Principles of Software Construction: Objects, Design, and Concurrency

(Towards) Building Web-Apps

Jonathan Aldrich Bogdan Vasilescu **Matt Davis**

Carnegie Mellon University
School of Computer Science

S3D
Software and Societal Systems Department

Administrative

- No quiz today

- HW4 (UI) and HW5 (God Cards) are released!
  - HW4 due Fri 3/17
  - HW5 due Mon 3/27
  - Both include some challenges for extra points!

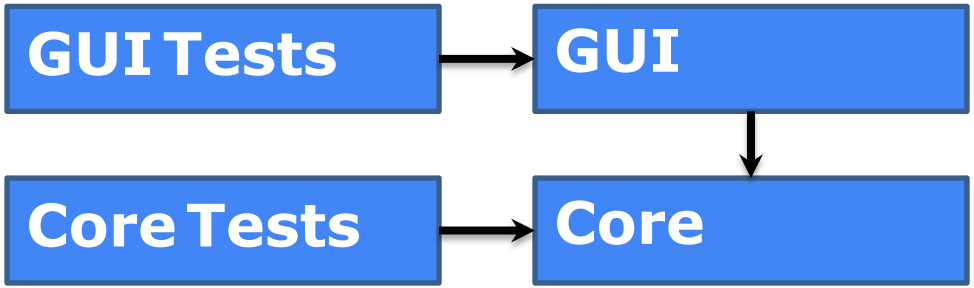
- Today we will be talking about Web App GUIs

Today

- Deeper into decoupling the front-end/GUI and back-end/logic
  - Architectural Pattern: Model-View-Controller
  - How to Web-App
  - ReactJS & Templates
- Concurrency: Into the abyss
  - A gentle introduction to asynchrony
  - Communication via callbacks
  - Threading in JS

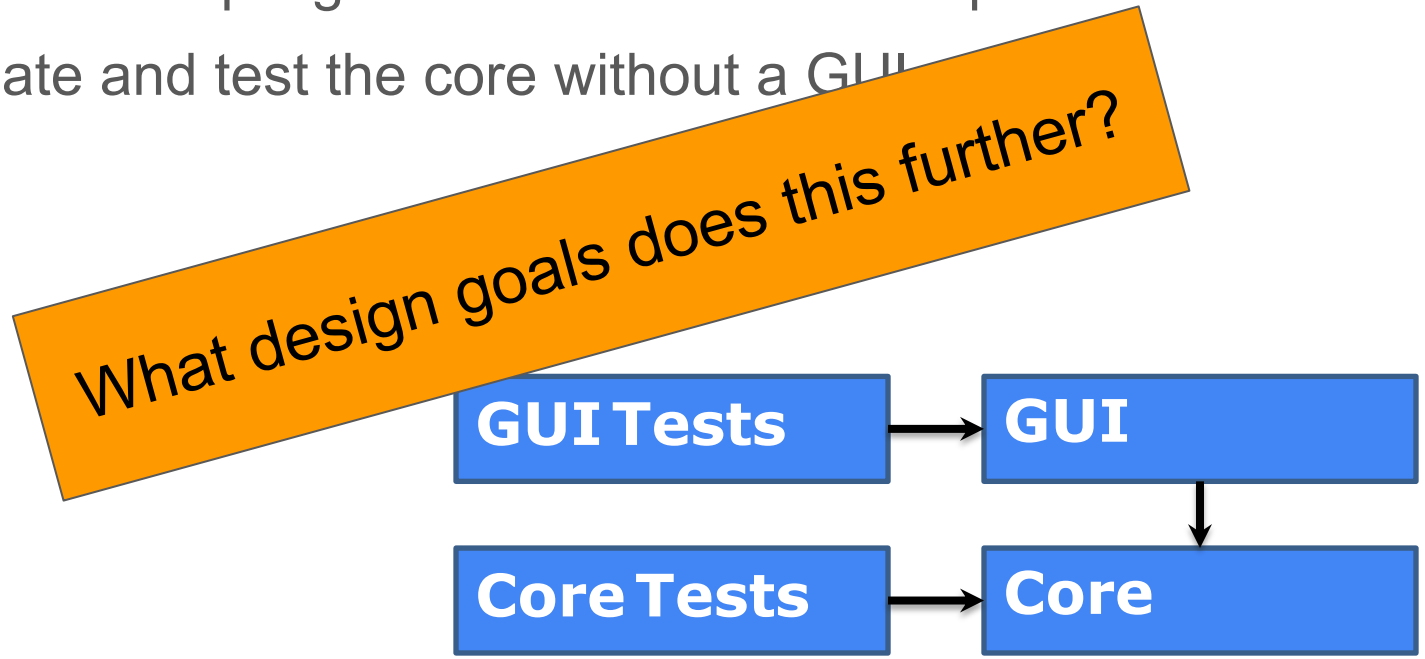
Recall: Separating application core and GUI

- Reduce coupling: do not allow core to depend on UI
- Create and test the core without a GUI



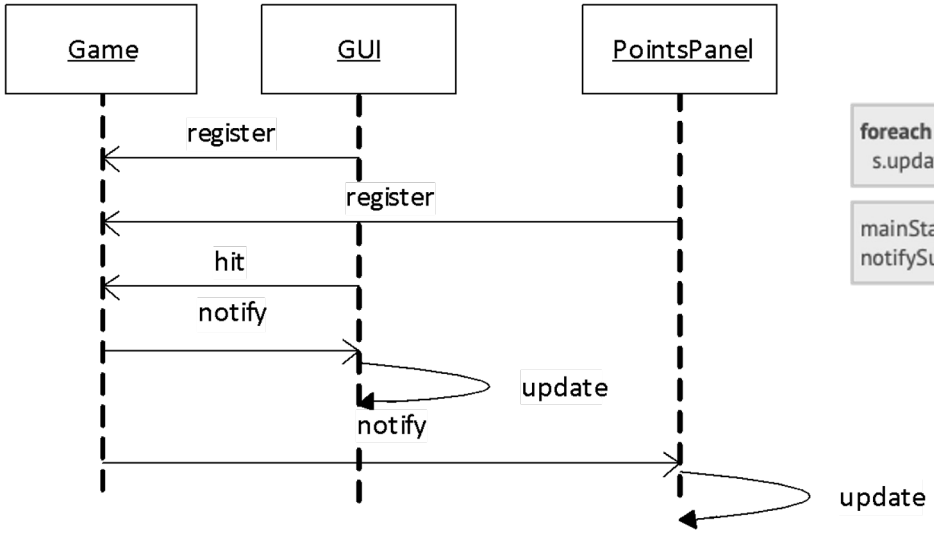
Recall: Separating application core and GUI

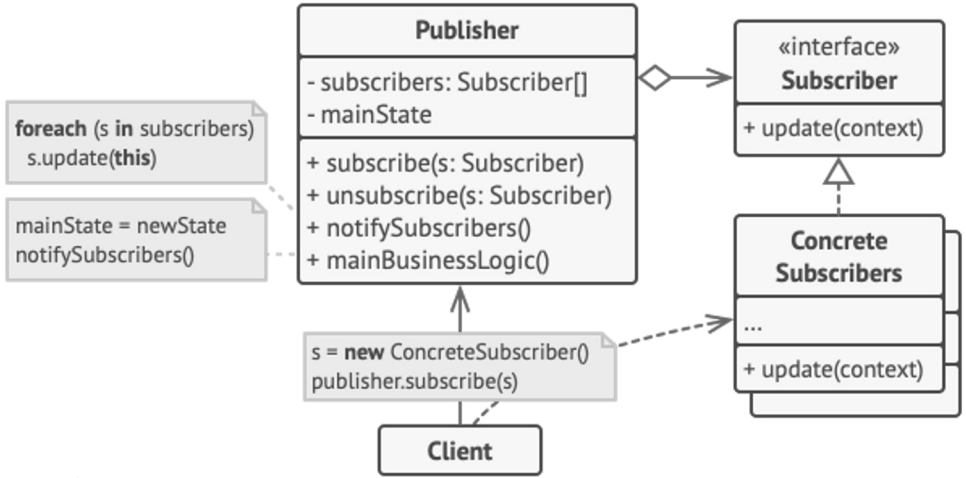
- Reduce coupling: do not allow core to depend on UI
- Create and test the core without a GUI



Recall: Single-Page yet Decoupled TicTacToe

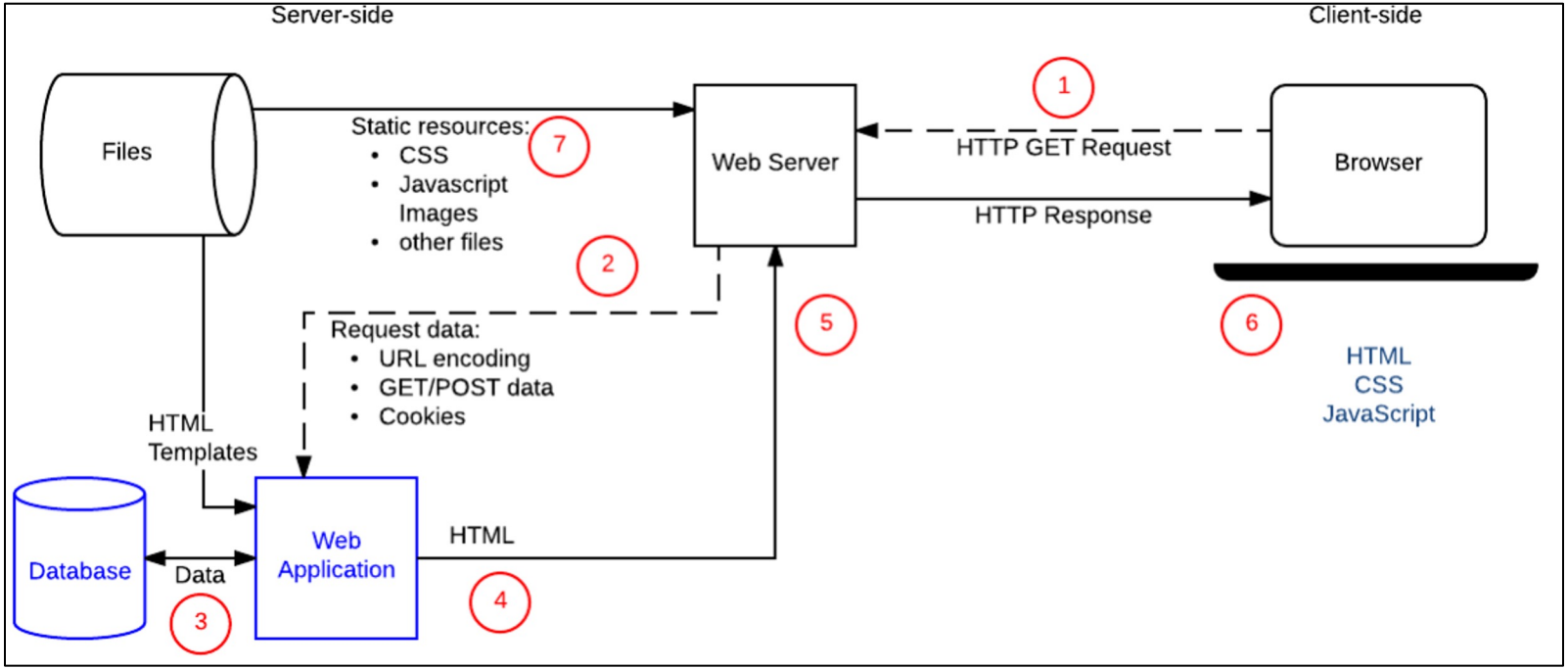
- Let the Game tell *all* interested components about updates
  - Use the Observer pattern to facilitate communication while preserving decoupling





https://refactoring.guru/design-patterns/observer

Recall: Client/Server



https://developer.mozilla.org/en-US/docs/Learn/Server-side/First_steps/Client-Server_overview#anatomy_of_a_dynamic_request

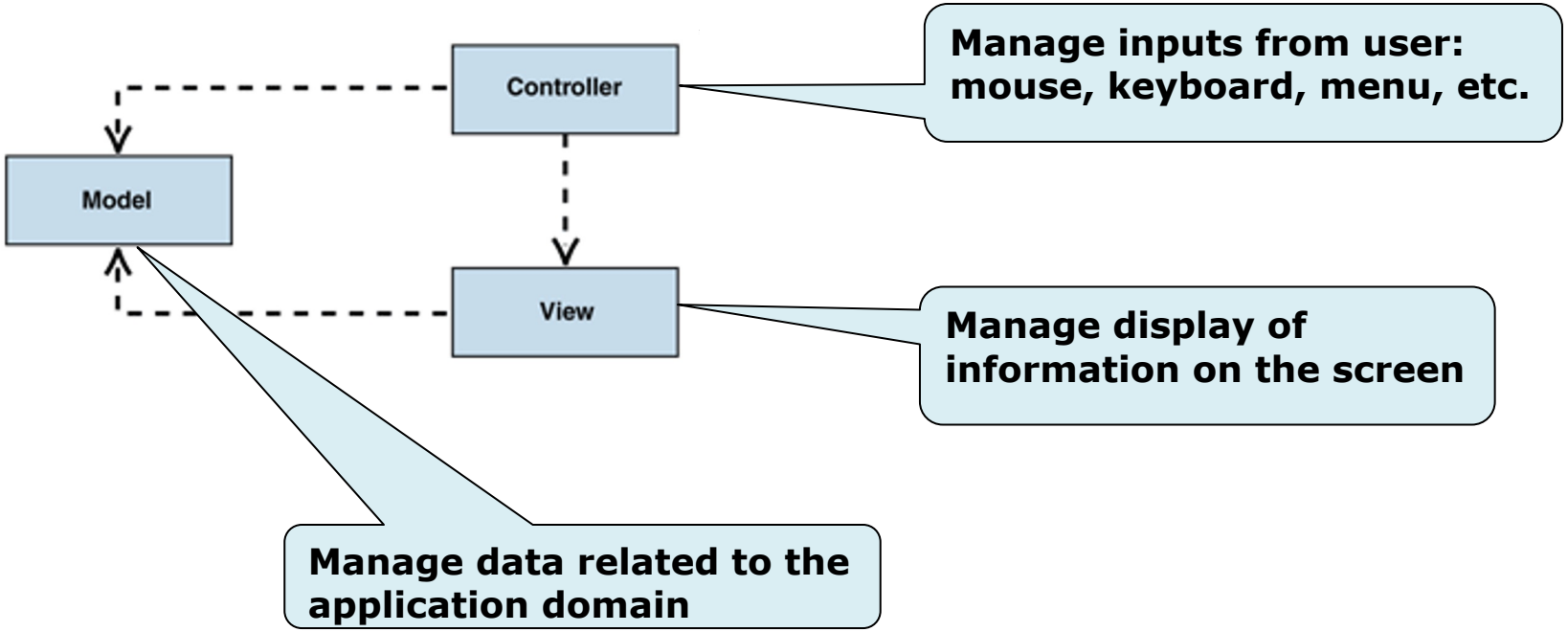
Recall: Client/Server TicTacToe

- TicTacToe with TS-Express
  - Two folders: ‘src’ and ‘views’
  - ‘views’ contains a *template* file
  - ‘src’ contains a *server* → and a *game*.
- The game knows nothing about the UI
  - Nor does the UI talk to the game
  - The server decouples them

```
75  function renderPage(res: Response<any, Record<string, any>, number>) {
76      res.render("main", genPage());
77  }
78
79  app.get("/newgame", (req, res) => {
80      startNewGameClicked()
81      renderPage(res)
82  });
83
84  app.get("/play", (req, res) => {
85      if (req.query.x && req.query.y)
86          clickCell(parseInt(req.query.x as string), parseInt(req.query.y as string))
87      renderPage(res)
88  });
89
90
91  app.get("/", (req, res) => {
92      renderPage(res)
93  });
94
95
96
97  // start the Express server
98  app.listen(port, () => {
99      console.log(`server started at http://localhost:${port}`);
100 });
```

Notice how we've begun to more explicitly separate out the HTML from the logic.

An architectural pattern: Model-View-Controller (MVC)



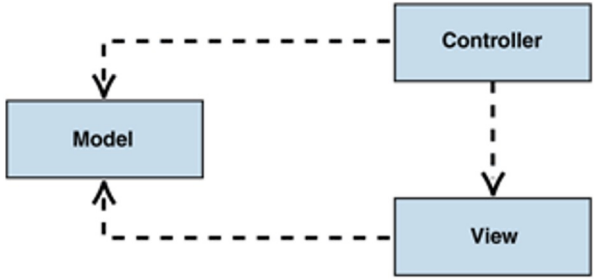
MVC is ubiquitous

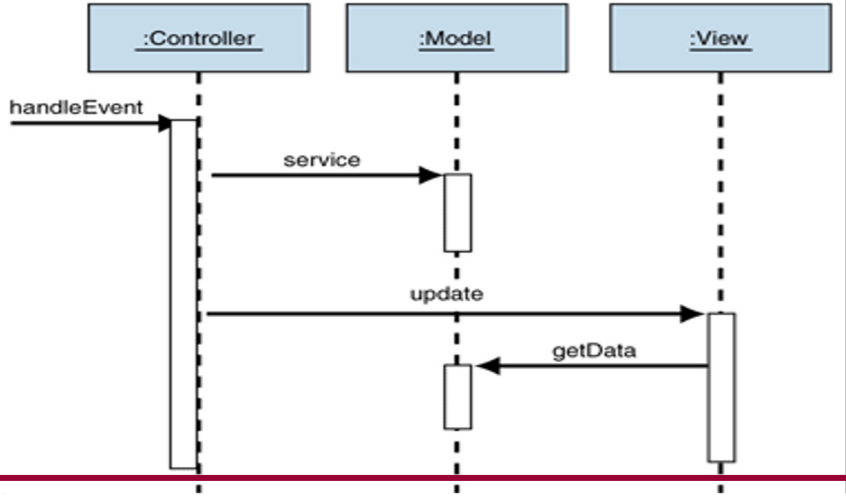
Separates:

- Model: data organization
  - Interface to the database
- View: visual representation (typically HTML)
  - Often called *templates* in web-dev; “view” is a bit overloaded
- Controller: intermediary between client and model/view
  - Typically asks *model* for data, *view* for HTML

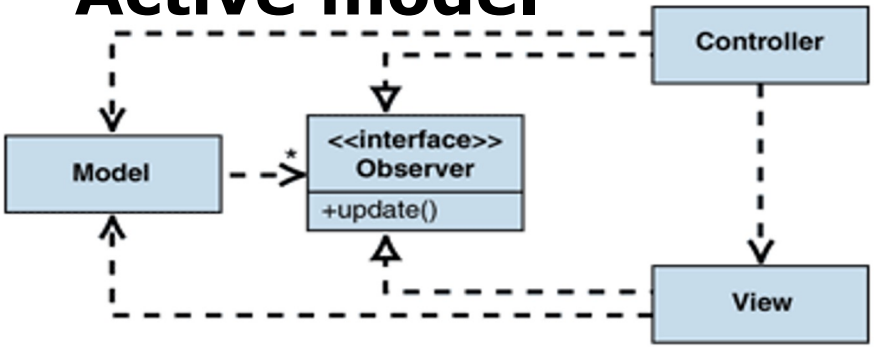
Model-View-Controller (MVC)

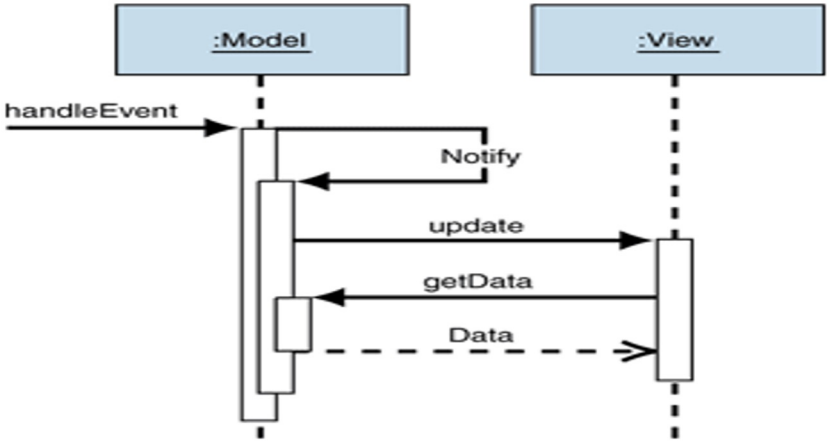
Passive model





Active model



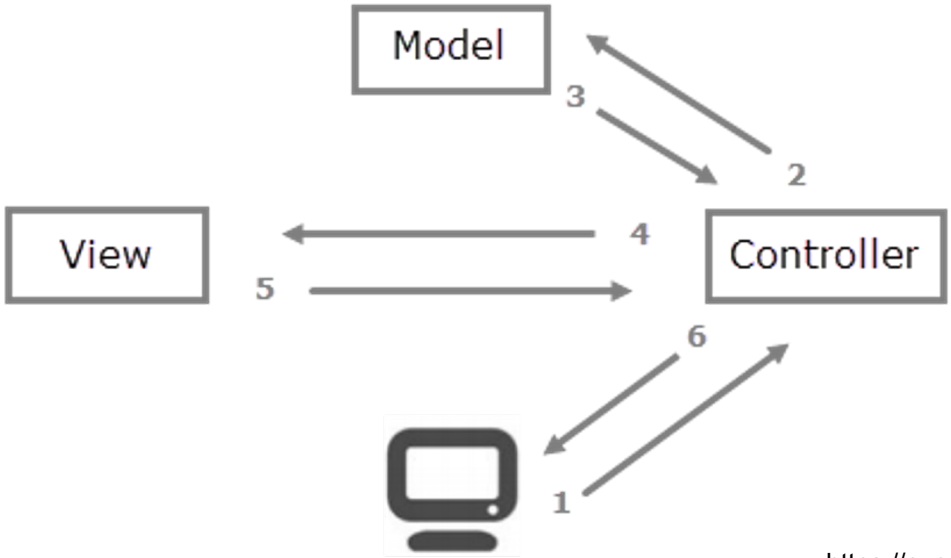


http://msdn.microsoft.com/en-us/library/ff649643.aspx

Model-View-Controller in TicTacToe?

Let's return to the ts-express version

+ talk about how the view gets updated



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

Web Apps are Applications Served via the Web

- Obvious, I know
- The key challenge: can't run everything on the client. Instead:
  - Multiple "tiers": presentation (front-end), logic/application (server), data (e.g., DB) layers.
    - MVC is a popular choice for how to connect these
    - Other ways to distribute these layers exist – we'll talk about a few soon
    - More tiers are possible too; out of scope for this class
  - Front-end/back-end separation via a communication layer
    - Which creates fun communication problems – more later.

Updating the View (or: How to Web App?)

- Let's avoid generating HTML from scratch on every call
  - Map requests to handler code
    - Fetch data, process
  - Generate and return HTML
    - Often processed using a template library

```
58      <div id="board">
59        {{#each cells}}
60          {{#if link}}
61            <a href={{link}}><div class="cell {{class}}">{{text}}</div></a>
62          {{else}}
63            <div class="cell {{class}}">{{text}}</div>
64          {{/if}}
65        {{/each}}
66      </div>
```

How to Web App?

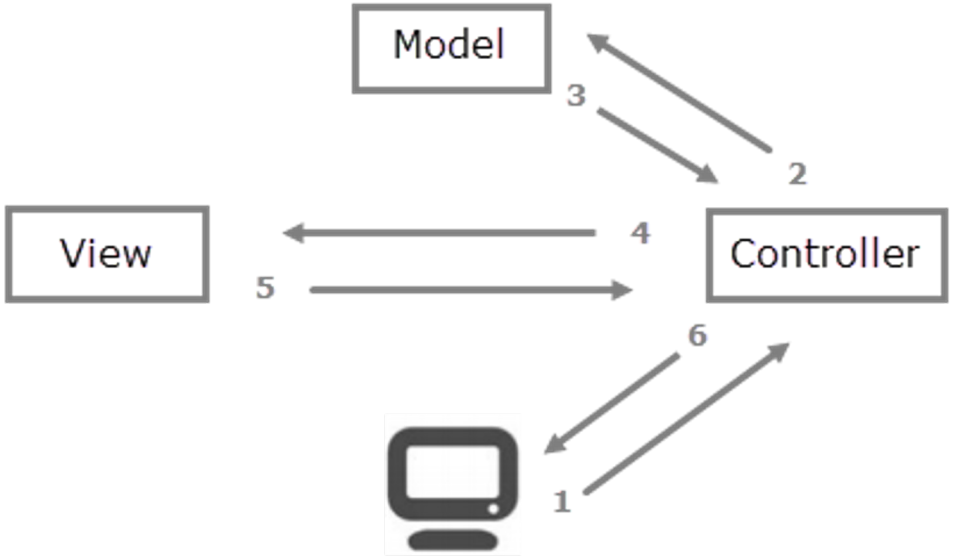
- Let’s avoid generating HTML from scratch on every call
  - Map requests to handler code
    - Fetch data, process
  - Generate and return HTML
- Historically: PHP
  - Modifies HTML pages server-side on request; strong ties to SQL

```php
<?php
  // The global $_POST variable allows you to access the data sent with the POST method by name
  // To access the data sent with the GET method, you can use $_GET
  $say = htmlspecialchars($_POST['say']);
  $to  = htmlspecialchars($_POST['to']);

  echo  $say, ' ', $to;
?>
```

How to Web App?

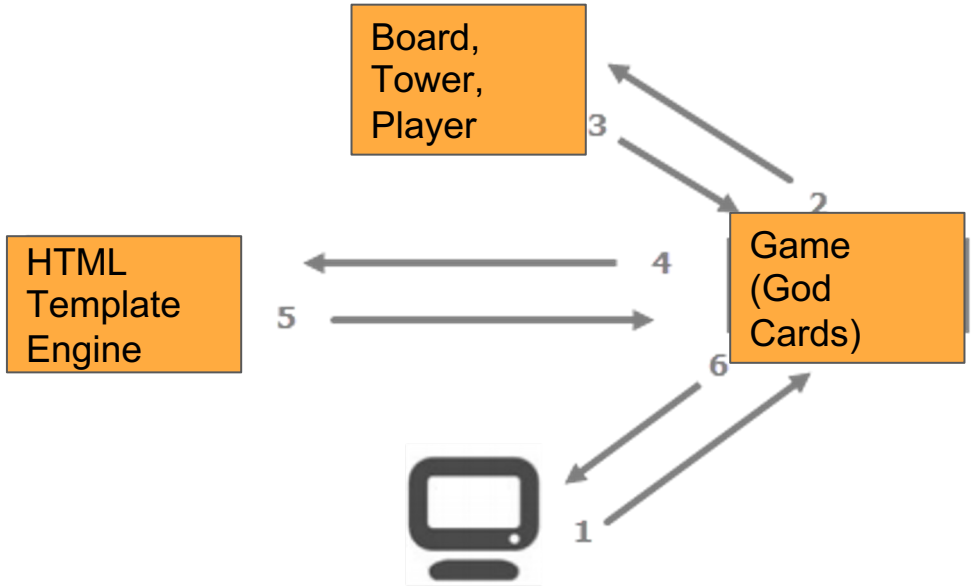
- Let’s avoid generating HTML from scratch on every call
  - Map requests to handler code
    - Fetch data, process
  - Generate and return HTML
- Or use a framework
  - Python: Flask, Django
  - NodeJS: Express
  - Spring for Java
  - Many others, differences in **weight**, features
  - React.js

Model View Controller in Santorini?



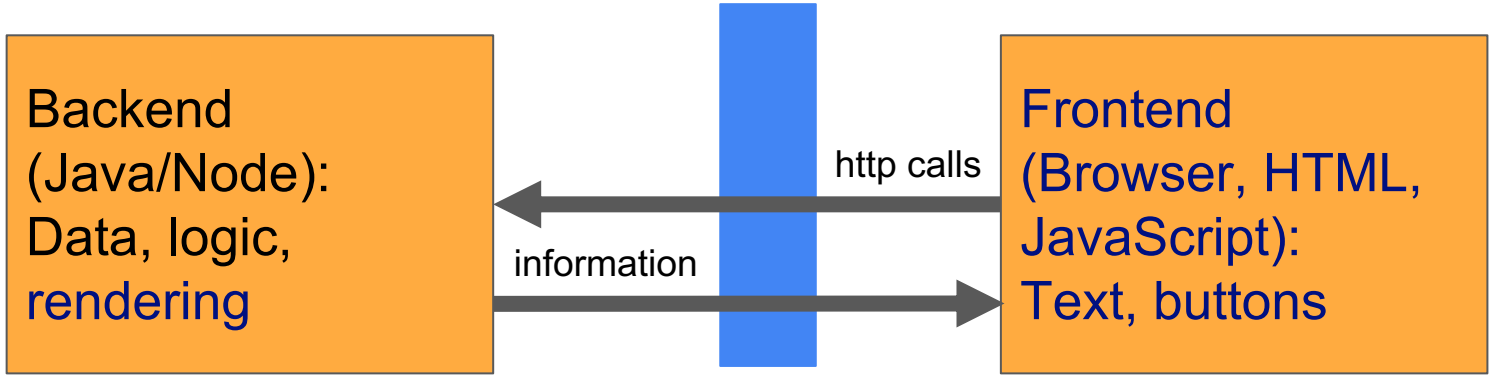
https://overiq.com/django-1-10/mvc-pattern-and-django/

Model View Controller in Santorini



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

Client-Server Programming forces Frontend-Backend Separation

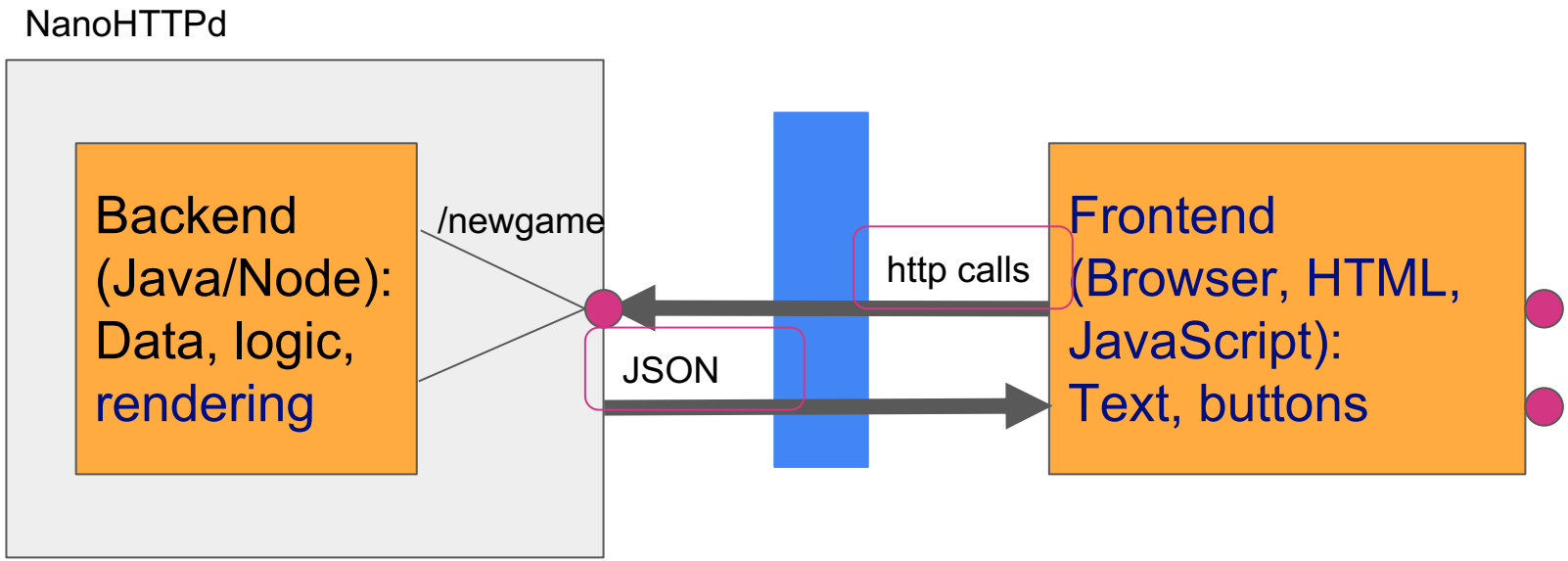


Browser can call web server, but not the other way around

Browser needs to *pull* for updates

Browser can request entire page, or just additional content (ajax, REST api calls, …)

TicTacToe


NanoHTTPd
Backend (Java/Node): Data, logic, rendering
/newgame
http calls
JSON
Frontend (Browser, HTML, JavaScript): Text, buttons

Some alternatives

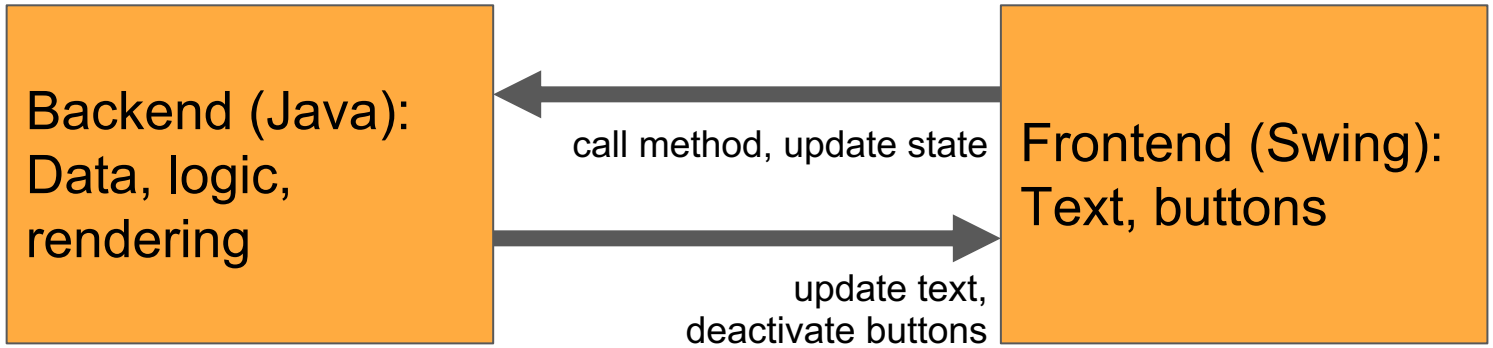
Is this needlessly complicated?

Core & Gui in same environment

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

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

Core logic could directly modify GUI

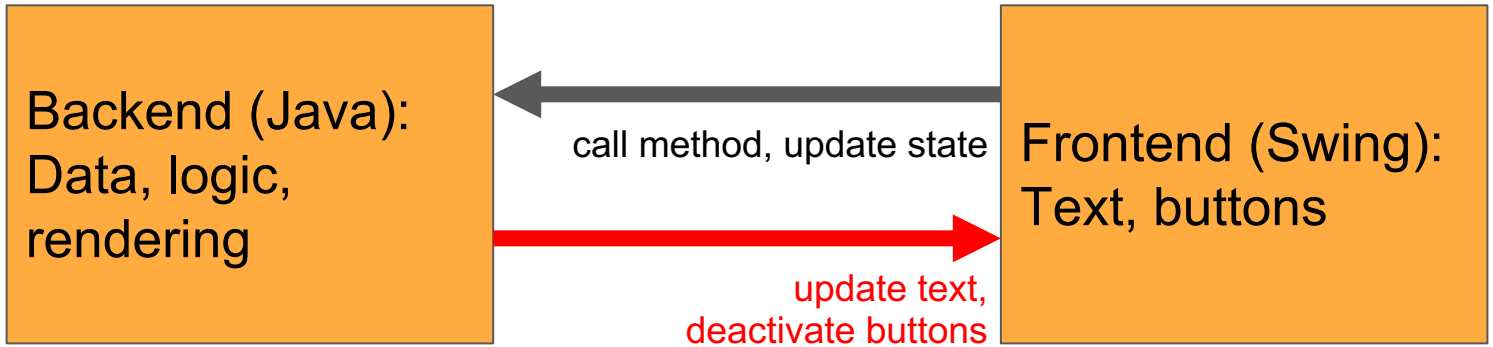


* WebWorkers do allow for *some* threading for longer-running tasks

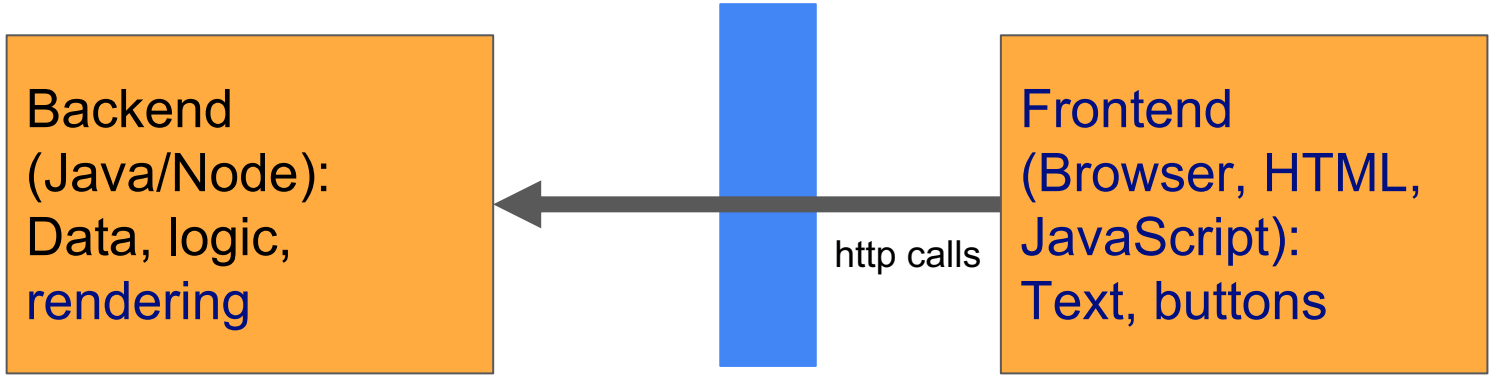
Avoid Core to Gui coupling

Never call the GUI from the Core

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



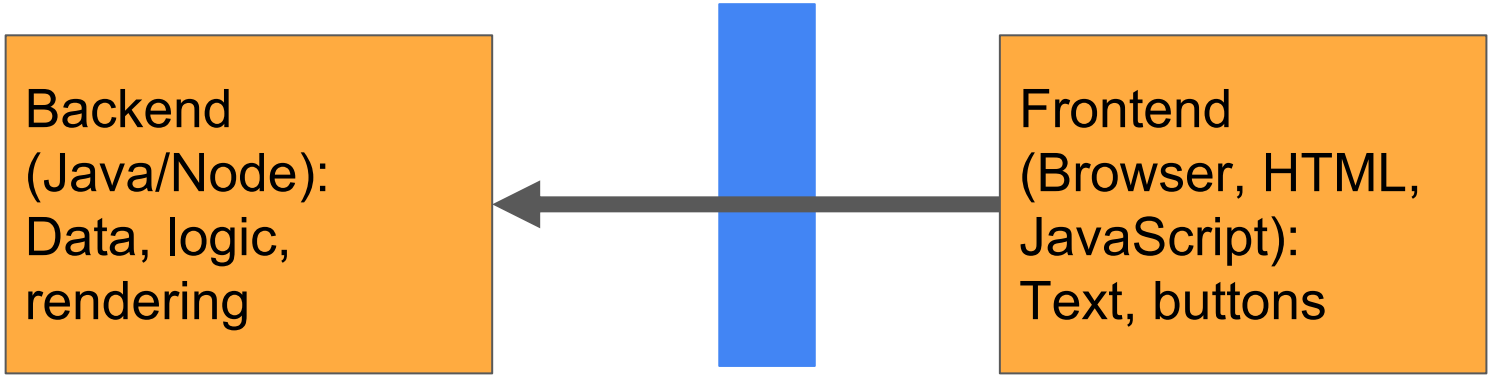
GUI Code in the Backend



Typically there is some 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

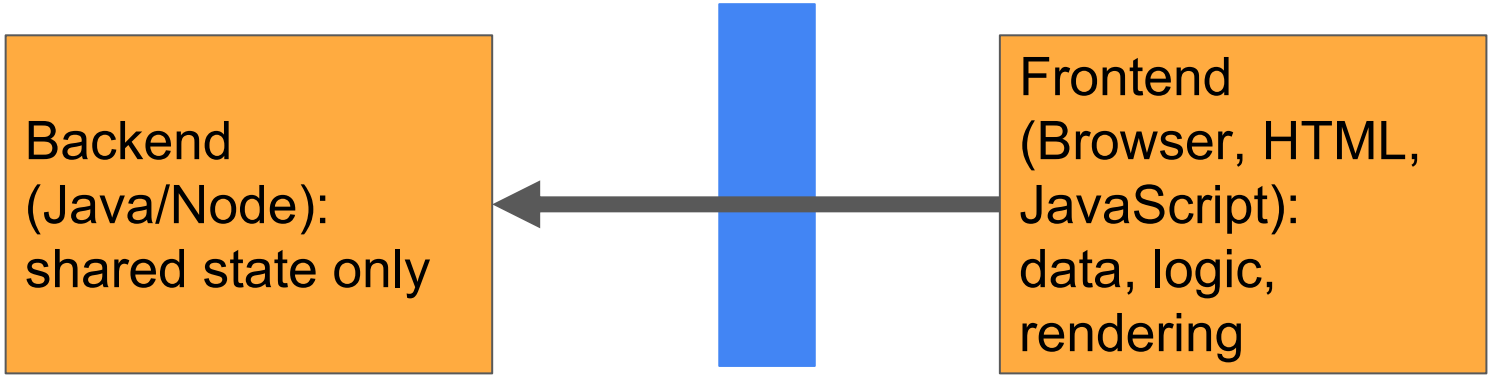


Option 1: All rendering in backend, update/refresh entire page after every action–simpler
Option 2: Some logic in frontend, use backend for checking–fewer calls, more responsive

Core Logic in Frontend?

Could move core logic largely to client, minimize backend interaction

Downside?

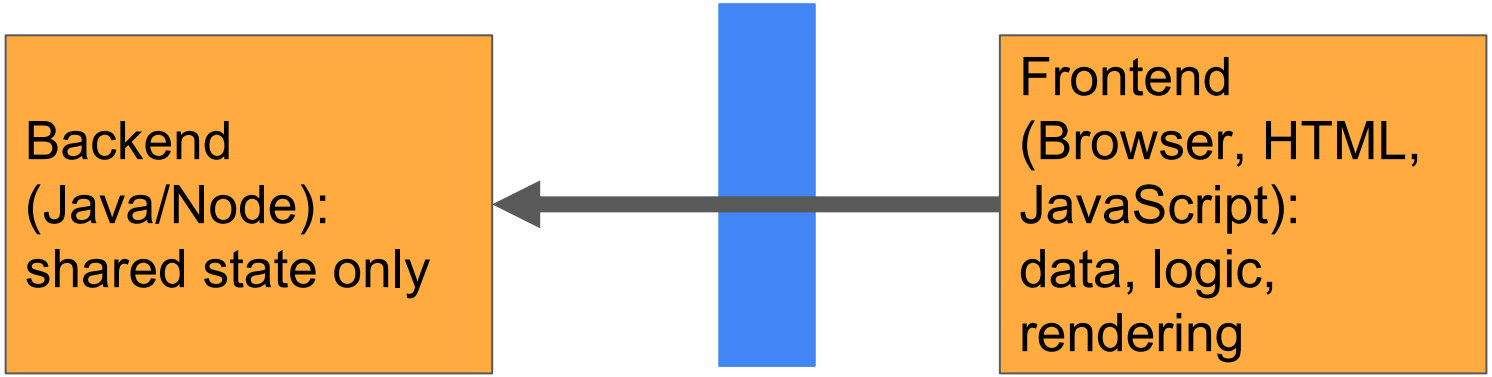


(React and other frameworks make it easy to introduce logic in the frontend; avoid tangling all core logic with GUI)

Core Logic in Frontend?

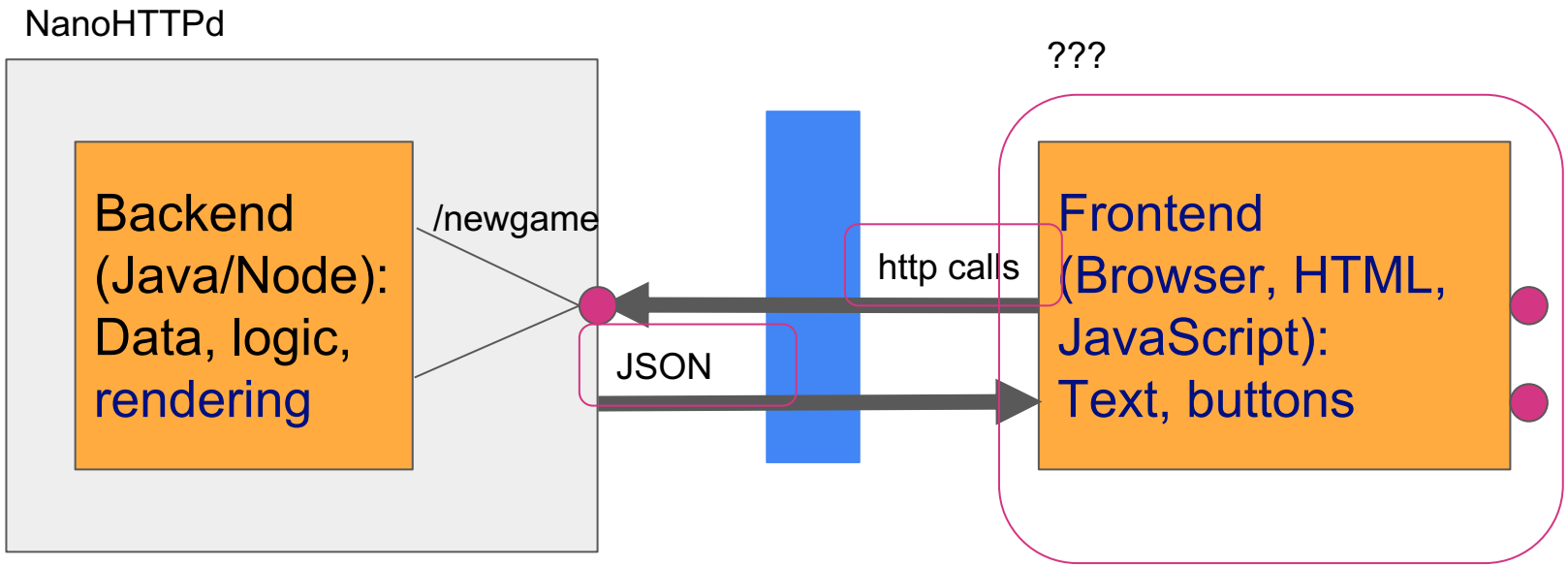
Could move core logic largely to client, minimize backend interaction

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

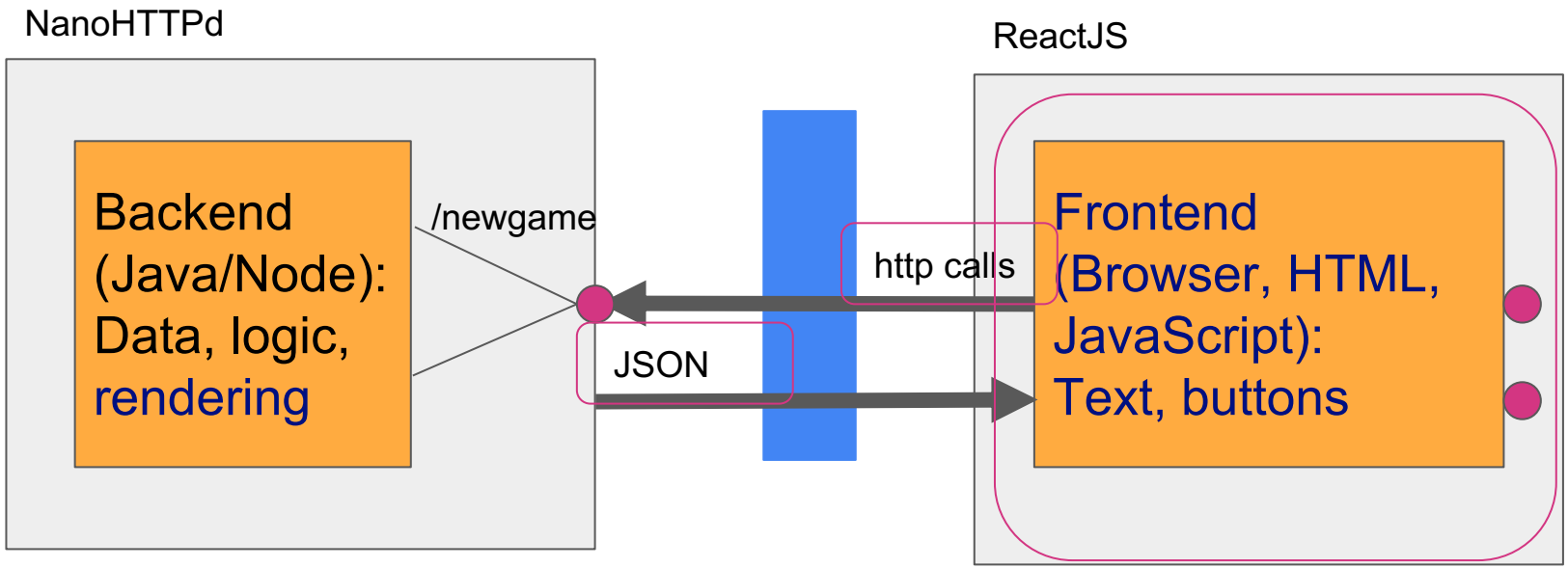


(React and other frameworks make it easy to introduce logic in the frontend; avoid tangling all core logic with GUI)

TicTacToe

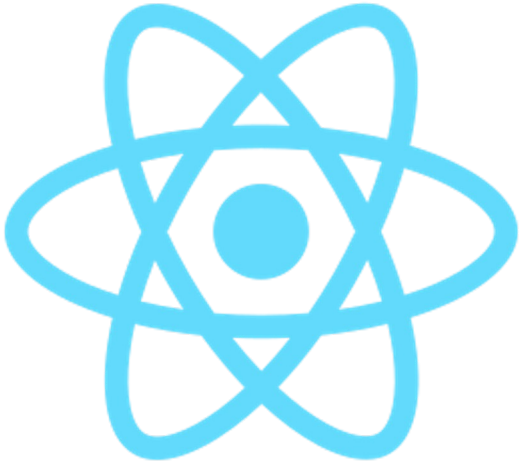

NanoHTTPd
Backend (Java/Node): Data, logic, rendering
/newgame
???
Frontend (Browser, HTML, JavaScript): Text, buttons
http calls
JSON

TicTacToe


NanoHTTPd
ReactJS
Backend
(Java/Node):
Data, logic,
rendering
/newgame
http calls
JSON
Frontend
(Browser, HTML,
JavaScript):
Text, buttons

ReactJS

ReactJS

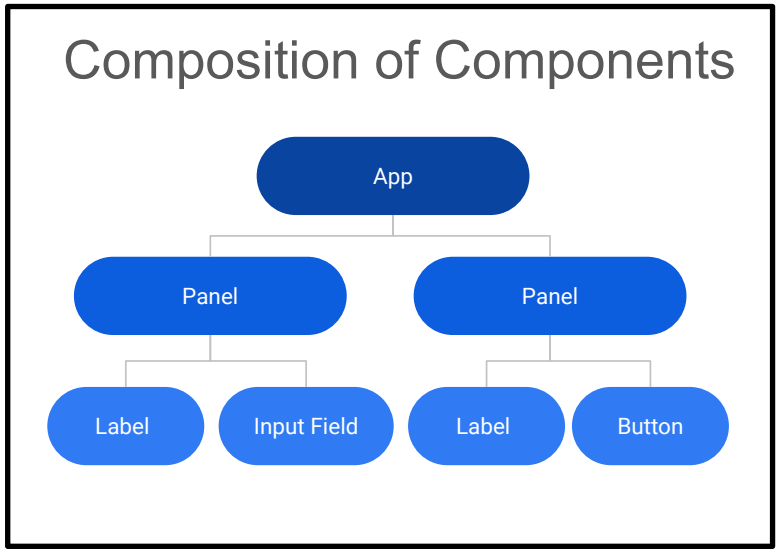


Popular frontend library by Meta / Facebook

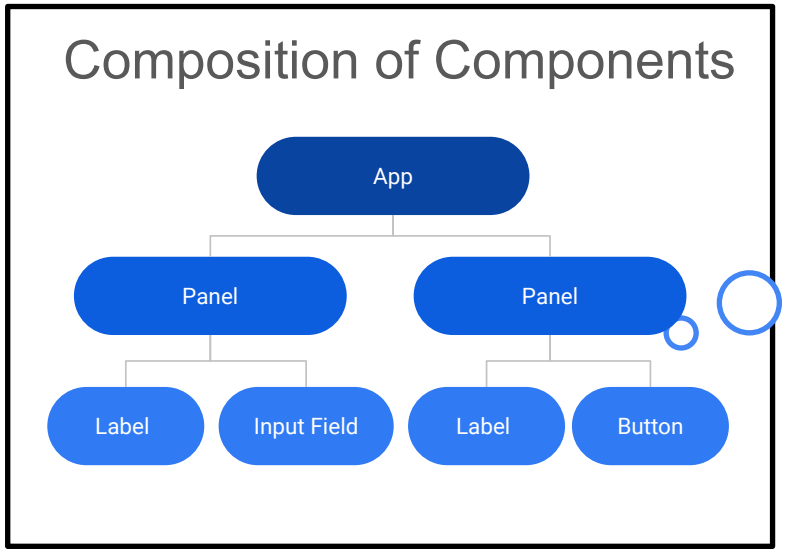
Template library and state management

(Not a reactive programming library, though it adopts some similar ideas – we'll get back to reactive programming)

ReactJS Overview

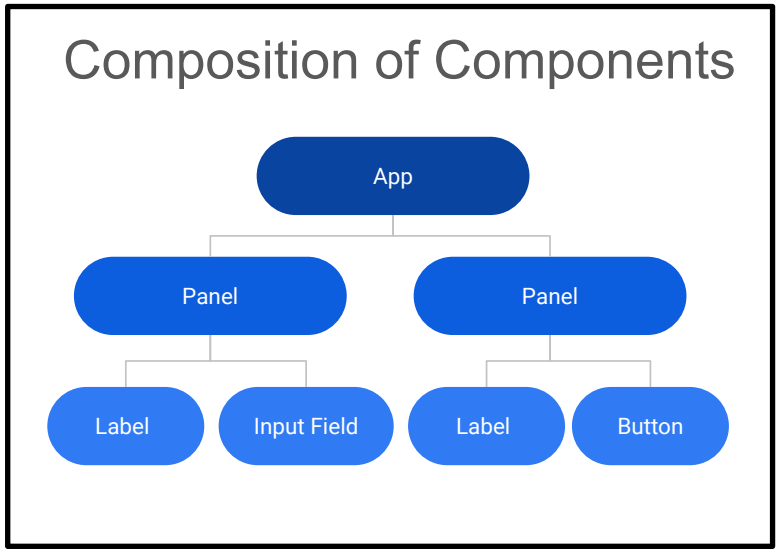


ReactJS Overview

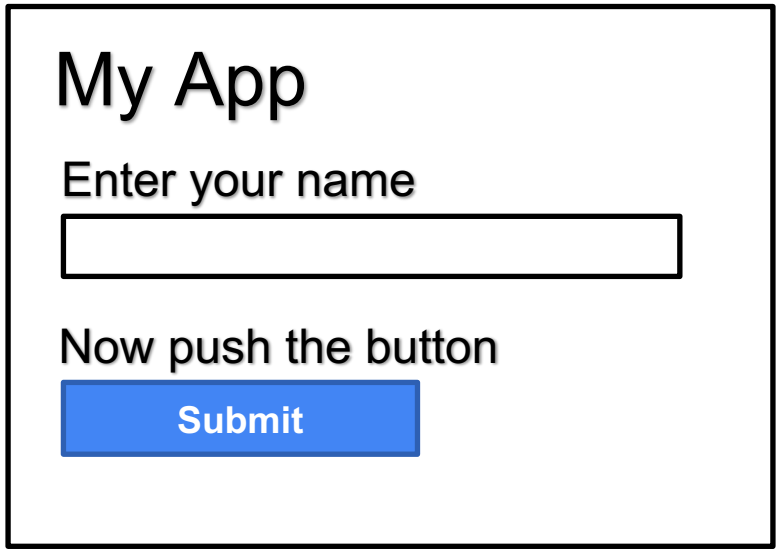


What design pattern does this remind you of?

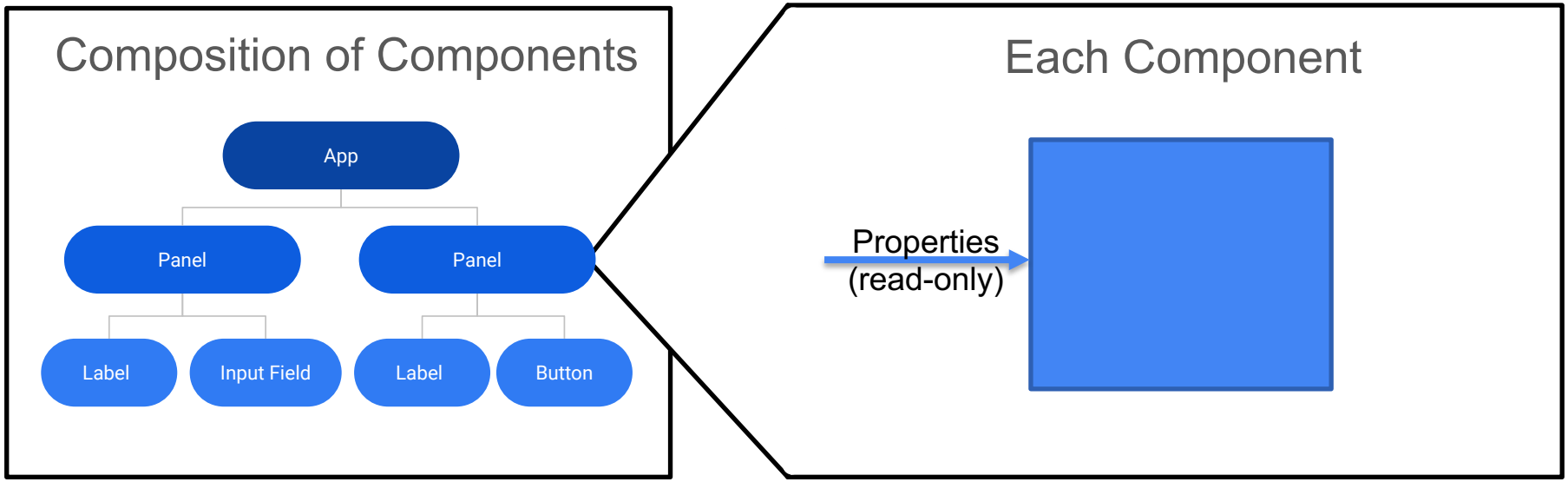
ReactJS Overview



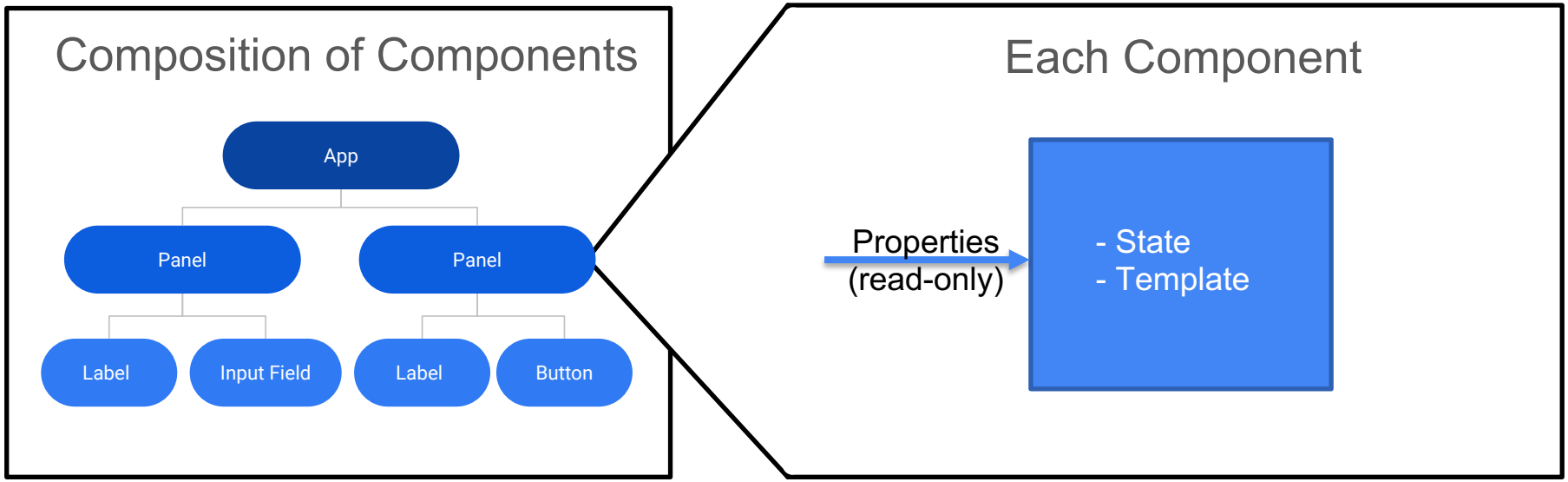
(Rendered Web Page)



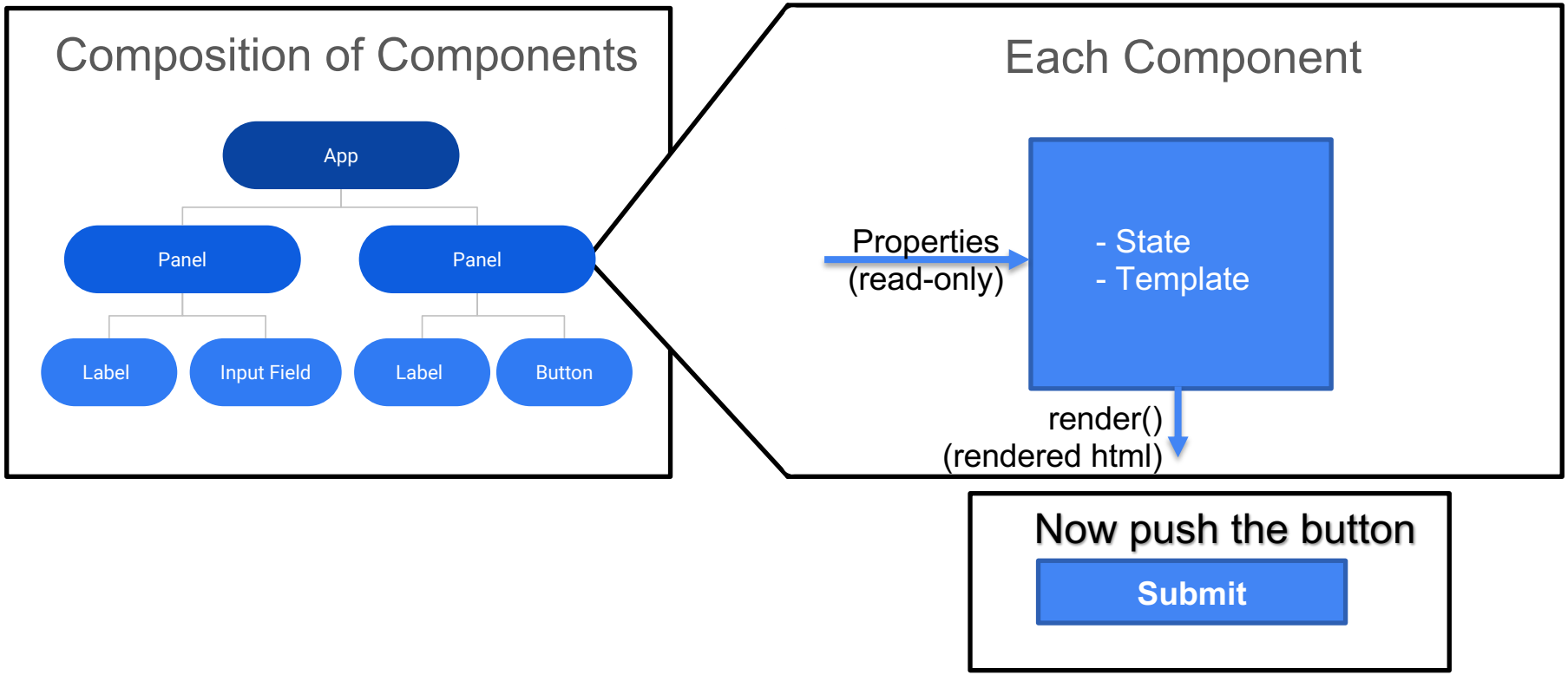
ReactJS Overview



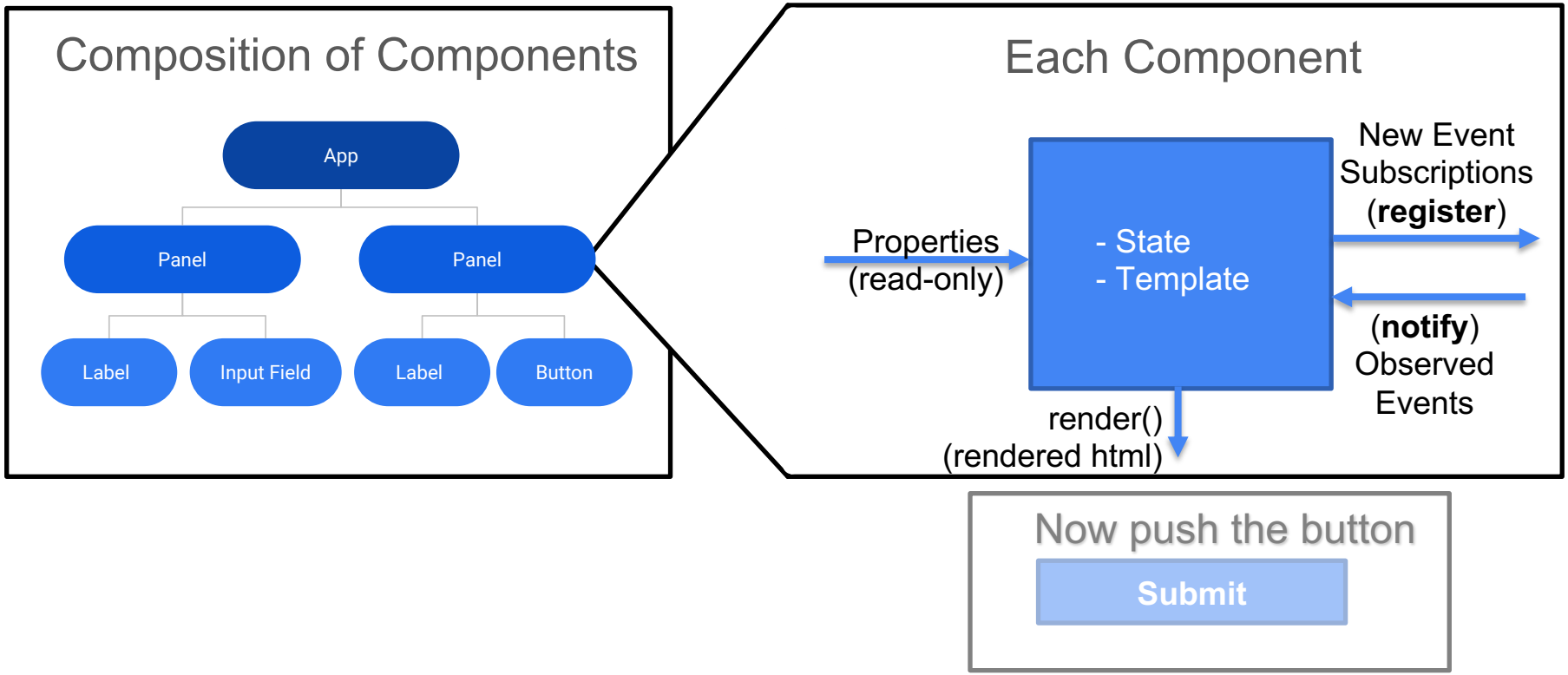
ReactJS Overview



ReactJS Overview

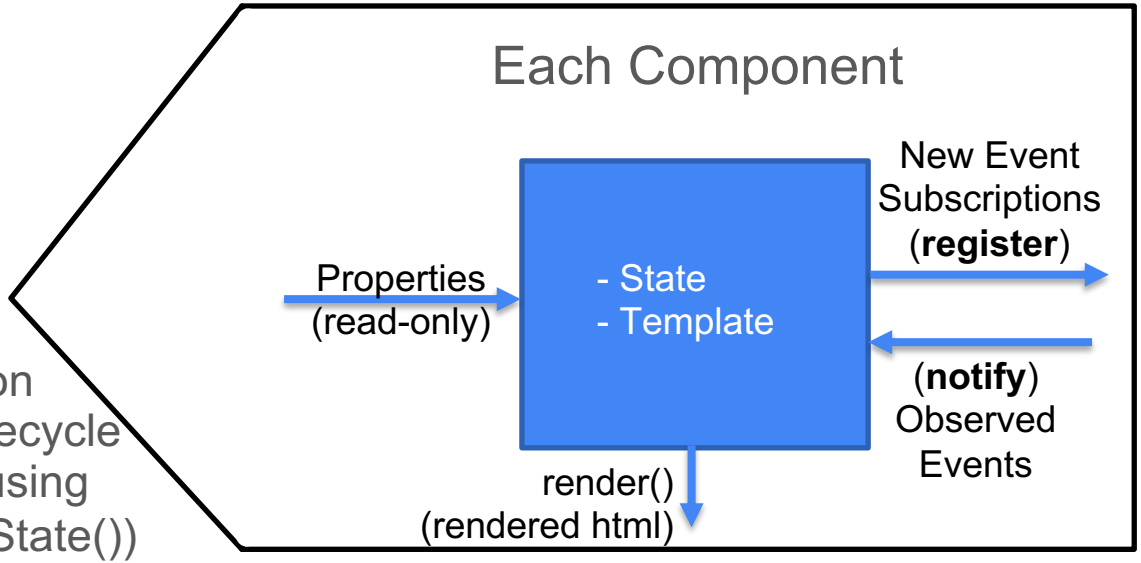


ReactJS Overview



Two Different ReactJS Component Types

- Class-based
  - Implemented as a class
  - Fully functional
  - Will use these tomorrow
- Function-based
  - Implemented as a function
  - Originally lacked state,lifecycle
  - But now fully-functional using React “hooks” (e.g., useState())



- Tomorrow in Recitation: You’ll use class-based components
- In the following slides we’ll use a mix of both component types

Components with ReactJS

Describe rendering of HTML, inputs given as objects

JSX language extension to embed HTML in JS

```
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')
);
```

Try it: https://reactjs.org/redirect-to-codepen/introducing-jsx

Composing Components

Nest templates

Pass arguments (properties) between templates

Try it: https://reactjs.org/redirect-to-codepen/components-and-props/composing-components

```
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')
);
```

Components with State

Class notation instead of function

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 Components

Can use arbitrary JavaScript code

Properties are read-only

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

Re-rendering is optimized and asynchronous, will re-render 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 previously)

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 Some Core

Use observer pattern to let react component observe changes

Encapsulate in *useEffect()* hook

Further discussion: https://reactjs.org/docs/hooks-custom.html

```
function App() {
  const [data, setData] =

        React.useState(null);
  React.useEffect(() => {
    function handleStatusChange(e) {
      setData(e.updatedData);
    }
    CoreAPI.subscribe(handleStatChange);
    return () => {
      CoreAPI.unsubscribe(handleStatChange);
    };
  });
  return (
```

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/

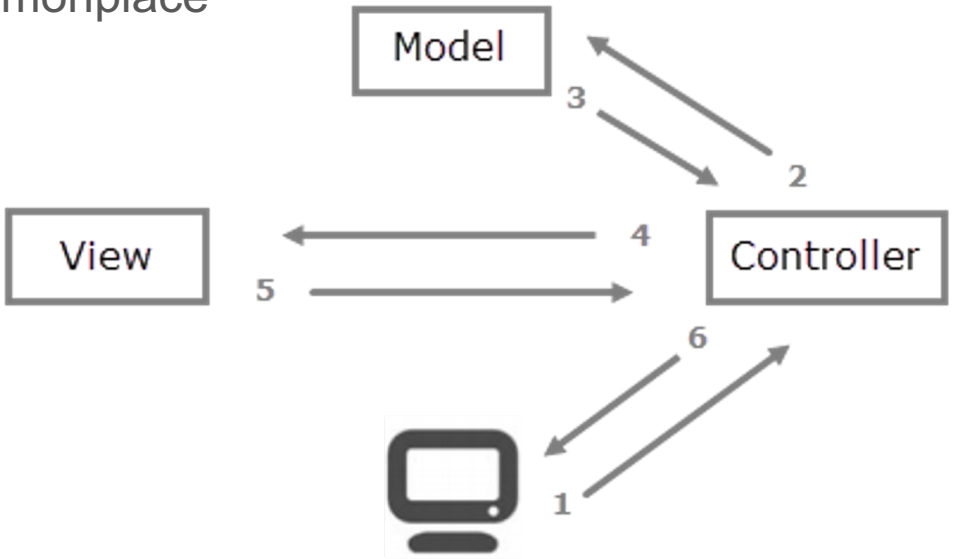
```
function App() {
  const [data, setData] =
                  React.useState(null);
  React.useEffect(() => {
    fetch("/api")
      .then((res) => res.json())
      .then((data) =>
                  setData(data.message));
  }, []);

  return (
    <div>/* using state in data */</div>
  );
}
```

For Homework 4

- You don't have to use a web app framework
  - The important thing is to decouple the GUI from the backend
  - There are many ways to do this
- We show you how to use React in Rec07
  - Many other template engines and frontend frameworks exists (e.g., Vue, Angular, …)
  - React adds complexity but also easy updates reacting to state changes
  - We show React.js because it is common today

Recapping: Where Are We?

- Decoupling improves design
  - MVC-like approaches are commonplace
- We've talked about:
  - Back-end: extensively
  - Front-end: last few classes
  - Controllers, servers
- What are we missing?



Principles of Software Construction: Objects, Design, and Concurrency

What have we not yet talked about? ^

Jonathan Aldrich Bogdan Vasilescu **Matt Davis**

Carnegie Mellon University
School of Computer Science

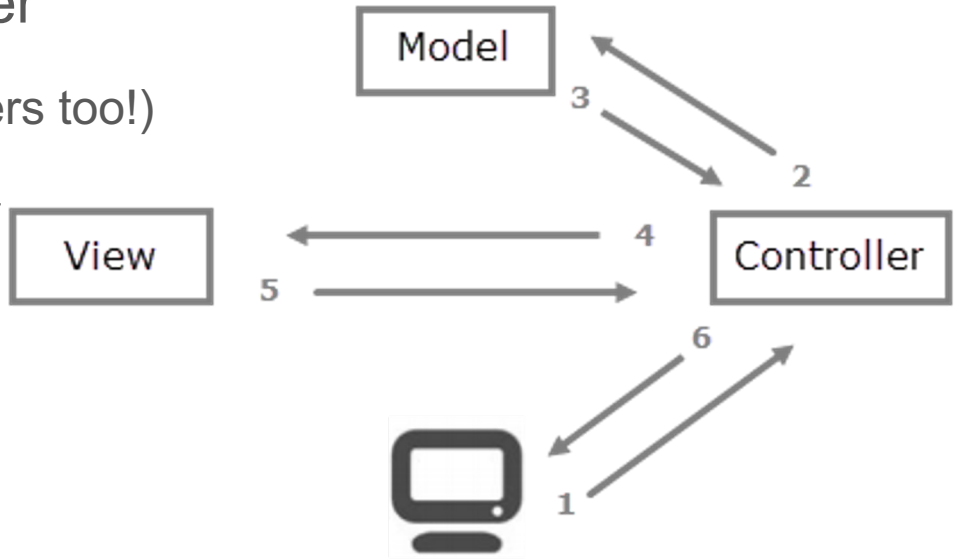
S3D
Software and Societal Systems Department

How Do We Talk?

These arrows hide a complicated truth:
The client is a separate computer

- (The server is often many computers too!)

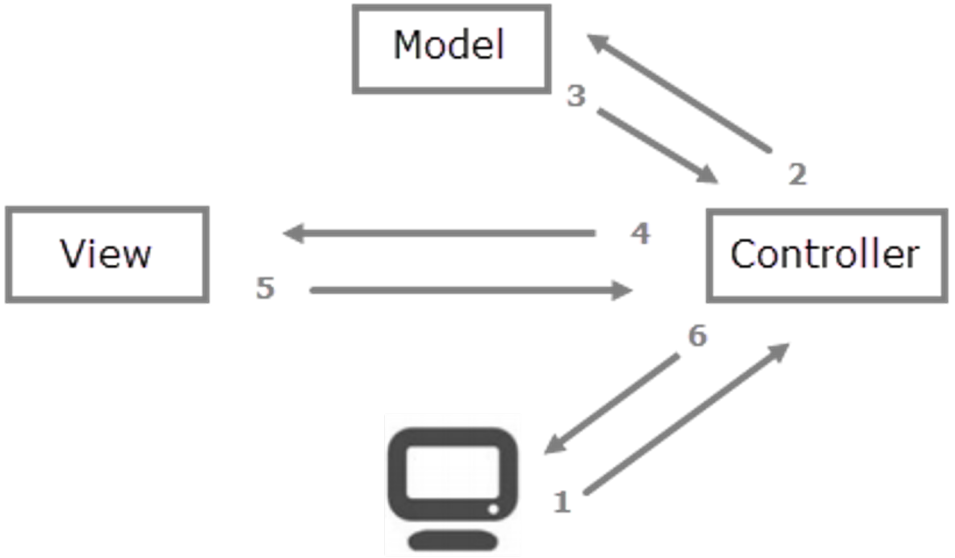
And talking to another computer is *hard*



How Do We Talk?

Talking to another computer is *hard*

- Why? We already covered HTTP (GET/POST), right?



Suppose Everything were Synchronous

That is, when we call something, we wait for the return, doing nothing until that happens. So HTTP is just like any other method call.

Let’s try that out!

Suppose Everything were Synchronous

Two demonstrations:

1. Active waiting – what happens to the webpage if a request takes a long time?
   a. Not great! Let's talk about *threading* next

Note: I'm not showing the code here because it is contrived

Suppose Everything were Synchronous

Two demonstrations:

1. Active waiting – what happens to the webpage if a request takes a long time?
   a. Not great! Let's talk about *threading* next
2. The alternative: allowing other execution to happen
   a. New and exciting problems :) Need to handle **concurrency**

Note: I'm not showing the code here because it is contrived

Asynchrony

- The general concept of things happening outside the main flow
  - Recall the start of this class: we don’t always control when things happen.
  - Nor do we want to wait for them

- We use an asynchronous method call:
  - Normally, when we need to do work away from the current application;
  - And we don't want to block our application while awaiting the response

Asynchrony in User Interfaces

What happens here:

```
document.addEventListener('click', () => console.log('Clicked!'))
```

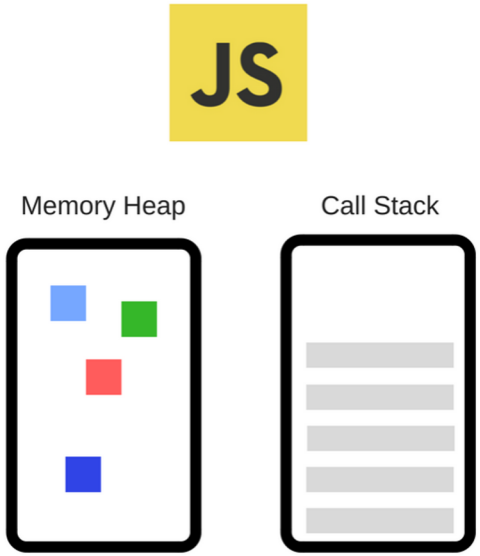
Asynchrony in User Interfaces

Callback functions

- Perhaps *the* building blocks of the internet's UI.
- Specifies work that should be done once something happens
  - Called asynchronously from the literal flow of the code
  - Not concurrent: JS is **single-threaded**

```
document.addEventListener('click', () => {
  console.log('Clicked!'); console.log('Clicked again!'); })
```

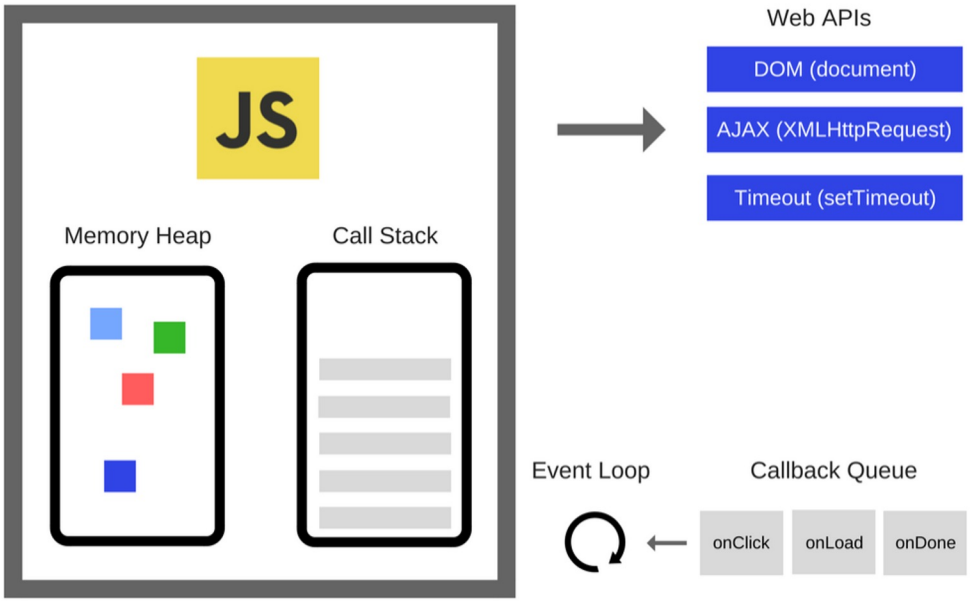
The JavaScript Engine (e.g., V8)



Two main components:

- Memory Heap — this is where the memory allocation happens
- Call Stack — this is where your stack frames are as your code executes

The JavaScript Runtime



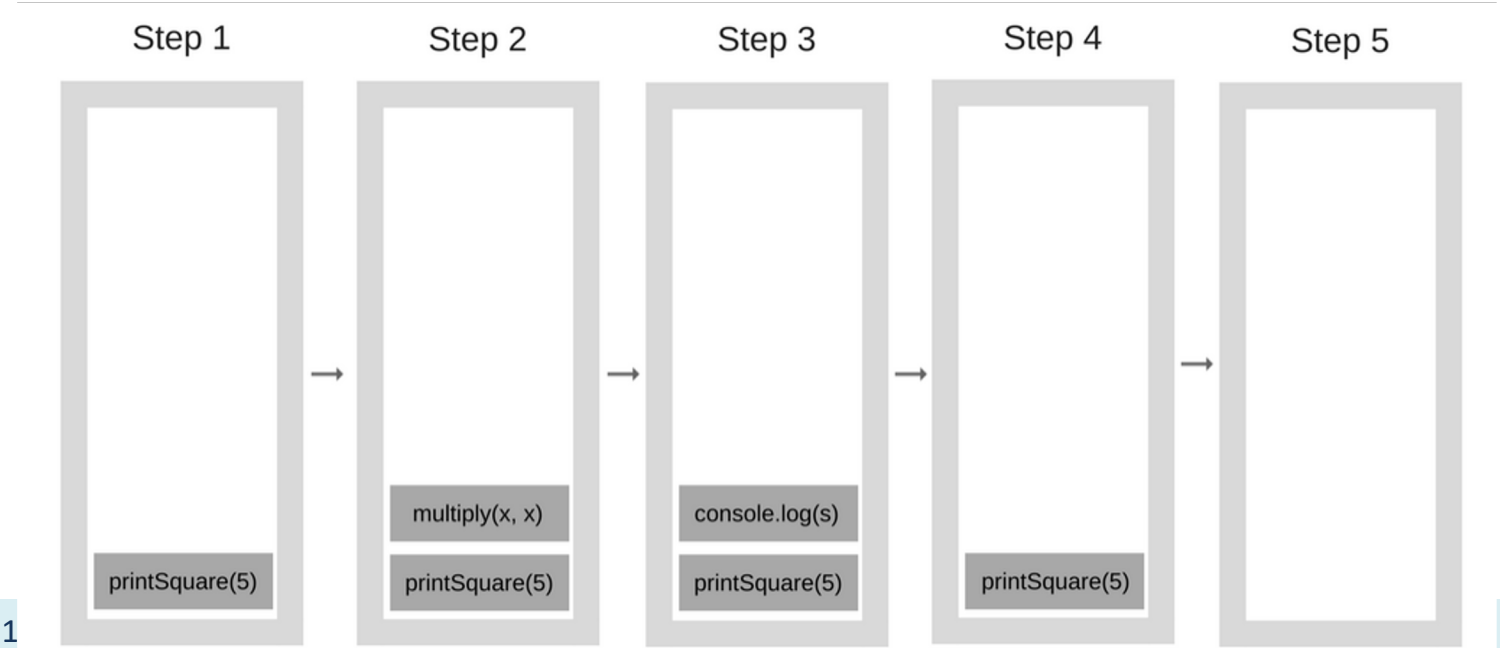
Engine plus:

- Web APIs — provided by browsers, like the DOM, AJAX, setTimeout and more.
- Event loop
- Callback queue

The Call Stack

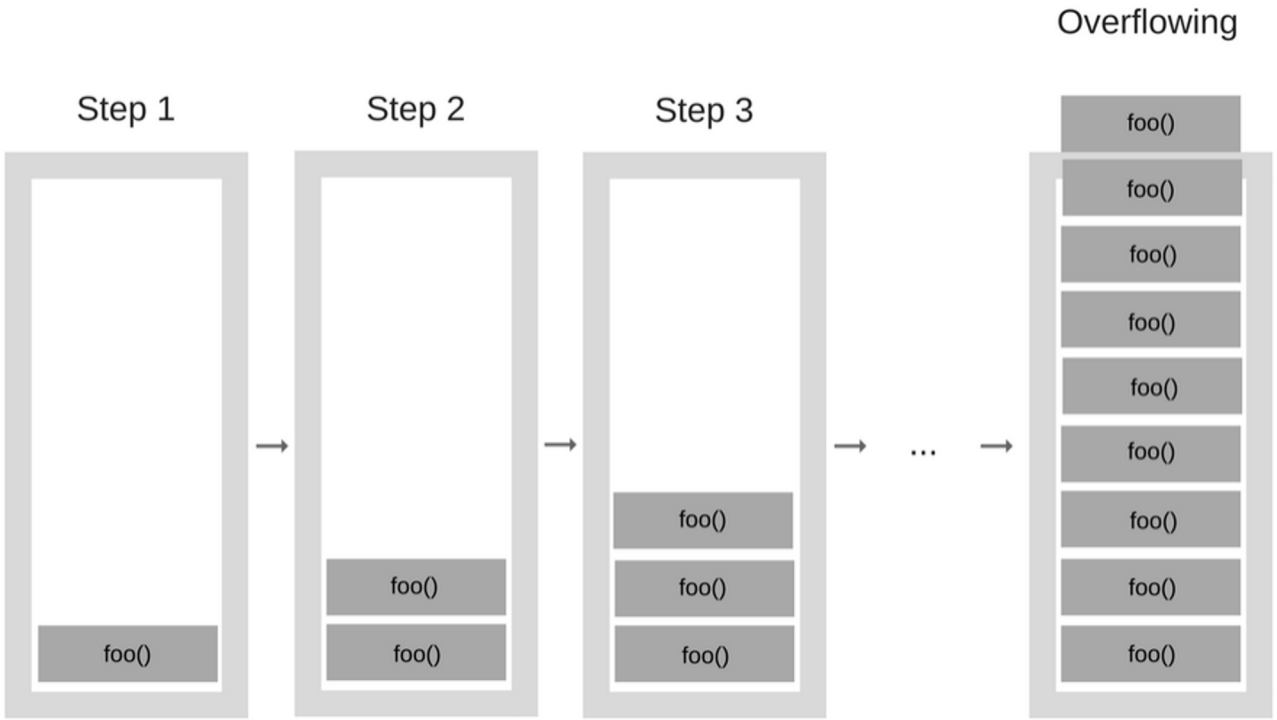
Is a data structure that records where in the program we are. Each entry is called a **Stack Frame**.

```
function multiply(x, y) {
    return x * y;
}
function printSquare(x) {
    var s = multiply(x, x);
    console.log(s);
}
printSquare(5);
```



Aside: The Call Stack can overflow

```
function foo() {
    foo();
}
foo();
```



What happens when things are slow?

JavaScript is **single threaded** (single Call Stack).

Problem: while the Call Stack has functions to execute, the browser can't actually do anything else — it's getting blocked.

What happens when things are slow?

JavaScript is **single threaded** (single Call Stack).

Problem: while the Call Stack has functions to execute, the browser can't actually do anything else — it's getting blocked.

```
Start script...
Download a file.
Done!
```

```
function task(message) {
    // emulate time consuming task
    let n = 10000000000;
    while (n > 0){
        n--;
    }
    console.log(message);
}

console.log('Start script...');
task('Download a file.');
console.log('Done!');
```

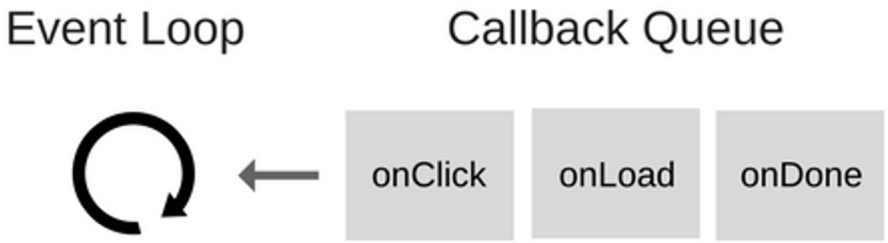
Solution: Callbacks

By far the most common way to express and manage asynchronicity in JavaScript programs.

```
function task(message) {
    // emulate time consuming task
    let n = 10000000000;
    while (n > 0){
        n--;
    }
    console.log(message);
}

console.log('Start script...');
setTimeout(() => {
    task('Download a file.');
}, 1000);
console.log('Done!');
```

```
Start script...
Done!
Download a file.
```

The Event Loop

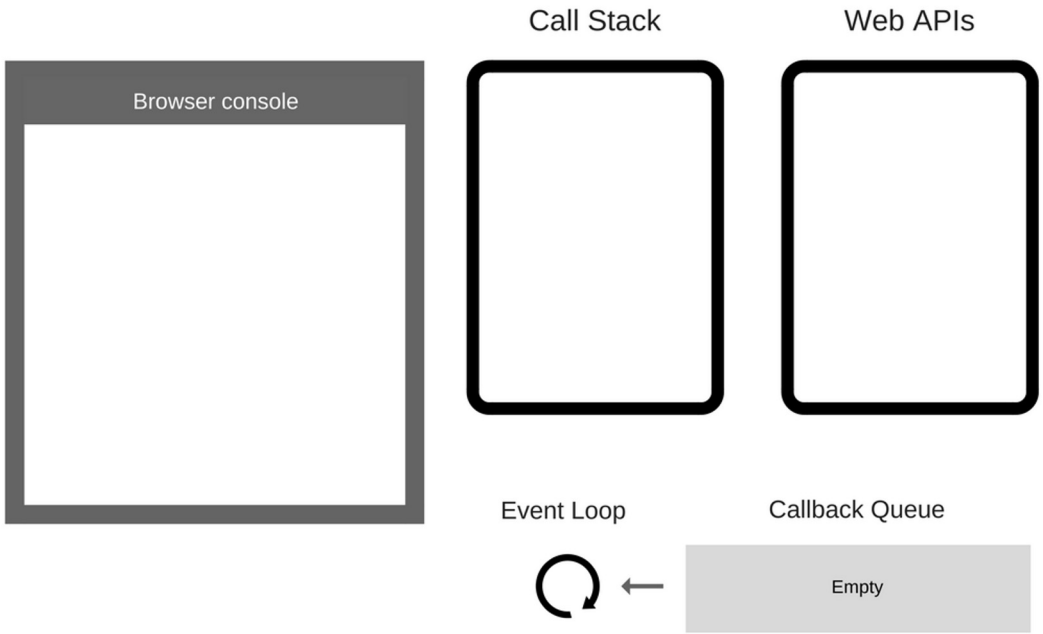


The Event Loop monitors the Call Stack and the Callback Queue.

If the Call Stack is empty, the Event Loop will take the first event from the queue and will push it to the Call Stack, which effectively runs it.

The Event Loop

1 / 16



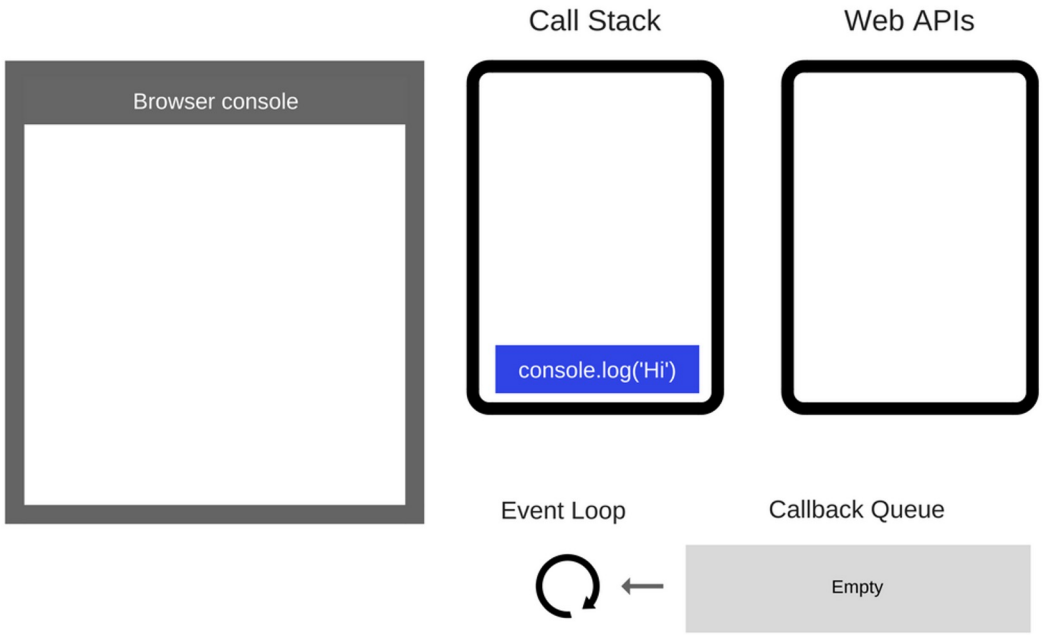

```
console.log('Hi');
setTimeout(function cb1() {
    console.log('cb1');
}, 5000);
console.log('Bye');
```

The state is clear.

The Event Loop

2 / 16

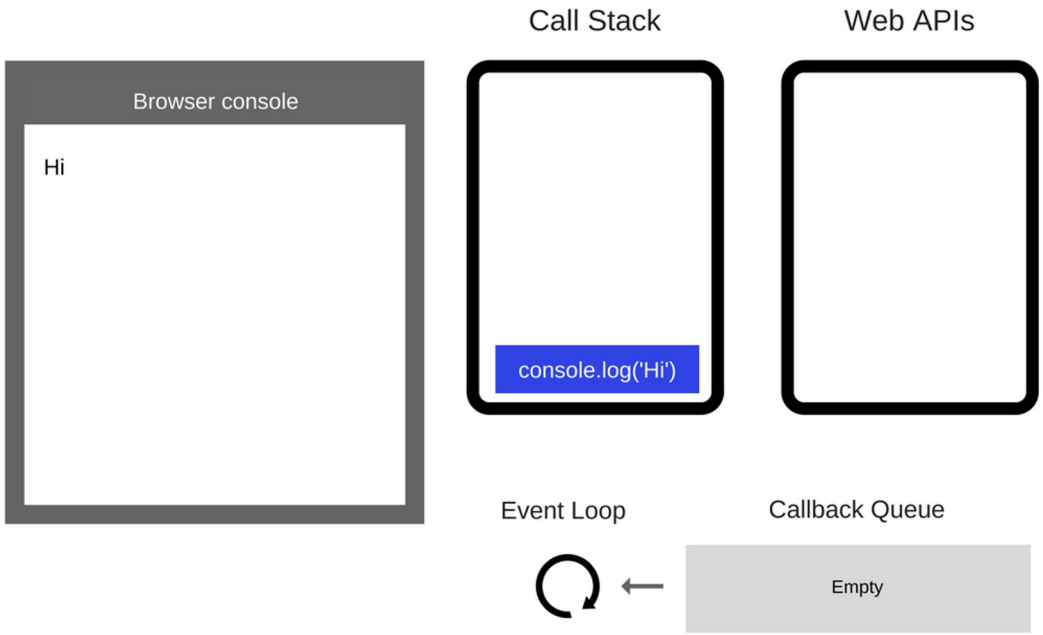
```
console.log('Hi');
setTimeout(function cb1() {
    console.log('cb1');
}, 5000);
console.log('Bye');
```



`console.log('Hi');` is added to the Call Stack.

The Event Loop

3 / 16



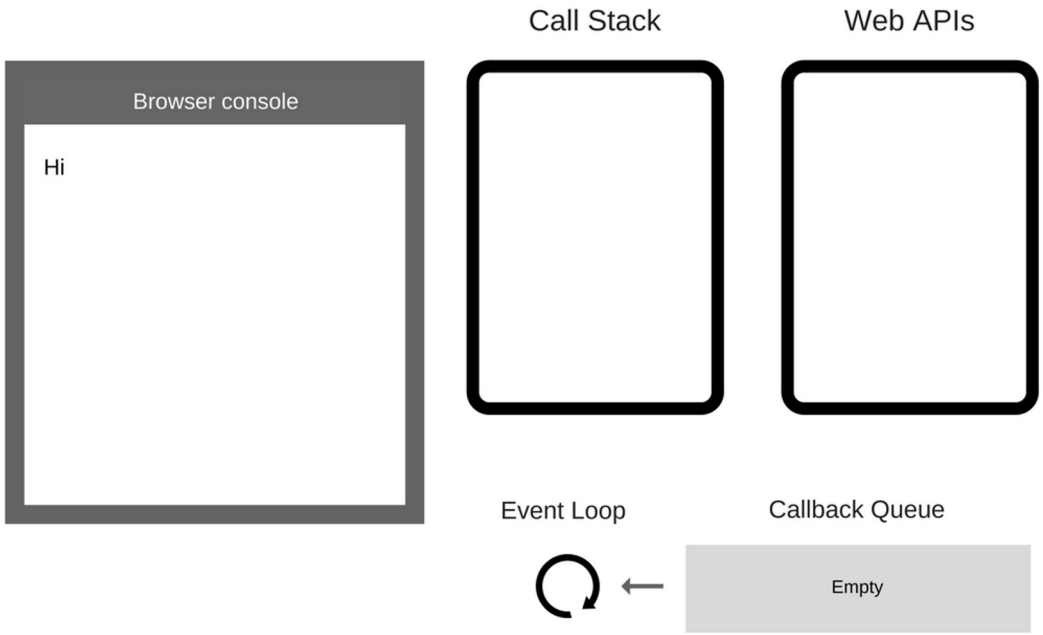

```
console.log('Hi');
setTimeout(function cb1() {
    console.log('cb1');
}, 5000);
console.log('Bye');
```

`console.log('Hi');` is executed.

The Event Loop

4 / 16

```
console.log('Hi');
setTimeout(function cb1() {
    console.log('cb1');
}, 5000);
console.log('Bye');
```

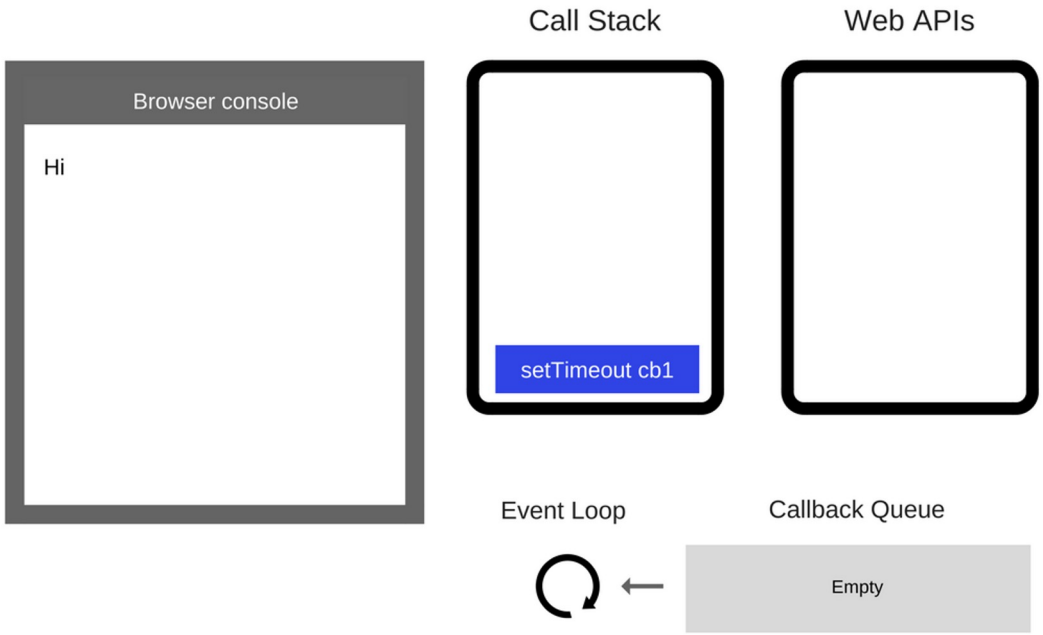


`console.log('Hi');` is removed from the Call Stack.

The Event Loop

5 / 16

```
console.log('Hi');
setTimeout(function cb1() {
    console.log('cb1');
}, 5000);
console.log('Bye');
```

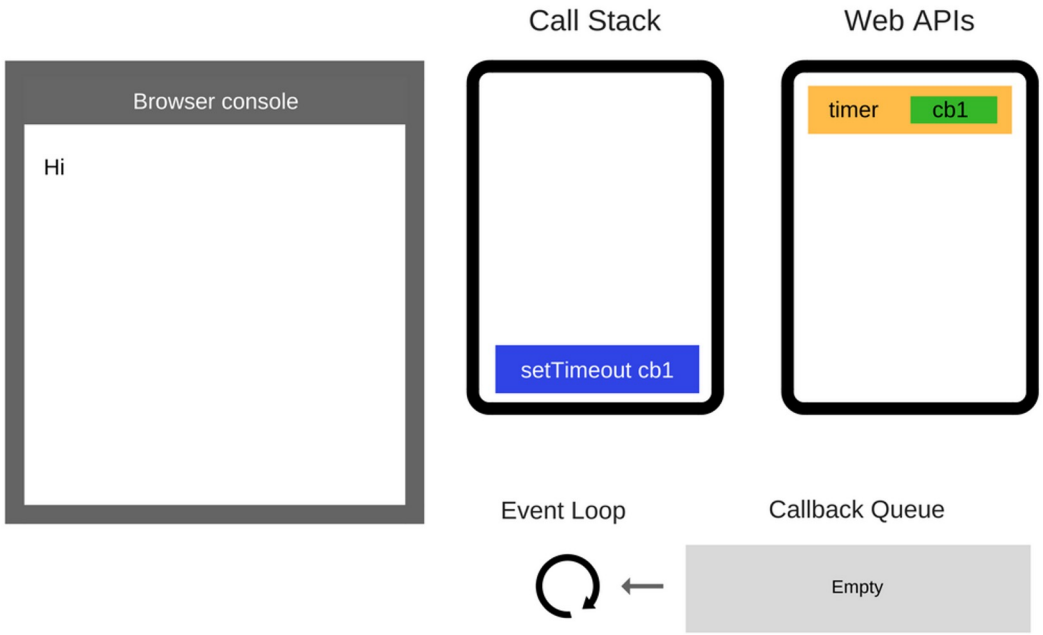


`setTimeout(function cb1() {...});` is added to the Call Stack.

The Event Loop

6 / 16

```
console.log('Hi');
setTimeout(function cb1() {
    console.log('cb1');
}, 5000);
console.log('Bye');
```



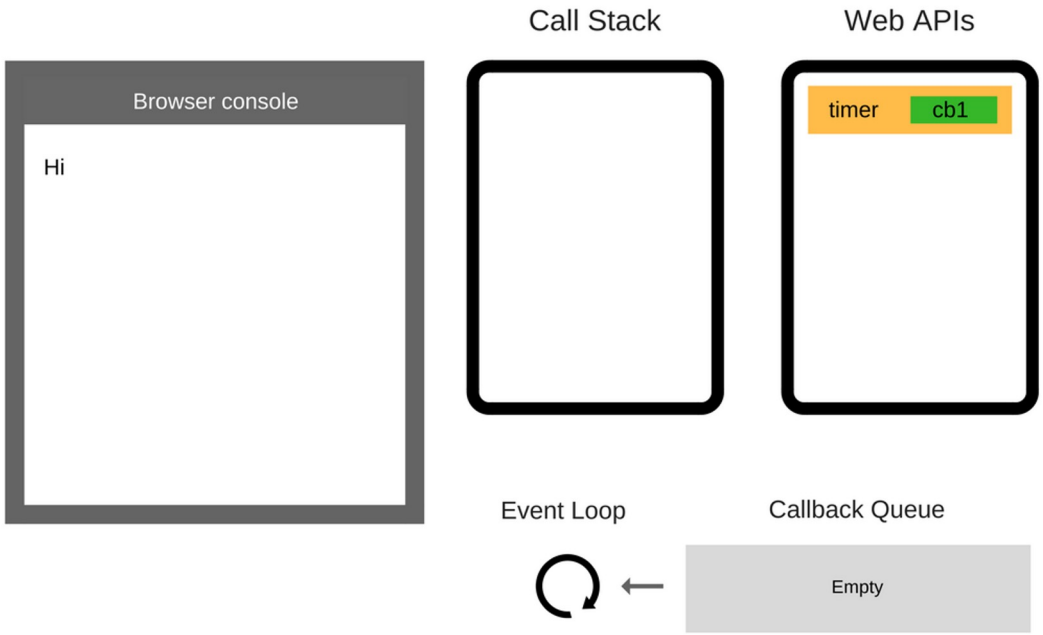
`setTimeout(function cb1() {...});` is executed.

The browser creates a timer as part of the Web APIs. It will handle the countdown for you.

The Event Loop

7 / 16

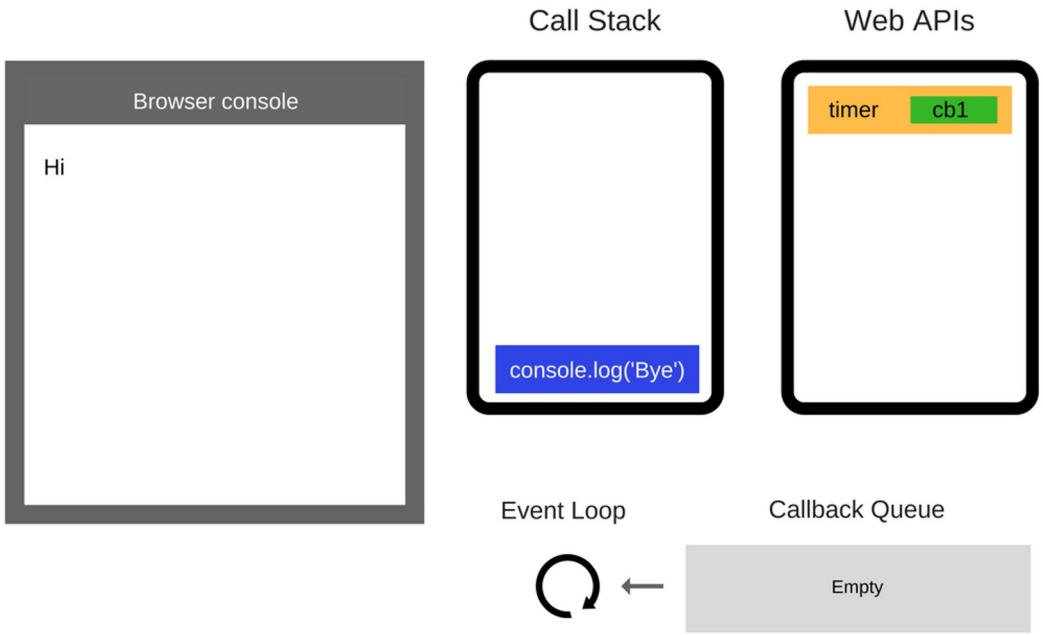
```
console.log('Hi');
setTimeout(function cb1() {
    console.log('cb1');
}, 5000);
console.log('Bye');
```



`setTimeout(function cb1() {...});` itself is complete and is removed from the Call Stack

The Event Loop


```
console.log('Hi');
setTimeout(function cb1() {
    console.log('cb1');
}, 5000);
console.log('Bye');
```

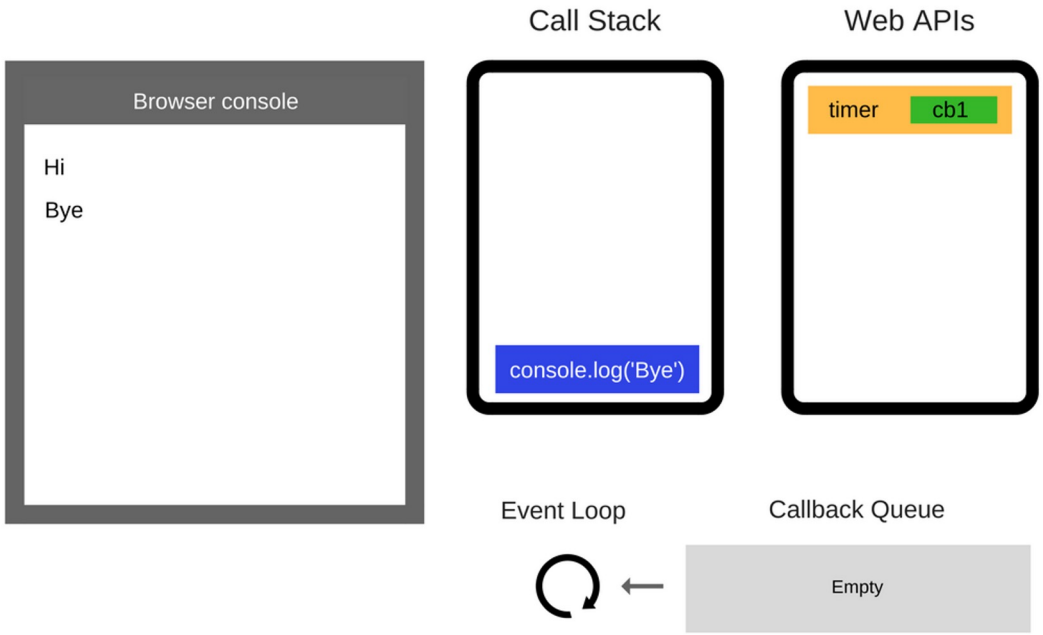


`console.log('Bye');` is added to the Call Stack.

The Event Loop

9 / 16

```
console.log('Hi');
setTimeout(function cb1() {
    console.log('cb1');
}, 5000);
console.log('Bye');
```

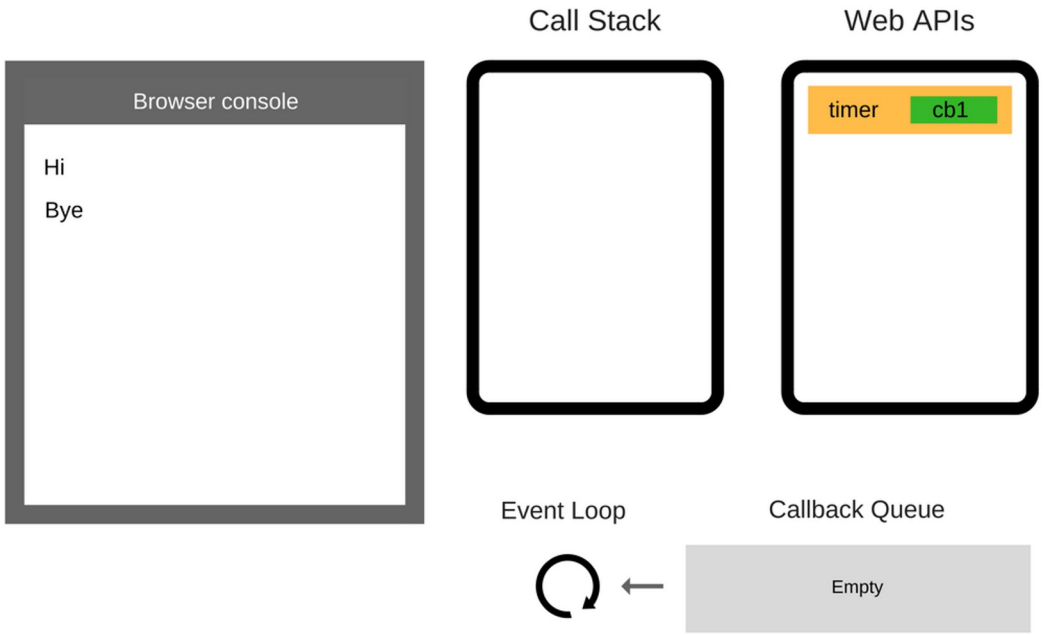


`console.log('Bye');` is executed.

The Event Loop

10 / 16

```
console.log('Hi');
setTimeout(function cb1() {
    console.log('cb1');
}, 5000);
console.log('Bye');
```

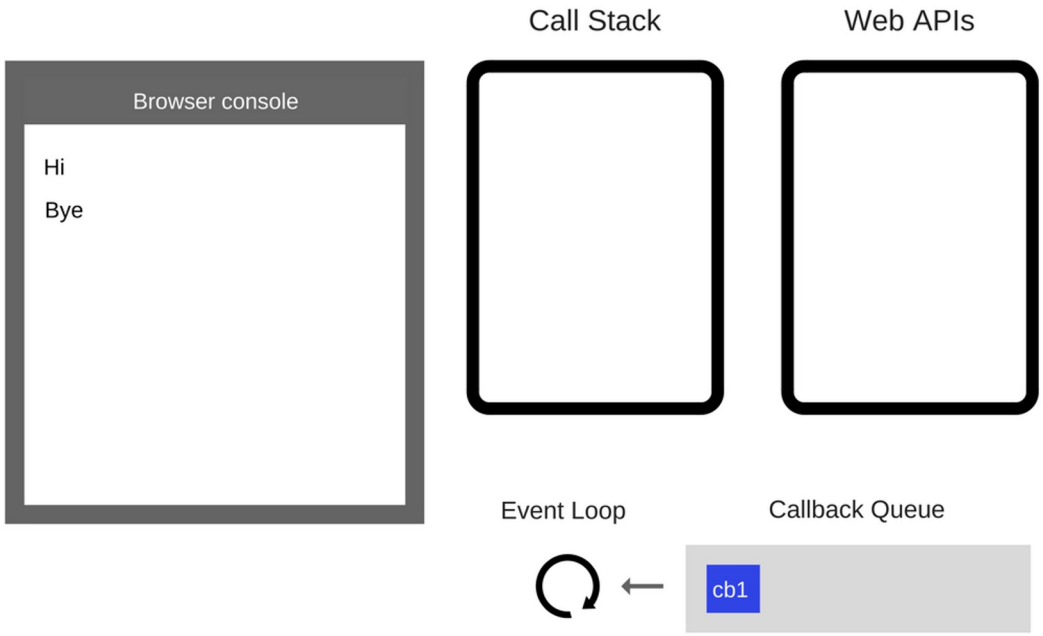


`console.log('Bye');` is removed from the Call Stack.

The Event Loop

11 / 16

```
console.log('Hi');
setTimeout(function cb1() {
    console.log('cb1');
}, 5000);
console.log('Bye');
```

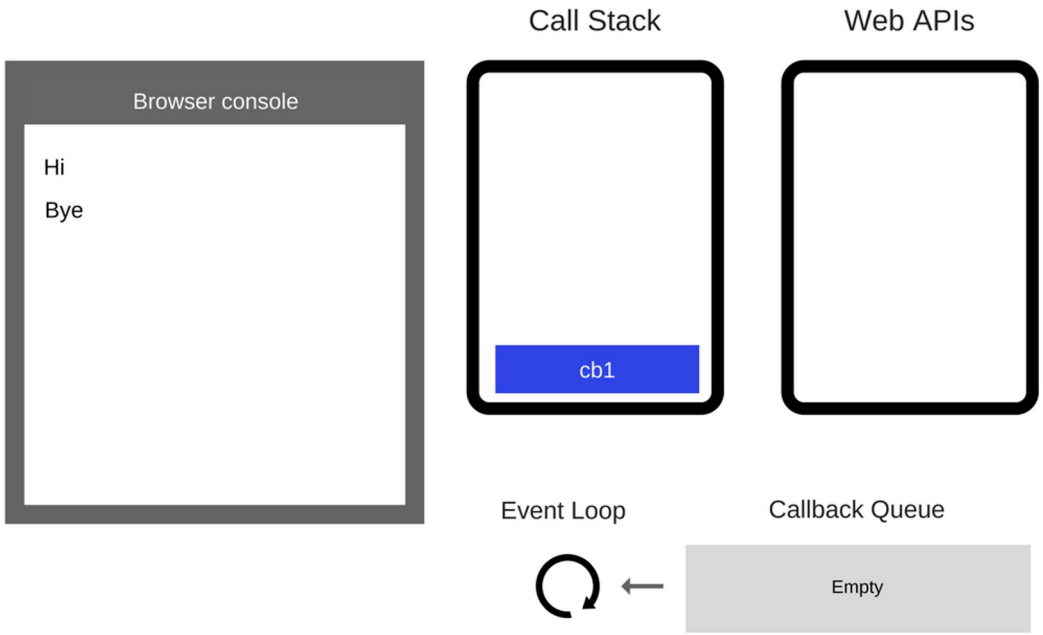


After at least 5000 ms, the timer completes and it pushes the cb1 callback to the Callback Queue.

The Event Loop

12 / 16

```
console.log('Hi');
setTimeout(function cb1() {
    console.log('cb1');
}, 5000);
console.log('Bye');
```

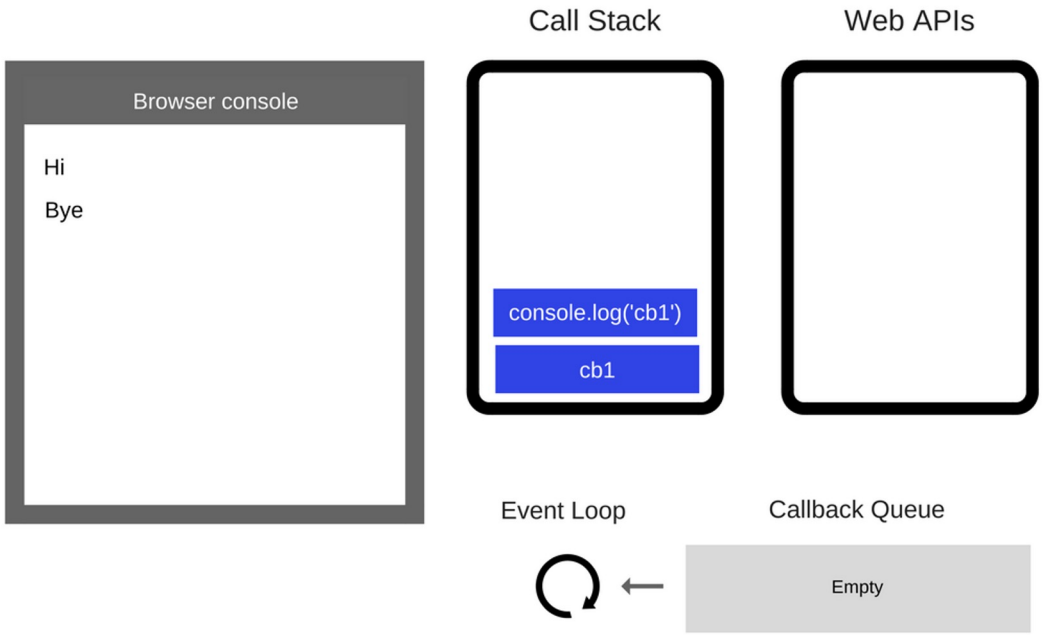


The Event Loop takes cb1 from the Callback Queue and pushes it to the Call Stack.

The Event Loop

13 / 16

```
console.log('Hi');
setTimeout(function cb1() {
    console.log('cb1');
}, 5000);
console.log('Bye');
```

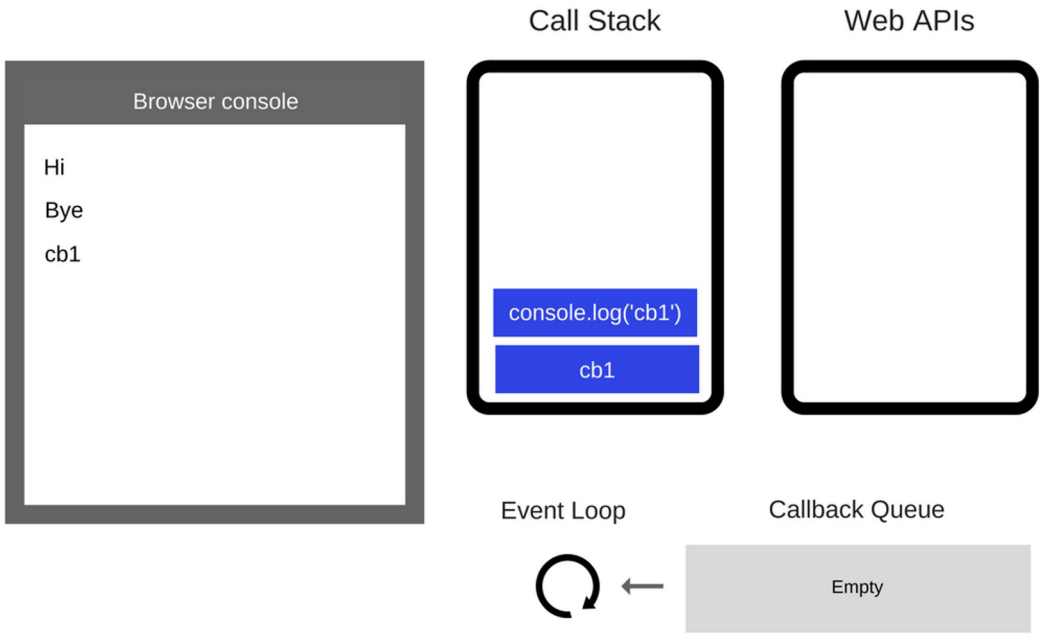


cb1 is executed and adds `console.log('cb1');` to the Call Stack.

The Event Loop

14 / 16

```
console.log('Hi');
setTimeout(function cb1() {
    console.log('cb1');
}, 5000);
console.log('Bye');
```

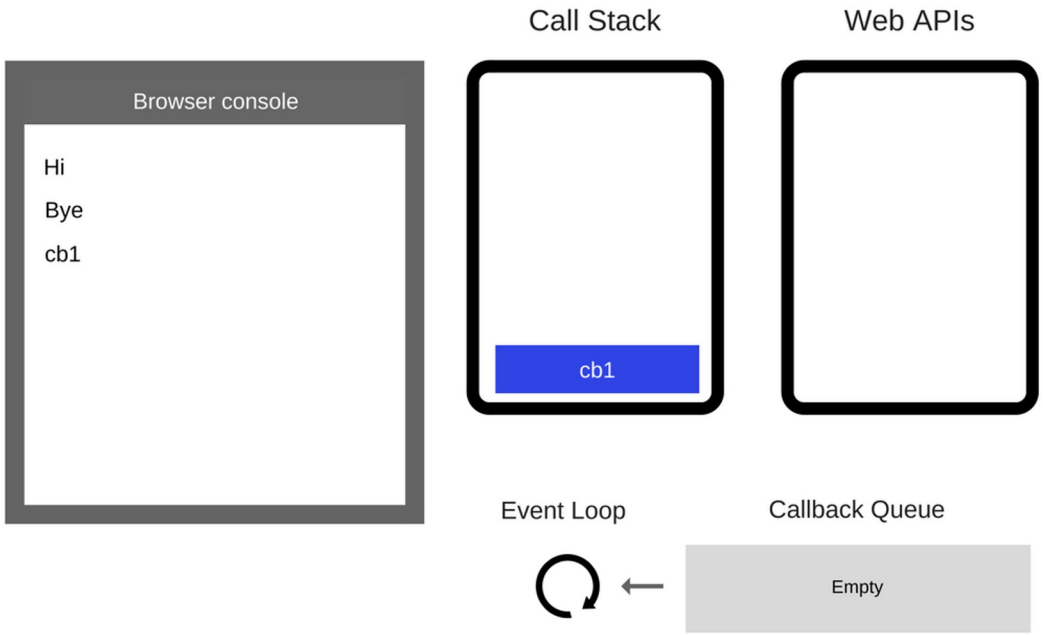


`console.log('cb1');` is executed

The Event Loop

15 / 16

```
console.log('Hi');
setTimeout(function cb1() {
    console.log('cb1');
}, 5000);
console.log('Bye');
```

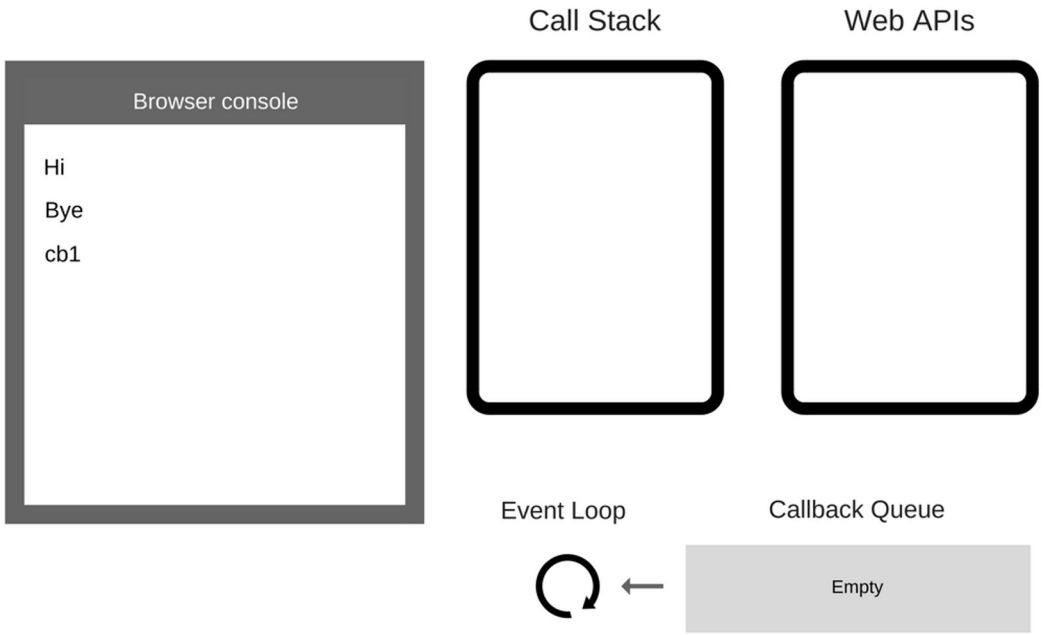


`console.log('cb1');` is removed from the Call Stack.

The Event Loop

16 / 16

```
console.log('Hi');
setTimeout(function cb1() {
    console.log('cb1');
}, 5000);
console.log('Bye');
```



cb1 is removed from the Call Stack.

“Callback Hell”?

- Issue caused by coding with complex nested callbacks.
- Every callback takes an argument that is a result of the previous callbacks.

Let’s imagine we’re trying to make a burger:

1. Get ingredients
2. Cook the beef
3. Get burger buns
4. Put the cooked beef between the buns
5. Serve the burger

“Callback Hell”?

- Issue caused by coding with complex nested callbacks.
- Every callback takes an argument that is a result of the previous callbacks.

If synchronous:

```
const makeBurger = () => {
  const beef = getBeef();
  const patty = cookBeef(beef);
  const buns = getBuns();
  const burger = putBeefBetweenBuns(buns, beef);
  return burger;
};

const burger = makeBurger();
serve(burger);
```

“Callback Hell”?

- Issue caused by coding with complex nested callbacks.
- Every callback takes an argument that is a result of the previous callbacks.

If asynchronous:

```
const makeBurger = nextStep => {
  getBeef(function (beef) {
    cookBeef(beef, function (cookedBeef) {
      getBuns(function (buns) {
        putBeefBetweenBuns(buns, beef, function(burger) {
          nextStep(burger)
        })
      })
    })
  })
}

// Make and serve the burger
makeBurger(function (burger) => {
  serve(burger)
})
```

Modern Alternatives (to be revisited)

- Promises
  - a way to write async code that still appears as though it is executing in a top-down way.
  - handles more types of errors due to encouraged use of try/catch style error handling.
- Generators
  - let you 'pause' individual functions without pausing the state of the whole program.
- Async functions
  - since ES7
  - further wrap generators and promises in a higher-level syntax

Useful References

- https://blog.sessionstack.com/how-does-javascript-actually-work-part-1-b0bacc073cf
- https://blog.sessionstack.com/how-javascript-works-event-loop-and-the-rise-of-async-programming-5-ways-to-better-coding-with-2f077c4438b5
- https://www.javascripttutorial.net/javascript-event-loop/
- https://www.freecodecamp.org/news/how-to-deal-with-nested-callbacks-and-avoid-callback-hell-1bc8dc4a2012/

Forming Design Patterns

- We've seen:
  - Function-based dispatch (callbacks)
  - Using queues to manage asynchronous events
- Some of the most common building blocks of concurrent, distributed systems

Summary

- We’re not in Kansas anymore
  - Real-world programs aren’t only back-end, like in HW3, nor only front-end, like TicTacToe with browserify, nor some entangled mix, like FlashCards.
- To balance a front-end and back-end, we need:
  - Good design, based on decoupling the UI and back-end
    - We talked about MVC, Client-server
  - Structures to implement and handle concurrency
    - We talked about callbacks
  - Way more concurrency in upcoming lectures