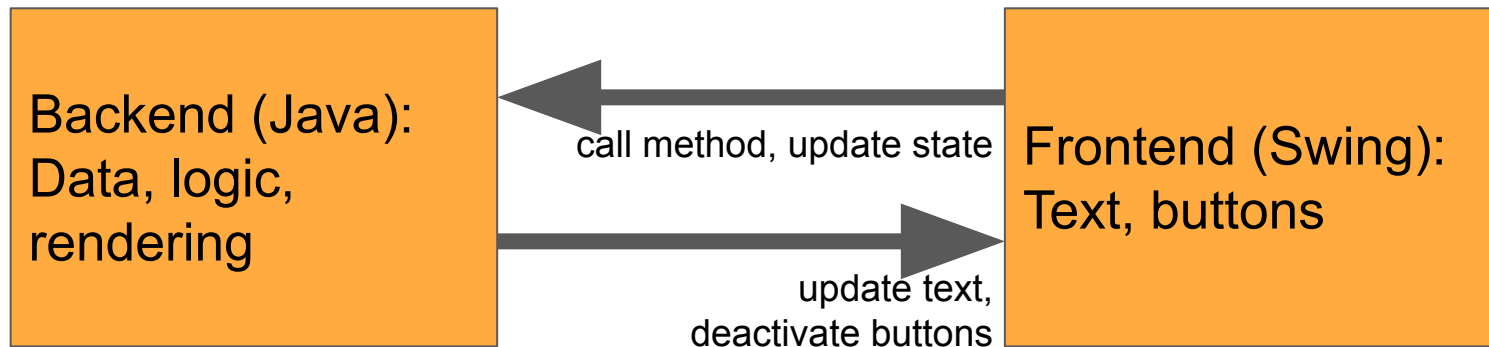
Alternative: regular pulling

# Core & Gui in same environment

JavaScript frontend and backend together in browser  
(e.g. using *browserify*) -- single threaded!

Java Swing GUI running in same VM as core logic -- multi threaded

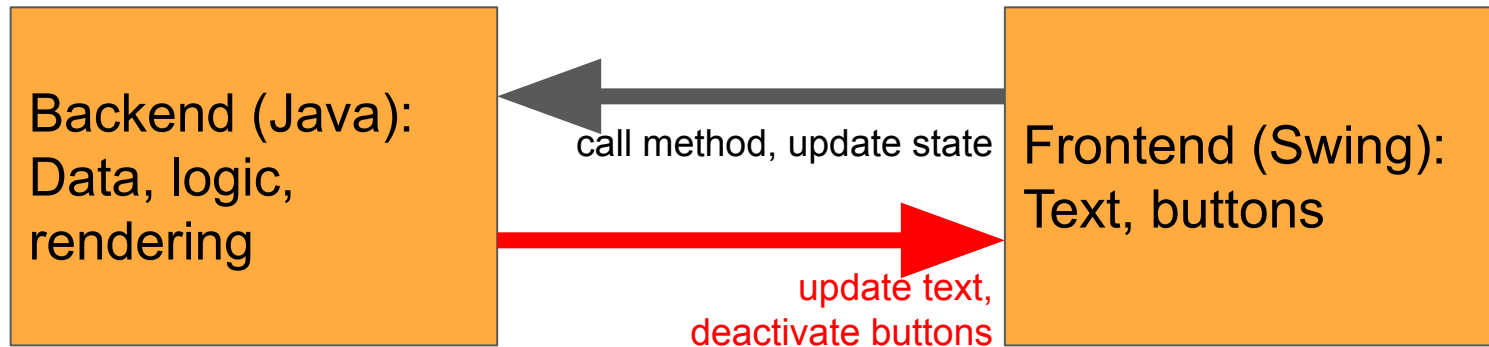
Core logic could directly modify GUI



# Avoid Core to Gui coupling

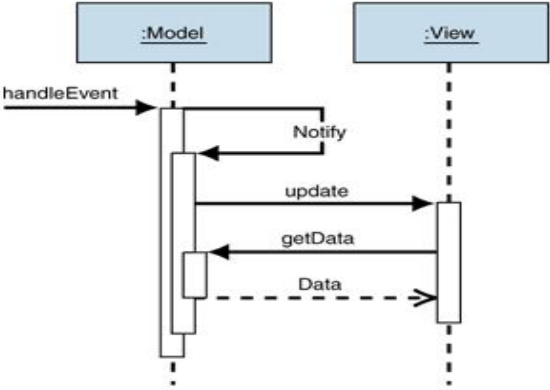
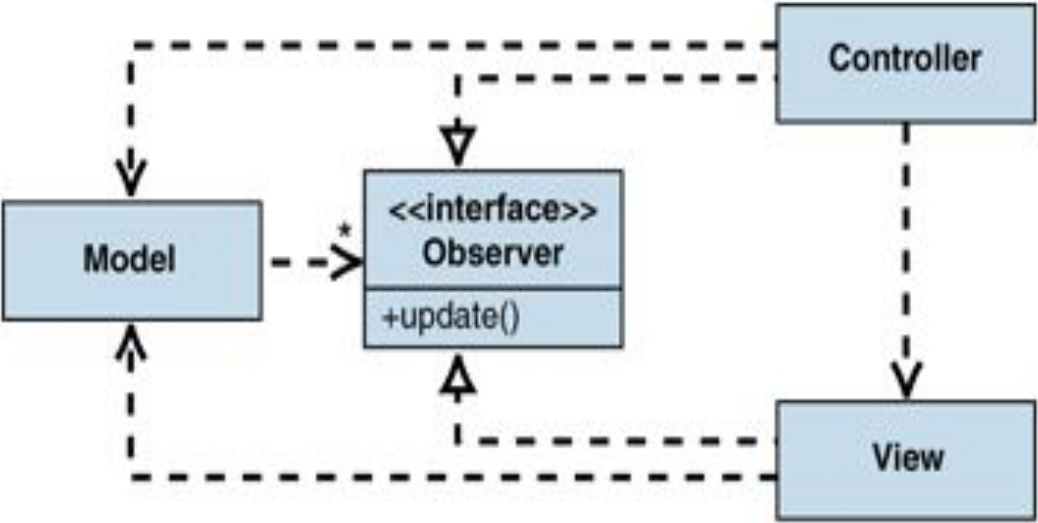
Never call the GUI from the Core

Update GUI after action (pull) or use observer pattern instead to inform GUI of updates (push)



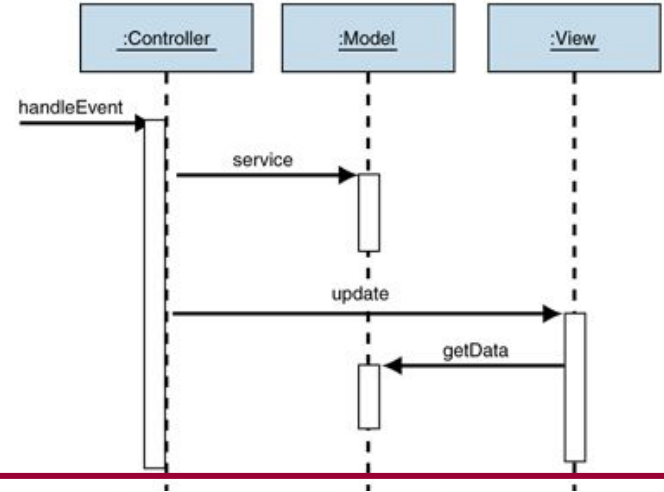
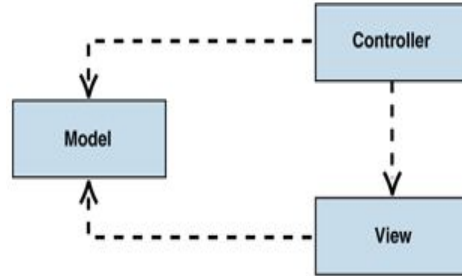
# Excursion: Active Model View Controller

(Not commonly used)

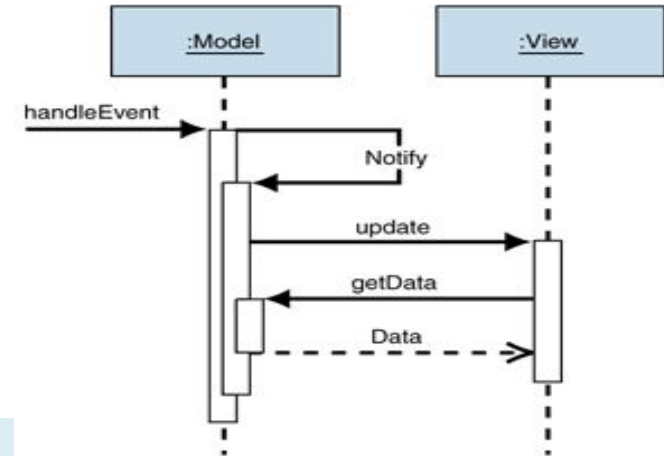
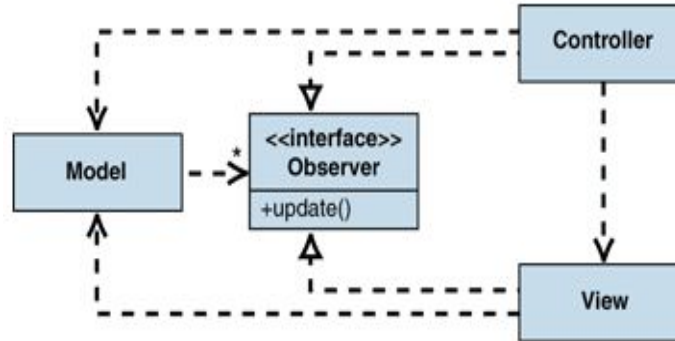


# Model-View-Controller (MVC)

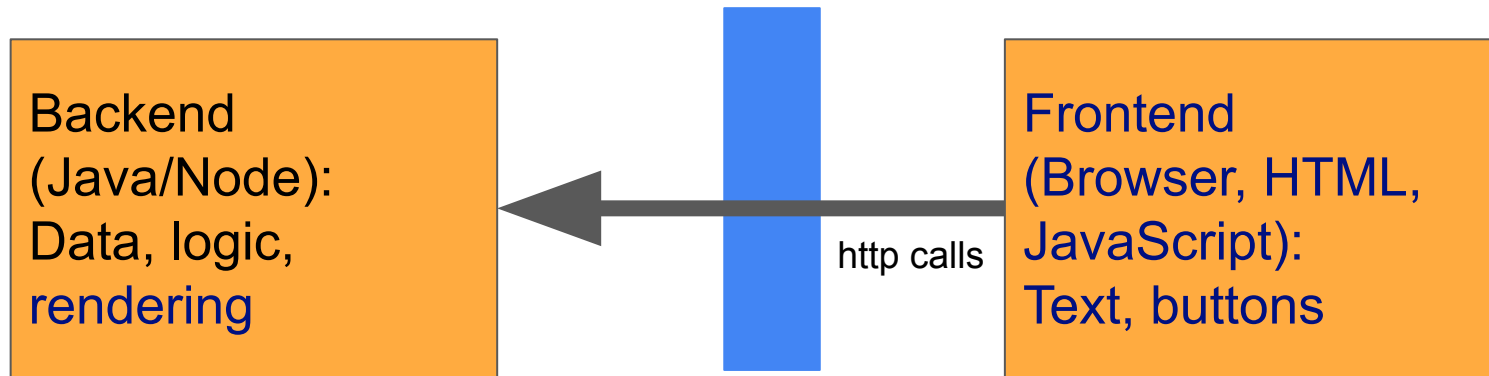
## Passive model



## Active model



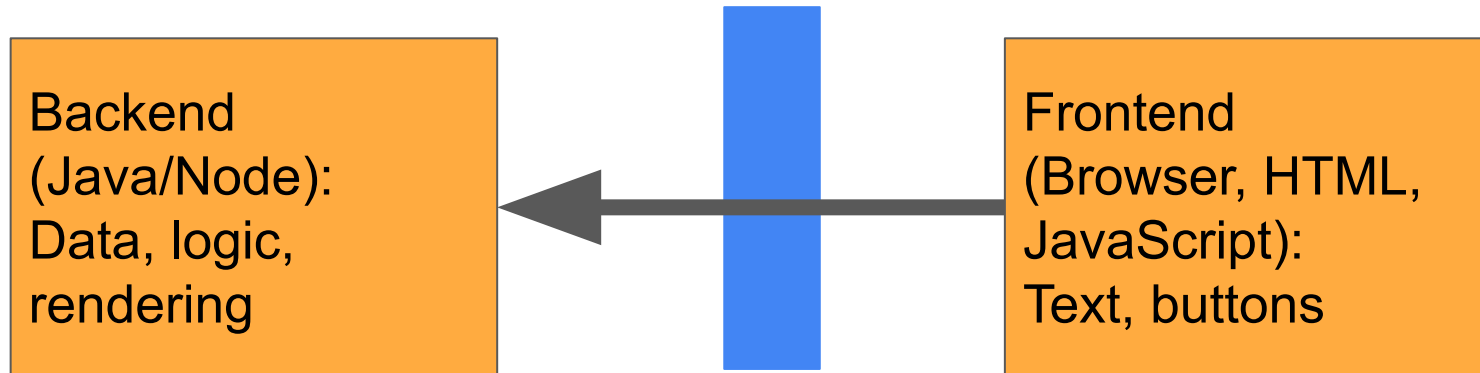
# GUI Code in the Backend



Typically there is GUI code in Backend (rendering/view)  
Could also send entire program state to frontend (e.g, json) and render there with JavaScript

# Where to put GUI Logic?

Example: Deactivate undo button in first round of TicTacToe,  
deactivate game buttons after game won



Option 1: All rendering in backend, update/refresh the entire page after every action -- simpler

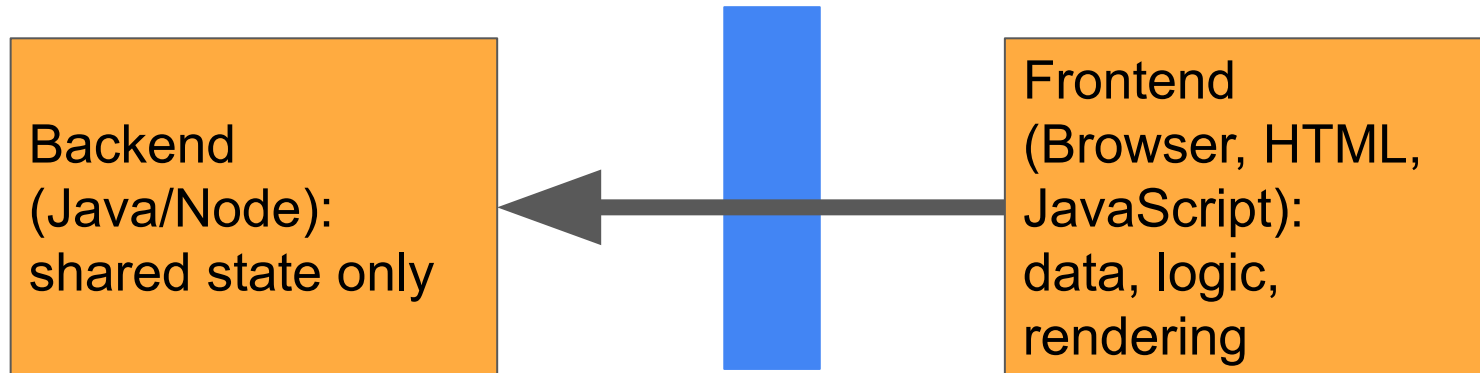
Option 2: Handle some logic in frontend, use backend for checking -- fewer calls, more responsive



# Core Logic in Frontend?

Could move core logic largely to client, minimize backend interaction

Can frontend be trusted? Need to replicate core in front and backend?



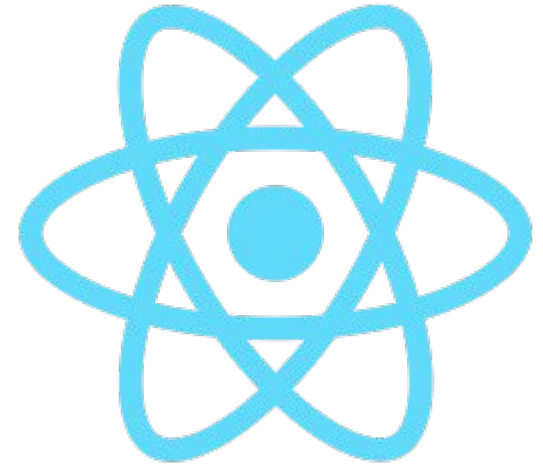
(React and other frameworks make it easy to introduce logic in the frontend; avoid tangling all core logic with GUI)

# ReactJS

# ReactJS

Popular frontend library by Facebook

Template library and state management



Not reactive programming library, though it adopts some similar ideas

# Templates with ReactJS

(Similar ideas to Handlebars in HW4 and Rec7)

Describe rendering of HTML, inputs given as objects

JSX language extension to embed HTML in JS

Try it:

<https://reactjs.org/redirect-to-codepen/introducing-jsx>

17-214/514

```
function formatName(user) {
  return user.firstName + ' ' +
    user.lastName;
}

const user = {
  firstName: 'Harper',
  lastName: 'Perez'
};

const element = (
  <h1>Hello, {formatName(user)}!</h1>
);

ReactDOM.render(
  element,
  document.getElementById('root')
);
```

# Composing Templates

(Corresponds to Fragments in  
Handlebars)

Nest templates

Pass arguments (properties)  
between templates

Try it:

<https://reactjs.org/redirect-to-codepen/components-and-props/composing-components>

17-214/514

```
function Welcome(props) {  
  return <h1>Hello, {props.name}</h1>;  
}  
  
function App() { return (  
  <div>  
    <Welcome name="Sara" />  
    <Welcome name="Edite" />  
  </div>  
);}  
  
ReactDOM.render(  
  <App />,  
  document.getElementById('root')  
);
```

# Templates with State

Class notation instead of function

State is like a cell in reactive programming, *if state changes, page is re-rendered*

Try it:

<https://codepen.io/gaearon/pen/xEmzGg?editors=0010>

```
class Toggle extends React.Component {
  constructor(props) {
    super(props);
    this.state = {isToggleOn: true};
    this.handleClick = this.handleClick.bind(this);
  }
  handleClick() {
    this.setState(prevState => ({
      isToggleOn: !prevState.isToggleOn
    }));
  }
  render() { return (
    <button onClick={this.handleClick}>
      {this.state.isToggleOn ? 'ON' : 'OFF'}
    </button>
  ); }
}

ReactDOM.render(
  <Toggle />,
  document.getElementById('root')
);
```

# ReactJS Templates

Can use arbitrary JavaScript code (Handlebars can only access object properties)

Properties are read-only

State is mutable and *observed* for re-rendering (state updates are asynchronous)

Re-rendering is optimized and asynchronous, will rerender inner components too if their properties change

# ReactJS and Core Logic

React makes it easy to add functionality in GUI

This really tangles GUI and logic (violating separation argued for above)

Suggestion: Use React state primarily for UI-related logic (e.g., selecting workers) and keep the core logic in the backend or as a separate library -- be very explicit about what information is shared



# Connecting React to backend

Return json from server backend and store as component state

Full example:

<https://www.freecodecamp.org/news/how-to-create-a-react-app-with-a-node-backend-the-complete-guide/>

```
function App() {  
  const [data, setData] =  
    React.useState(null);  
  React.useEffect(() => {  
    fetch("/api")  
      .then((res) => res.json())  
      .then((data) =>  
        setData(data.message));  
  }, []);  
  
  return (  
    <div>/* using state in data */</div>  
  );  
}
```

# React and Homework 5/6

Using React is entirely optional

We use Handlebars by default (HW4 and Rec7)

Many other template engines and frontend frameworks exists (e.g., Vue, Angular, ...)

React adds complexity but also easy updates reacting to state changes

# Summary

Immutable objects are great! Use them

Reactive programming decouples programs along data  
Observer pattern on steroids

New Design Pattern: Adapter

Decompose GUI from Core with Model View Controller Pattern

Brief intro to ReactJS