



## 6.React for Two Computers

Talk about Server / Client relation i.e. request / response model.

We start from the following app.

```
const catNames = [
  'Alonzo',
  'Bill Bailey',
  'Bombalurina',
  'Electra',
  'Plato'
];

function onClick() {
  const index = Math.floor(Math.random() * catNames.length)
  const catName = catNames[index];
  document.body.innerText = catName;
}
```

A screenshot of a code editor window titled "client.js". The code defines a constant `catNames` containing five strings: 'Alonzo', 'Bill Bailey', 'Bombalurina', 'Electra', and 'Plato'. It then defines a function `onClick()` that uses `Math.floor(Math.random() * catNames.length)` to select a random index from the array, retrieves the corresponding cat name, and sets it as the inner text of the document body. A "Reveal" button is visible on the right side of the editor.

The data should not be hardcoded. We need to be able to fetch it from database / file system / etc. Traditionally we could do it in the following manner:

```
async function onClick() {
  const response = await fetch('/api/cat-names');
  const json = await response.json();
  const { catNames } = json;
  const index = Math.floor(Math.random() * catNames.length);
  const catName = catNames[index];
  document.body.innerText = catName;
}
```

A screenshot of a browser window. On the left, the code editor shows the same `client.js` file, but the function `onClick()` now uses `fetch('/api/cat-names')` to get data from an API instead of hardcoding the names. On the right, the browser's developer tools are open, showing the "Console" tab with the word "Electra" printed in it, indicating the fetched name has been displayed.

Now the behavior of the app is changed. On slow connections there is a bit of delay and it is not instantaneous anymore. That is because it takes some time to load the data. For some tasks we expect to have some kind of delay in the UI if for example we submit a new blog post or save settings, etc. We want to be able to say that given interaction has to be synchronous i.e. we want the data to be already there and to not wait for it to load and visualize later. From client-side perspective there is not much we can do.

One thing that we can do is to look at the server code.

Name	Status	Domain	Size	T...
localhost	200 OK	localhost	428 B 285 B	3...
cat-names /api	(fail... net...)	localhost	0 B 0 B	4...

```

import { createServer } from 'http';
import { readFile } from 'fs/promises';

async function server(url) {
  if (url === '/api/cat-names') {
    const catFile = await readFile('./cats.txt', 'utf8');
    const catNames = catFile.split('\n');
    const json = { catNames };
    return json;
  }

  if (url === '/') {
    return `<!doctype html>
      <html>
        <head>
          <link rel="stylesheet" href="/style.css">
        </head>
        <body>
          <script src="client.js"></script>
        </body>
      </html>`;
  }
}

server().listen(3001);
  
```

Name	Status	Domain	Size	T...
localhost	200 OK	localhost	428 B 285 B	3...
cat-names /api	(fail... net...)	localhost	0 B 0 B	4...

```

}
}

if (url === '/') {
  return `<!doctype html>
    <html>
      <head>
        <link rel="stylesheet" href="/style.css">
      </head>
      <body>
        <script src="client.js"></script>
        <button onClick="onClick()">
          Reveal
        </button>
      </body>
    </html>`;
}

if (url === '/client.js') {
  return readFile('./client.js', 'utf8');
}  
```

The interesting thing we notice is the client.js script.

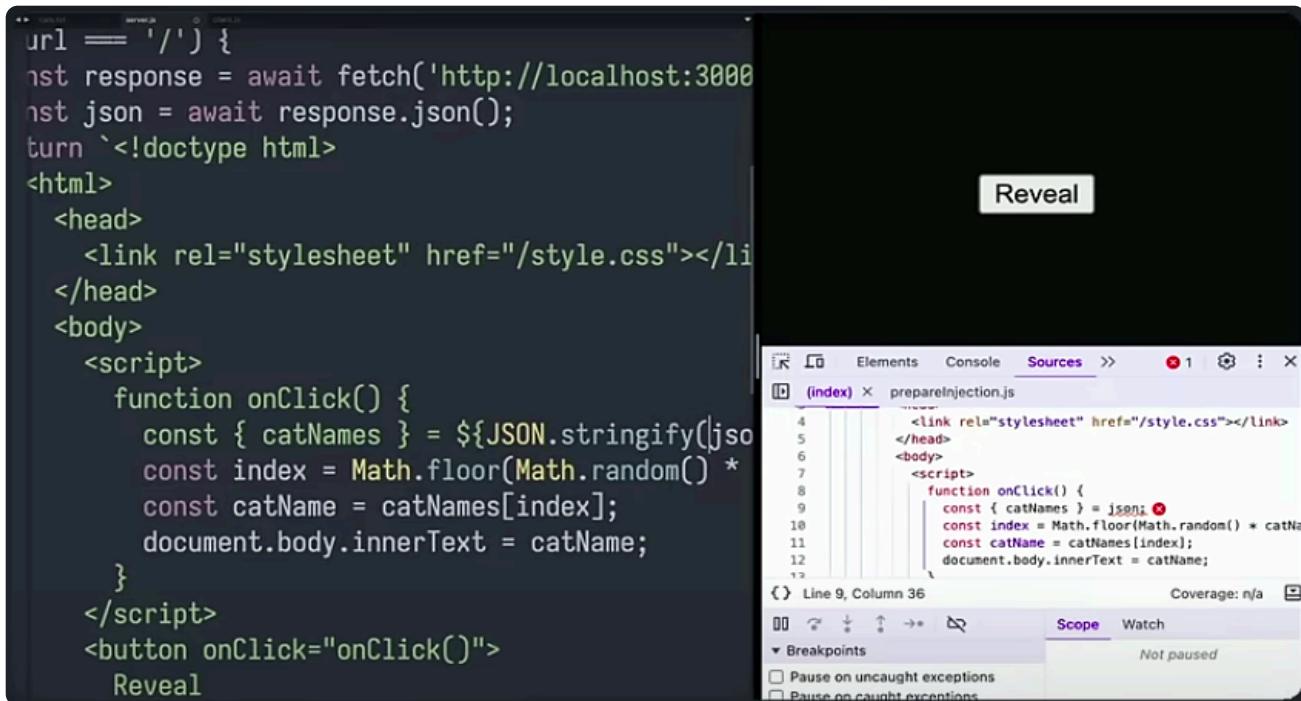
```
if (url === '/') {
  return `<!doctype html>
<html>
  <head>
    <link rel="stylesheet" href="/style.css"></head>
  </head>
  <body>
    <script>
      async function onClick() {
        const response = await fetch('/api/cat-names');
        const json = await response.json();
        const { catNames } = json;
        const index = Math.floor(Math.random() * catNames.length);
        const catName = catNames[index];
        document.body.innerText = catName;
      }
    </script>
    <button onClick="onClick()">
      Reveal
    </button>
  </body>
</html>`;
}

Reveal
```

We may think that the server and client code are two separate programs that talk to each other. We could look at different perspective - a single program that spans to different devices / computers. The program is evaluated into steps. Thinking in that direction we could do some optimizations to our code. Things that could simplify it, we could not notice otherwise.

For example we want to make the handler synchronous. Then we could "lift the data up" (related with fetching) in the same manner as React's "lift the state up" approach to parent.

Personal Note: So here we talk about RSC and we move the fetching data in the parent Server Component.



```

url == '/' ) {
  const response = await fetch('http://localhost:3000/api/cats');
  const json = await response.json();
  turn `<!doctype html>
<html>
  <head>
    <link rel="stylesheet" href="/style.css"></link>
  </head>
  <body>
    <script>
      function onClick() {
        const { catNames } = ${JSON.stringify(json)};
        const index = Math.floor(Math.random() * catNames.length);
        const catName = catNames[index];
        document.body.innerText = catName;
      }
    </script>
    <button onClick="onClick()">
      Reveal
    </button>
  </body>
</html>`;

  document.body.innerHTML = turn;
}

export default server;

```

After the change when we press the button we notice that there is no network request. That is because the data is already there.



Name	Status	Domain	Size	Time	Waterfall
localhost	200 OK	localhost	821 B	1...	678 B / 1.1 kB resources

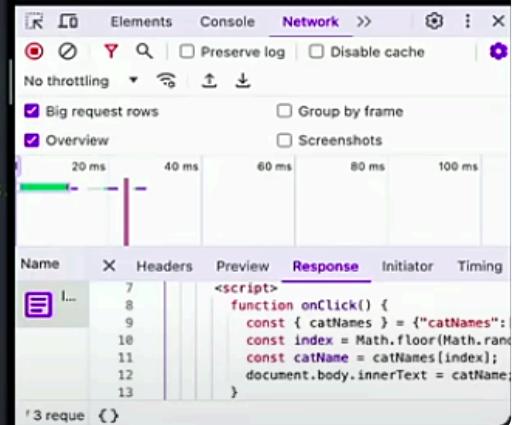
Other thing that we can do remove the api handler.

```

async function server(url) {
  if (url == '/') {
    const catFile = await readfile('./cats.txt', 'utf8')
    const catNames = catFile.split('\n')
    const json = { catNames };
    return `<!doctype html>
<html>
  <head>
    <link rel="stylesheet" href="/style.css"></link>
  </head>
  <body>
    <script>
      function onClick() {
        const { catNames } = ${JSON.stringify(json)};
        const index = Math.floor(Math.random() * catNames.length);
        const catName = catNames[index];
        document.body.innerText = catName;
      }
    </script>
    <button onClick="onClick()">
      Reveal
    </button>
  </body>
</html>`;
  }
}

```

*Quaxo*



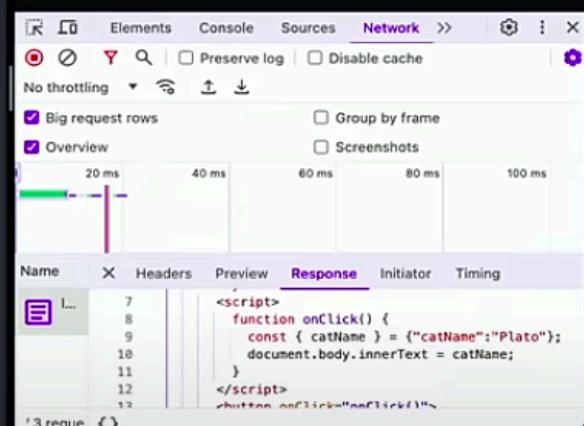
If we take a look at the response again we will notice that there is the whole data i.e. the array but we really need just one item from the array.

```

async function server(url) {
  if (url == '/') {
    const catFile = await readfile('./cats.txt', 'utf8')
    const catNames = catFile.split('\n')
    const index = Math.floor(Math.random() * catNames.length)
    const catName = catNames[index];
    const json = { catName };
    return `<!doctype html>
<html>
  <head>
    <link rel="stylesheet" href="/style.css"></link>
  </head>
  <body>
    <script>
      function onClick() {
        const { catName } = ${JSON.stringify(json)};
        document.body.innerText = catName;
      }
    </script>
    <button onClick="onClick()">
      Reveal
    </button>
  </body>
</html>`;
  }
}

```

*Plato*



After the changes now we have code that executes first on one computer and emits code that will be sent on another computer and we can pass some data together with this code. This is the fundamental shape of the problem and not matter which library or approach we are using. It kind of looks like this. There is a very clear separation of how the data flows through the application. But we don't want to write code like this i.e. in the template strings and without syntax highlighting / type checking / etc. Generally we

want to split the things and give them appropriate names. We have the concept of components and may use it.

The screenshot shows a code editor with two files open. On the left is `server.js`, which contains the original code from the previous slide. On the right is `CatNameGenerator.js`, which has been refactored into components. The `server.js` file remains largely the same, while `CatNameGenerator.js` defines a component named `RevealButton`.

```
server.js
if (url === '/') {
  const catFile = await readFile('../cats.txt',
  const catNames = catFile.split('\n')
  const index = Math.floor(Math.random() * catNames.length);
  const catName = catNames[index];
  const json = { catName };
  return `<!doctype html>
<html>
<head>
<link rel="stylesheet" href="/style.css">
</head>
<body>
<script>
const { catName } = ${JSON.stringify(json)}

function onClick() {
  document.body.innerText = catName;
}
</script>
<button onClick="onClick()">
  Reveal
</button>
</body>
`}
```

```
CatNameGenerator.js
import { readFile } from 'fs/promises';

export default async function CatNameGenerator() {
  const catFile = await readFile('../cats.txt', 'utf8')
  const catNames = catFile.split('\n')
  const index = Math.floor(Math.random() * catNames.length);
  const catName = catNames[index];
  return <RevealButton catName={catName} />;
}

function RevealButton({ catName }) {
  function onClick() {
    document.body.innerText = catName;
  }
  return (
    <button onClick={onClick}>
      Reveal
    </button>
  );
}
```

But actually one code could be executed on different computer and much earlier (i.e. `CatNameGenerator` is considered as a Server Component) and the other (i.e. `ButtonReveal` is considered as a Client Component) is executed on the client.

The screenshot shows the same code editor setup as before, but the code has been further refined. The `server.js` file now uses imports and exports, and the `CatNameGenerator.js` file contains only the component definition.

```
server.js
if (url === '/') {
  const catFile = await readFile('../cats.txt',
  const catNames = catFile.split('\n')
  const index = Math.floor(Math.random() * catNames.length);
  const catName = catNames[index];
  const json = { catName };
  return `<!doctype html>
<html>
<head>
<link rel="stylesheet" href="/style.css">
</head>
<body>
<script>
const { catName } = ${JSON.stringify(json)}

function onClick() {
  document.body.innerText = catName;
}
</script>
<button onClick="onClick()">
  Reveal
</button>
</body>
`}
```

```
CatNameGenerator.js
import { readFile } from 'fs/promises';

export default async function CatNameGenerator() {
  const catFile = await readFile('../cats.txt', 'utf8')
  const catNames = catFile.split('\n')
  const index = Math.floor(Math.random() * catNames.length);
  const catName = catNames[index];
  return <RevealButton catName={catName} />;
}

function RevealButton({ catName }) {
  function onClick() {
    document.body.innerText = catName;
  }
  return (
    <button onClick={onClick}>
      Reveal
    </button>
  );
}
```

We could pass data from the server component to the client component.

```

// CatNameGenerator.js
= '/' {
atFile = await readfile('./cats.txt', 'utf8')
atNames = catFile.split('\n')
index = Math.floor(Math.random() * catNames.length)
catName = catNames[index];
son = { catName };
`<!doctype html>
>
<head>
<link rel="stylesheet" href="/style.css"></link>
</head>
<body>
<script>
const { catName } = ${JSON.stringify(json)};
function onClick() {
  document.body.innerText = catName;
}
</script>
<button onClick="onClick()">
  Reveal
</button>
</body>
</html>

```

```

// RevealButton.js
import { readfile } from 'fs/promises';

export default async function CatNameGenerator() {
  const catFile = await readfile('./cats.txt', 'utf8')
  const catNames = catFile.split('\n')
  const index = Math.floor(Math.random() * catNames.length)
  const catName = catNames[index];
  return <RevealButton catName={catName} />;
}

}

'use client';

function RevealButton({ catName }) {
  function onClick() {
    document.body.innerText = catName;
  }
  return (
    <button onClick={onClick}>
      Reveal
    </button>
  );
}

```

```

import { readfile } from 'fs/promises';
import RevealButton from './RevealButton';

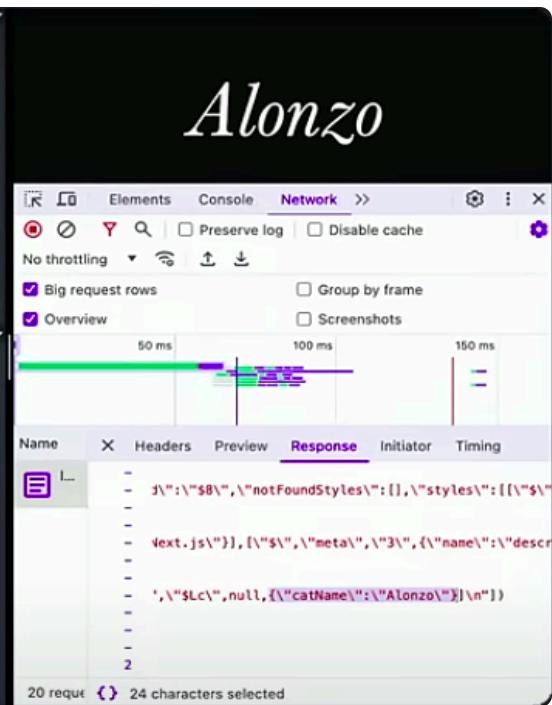
export default async function CatNameGenerator() {
  const catFile = await readfile('./cats.txt', 'utf8')
  const catNames = catFile.split('\n')
  const index = Math.floor(Math.random() * catNames.length)
  const catName = catNames[index];
  return <RevealButton catName={catName} />;
}

}

'use client';

export default function RevealButton({ catName }) {
  function onClick() {
    document.body.innerText = catName;
  }
  return (
    <button onClick={onClick}>
      Reveal
    </button>
  );
}

```



Recap: we used to think about the client and the server as a two separate programs. But we may also think about rendering a UI as a single program that is split between two computers. If we look at the vanilla JS representation the way it works is that this part executes in the server

```
server.js
```

```
async function server(url) {
  if (url == '/') {
    const catFile = await readfile('./cats.txt', 'utf8')
    const catNames = catFile.split('\n')
    const index = Math.floor(Math.random() * catNames.length)
    const catName = catNames[index]
    const json = { catName }
    return `<!doctype html>
      <html>
        <head>
          <link rel="stylesheet" href="/style.css"></link>
        </head>
        <body>
          <script>
            const { catName } = ${JSON.stringify(json)}

            function onClick() {
              document.body.innerText = catName;
            }
          </script>
          <button onClick="onClick()">
            Reveal
          </button>
        </body>
      </html>`
  }
}

CatNameGenerator.js
```

```
import { readfile } from 'fs/promises';

export default async function CatNameGenerator() {
  const catFile = await readfile('./cats.txt', 'utf8')
  const catNames = catFile.split('\n')
  const index = Math.floor(Math.random() * catNames.length)
  const catName = catNames[index]
  return <RevealButton catName={catName} />;
}
```

```
RevealButton.js
```

```
'use client';

function RevealButton({ catName }) {
  function onClick() {
    document.body.innerText = catName;
  }

  return (
    <button onClick={onClick}>
      Reveal
    </button>
  );
}
```

And then it emits a program that execute in the client

```
server.js
```

```
async function server(url) {
  if (url == '/') {
    const catFile = await readfile('./cats.txt', 'utf8')
    const catNames = catFile.split('\n')
    const index = Math.floor(Math.random() * catNames.length)
    const catName = catNames[index]
    const json = { catName }
    return `<!doctype html>
      <html>
        <head>
          <link rel="stylesheet" href="/style.css"></link>
        </head>
        <body>
          <script>
            const { catName } = ${JSON.stringify(json)}

            function onClick() {
              document.body.innerText = catName;
            }
          </script>
          <button onClick="onClick()">
            Reveal
          </button>
        </body>
      </html>`
  }
}

CatNameGenerator.js
```

```
import { readfile } from 'fs/promises';

export default async function CatNameGenerator() {
  const catFile = await readfile('./cats.txt', 'utf8')
  const catNames = catFile.split('\n')
  const index = Math.floor(Math.random() * catNames.length)
  const catName = catNames[index]
  return <RevealButton catName={catName} />;
}
```

```
RevealButton.js
```

```
'use client';

function RevealButton({ catName }) {
  function onClick() {
    document.body.innerText = catName;
  }

  return (
    <button onClick={onClick}>
      Reveal
    </button>
  );
}
```

And there is kind of like a hole through which we could pass an information i.e. serialized JSON.

```
server.js
```

```
async function server(url) {
  if (url == '/') {
    const catFile = await readFile('../cats.txt', 'utf8')
    const catNames = catFile.split('\n')
    const index = Math.floor(Math.random() * catNames.length)
    const catName = catNames[index];
    const json = { catName };
    return `<!doctype html>
      <html>
        <head>
          <link rel="stylesheet" href="/style.css"></link>
        </head>
        <body>
          <script>
            const { catName } = ${JSON.stringify(json)};
            function onClick() {
              document.body.innerText = catName;
            }
          </script>
          <button onClick="onClick()">
            Reveal
          </button>
        </body>
      </html>`;
  }
}

CatNameGenerator.js
```

```
import { readFile } from 'fs/promises';

export default async function CatNameGenerator() {
  const catFile = await readFile('../cats.txt', 'utf8')
  const catNames = catFile.split('\n')
  const index = Math.floor(Math.random() * catNames.length)
  const catName = catNames[index];
  return <RevealButton catName={catName} />;
}

RevealButton.js
```

```
'use client';

function RevealButton({ catName }) {
  function onClick() {
    document.body.innerText = catName;
  }

  return (
    <button onClick={onClick}>
      Reveal
    </button>
  );
}
```

In the React version we have Server and Client components. For practical reasons on the server we want to run both stages, because we want to generate the initial HTML so we run this stage with the initial state. But on the client we have the part that update and re-render in response to state changes.