

Full Stack Development Lab Instructions
ENSF381: Lab07

React

Created by: Mahdi Jaberzadeh Ansari (He/Him)
Schulich School of Engineering
University of Calgary
Calgary, Canada
mahdi.ansari1@ucalgary.ca

Week 9, March 04/06, 2024

This lab has been modified for this course based on the [Intro to React](#) tutorial.

Contents

1	Introduction	1
1.1	Objectives	1
1.2	Prerequisites	1
1.3	Forming groups	1
1.4	Before submission	2
1.5	Academic Misconduct/Plagiarism	3
1.6	Marking Scheme	3
1.7	Complains	3
2	Exercise A (Creating a repository)	4
2.1	Initialize the Repository	4
2.2	Using SourceTree with Bitbucket	4
2.3	Inviting Collaborators to Your Bitbucket Repository	5
2.4	Deliverable	6
3	Exercise B (Creating React app)	7
3.1	Setup for the Lab	7
3.1.1	Prerequisites	7
3.1.2	Initiating the React app	7
3.1.3	Code Highlighter for JS codes (Optional)	9
3.2	Overview	9
3.2.1	What Is React?	9
3.2.2	Inspecting the Starter Code	10
3.2.3	Passing Data Through Props	10
3.2.4	Making an Interactive Component	11
3.2.5	Developer Tools	13
3.3	Completing the Game	14
3.3.1	Lifting State Up	14
3.4	Why Immutability Is Important	17
3.4.1	Data Change with Mutation	17
3.4.2	Data Change without Mutation	18
3.4.3	Complex Features Become Simple	18
3.4.4	Detecting Changes	18
3.4.5	Determining When to Re-Render in React	18
3.5	Function Components	18
3.5.1	Taking Turns	19
3.5.2	Declaring a Winner	21
3.6	Adding Time Travel	22
3.6.1	Storing a History of Moves	22
3.6.2	Lifting State Up, Again	23
3.6.3	Showing the Past Moves	26
3.6.4	Picking a Key	27
3.6.5	Implementing Time Travel	28
3.7	Deliverable	30

1 Introduction

1.1 Objectives

In Lab 7, you will gain hands-on experience with React.js, a popular JavaScript library for building user interfaces. This lab aims to provide practical skills in the following areas:

- **Understanding React Components:** You will learn the fundamentals of React components, which are the building blocks of any React application. This includes understanding JSX, props, and states, as well as how to create functional and class components.
- **State Management and Lifecycle Methods:** The lab focuses on managing state within React components and utilizing lifecycle methods. You will learn how to handle data within a component, pass data between components, and use lifecycle methods to control component rendering and behavior.
- **Building a React Application:** You will enhance your skills by building a small React application. This includes setting up the development environment, creating a component hierarchy, and implementing routing and state management. The application will demonstrate the core features of React and how they work together in a real-world scenario.

1.2 Prerequisites

1. Basic understanding of computer operations.
2. A web browser (Chrome, Firefox, Safari, etc.) to view your HTML file.
3. BitBucket account.

1.3 Forming groups

- In this lab session, you **MUST** work with a partner (groups of three or more are not allowed).
- The main goal of working in a group is to learn:
 - how to do teamwork
 - how to not be a single player
 - how to not be bossing
 - how to play for team
 - how to tolerate others
 - how to behave with colleagues
 - how to form a winner team
 - and ...
- Working with a partner usually gives you the opportunity to discuss some of the details of the topic and learn from each other. Also, it will give you the opportunity to practice one of the popular methods of program development called pair programming. In this method, which is normally associated with the “Agile Software Development” technique, two programmers normally work together on the same workstation (you may consider a Zoom session for this purpose). While one partner, the driver, writes the code, the other partner, acting as an observer, looks over his or her shoulder, making sure the syntax and solution logic are correct. Partners should switch roles frequently in such a way that both have equivalent opportunities to practice both roles. **Please note that you MUST switch roles for each exercise.**
- When you have to work with a partner:

- Choose a partner that can either increase your knowledge or transfer your knowledge. (i.e., do not find a person with the same programming skill level!)
- Please submit only one lab report with both names. Submitting two lab reports with the same content will be considered copying and plagiarism.

1.4 Before submission

- For most of the labs, you will receive a DOCX file that you need to fill out the gaps. Make sure you have this file to fill out.
- All your work should be submitted as a single file in PDF format. For instructions about how to provide your lab reports, study the posted document on the D2L called [How to Hand in Your Lab assignment](#).
- Please note that if it is group work, only one team member must submit the solution in D2L. For ease of transferring your marks, please mention the group member's name and UCID in the description window of the submission form.
- If you have been asked to write code (HTML, CSS, JS, etc.), make sure the following information appears at the top of your code:
 - File Name
 - Assignment and exercise number
 - Your names in alphabetic order
 - Submission Date:

Here is an example for CSS and JS files:

```

1  <!--
2  =====
3  Name      : lab7_exe_D.html
4  Assignment : Lab 7, Exercise D
5  Author(s) : Mahdi Ansari, William Arthur Philip Louis
6  Submission : May 21, 2030
7  Description : React.
8  =====
9  -->

```

- Some exercises in this lab and future labs will not be marked. Please do not skip them, because these exercises are as important as the others in learning the course material.
- In courses like ENSF381, some students skip directly to the exercises that involve writing code, skipping sections such as “Read This First,” or postponing the diagram-drawing until later. That’s a bad idea for several reasons:
 - “Read This First” sections normally explain some technical or syntax details that may help you solve the problem or may provide you with some hints.
 - Some lab exercises may ask you to draw a diagram, and most of the students prefer to hand-draw them. In these cases, you need to scan your diagram with a scanner or an appropriate device, such as your mobile phone, and insert the scanned picture of your diagram into your PDF file (the lab report). A possible mobile app to scan your documents is Microsoft Lens, which you can install on your mobile device for free. Please make sure your diagram is clear and readable; otherwise, you may either lose marks or it could be impossible for TAs to mark it at all.
 - Also, it is better to use the [Draw.io](#) tool if you need to draw any diagram.

- Drawing diagrams is an important part of learning how to visualize data transfer between modules and so on. If you do diagram-drawing exercises at the last minute, you won't learn the material very well. If you do the diagrams first, you may find it easier to understand the code-writing exercises, so you may be able to finish them more quickly.

- **Due Dates:**

- You must submit your solution until 11:59 p.m. on the same day that you have the lab session.
- Submissions until 24 hours after the due date get a maximum of 60% of the mark, and after 24 hours, they will not be evaluated and get 0.

1.5 Academic Misconduct/Plagiarism

- Ask for help, but don't copy.
 - You can get help from lab instructor(s), TAs, or even your classmates as long as you do not copy other people's work.
 - If we realize that even a small portion of your work is a copy from a classmate, both parties (the donor and the receiver of the work) will be subject to academic misconduct (plagiarism). More importantly, if you give exercise solutions to a friend, you are not doing him or her a favor, as he or she will never learn that topic and will pay off for this mistake during the exam or quiz. So, please do not put yourself and your friend in a position of academic misconduct.
 - You can use ChatGPT, but please note that it may provide similar answers for others too, or even the wrong answers. For example, it has been shown that AI can hallucinate, proposing the use of libraries that do not actually exist ¹. So, we recommend that you imagine ChatGPT as an advanced search engine, not a solution provider.
 - In order to find out who is abusing these kinds of tools, we will eventually push you toward the incorrect responses that ChatGPT might produce. In that case, you might have failed for the final mark and be reported to administration.
 - If we ask you to investigate something, don't forget to mention the source of your information. Reporting without reference can lead to a zero mark even by providing a correct answer.

1.6 Marking Scheme

- You should not submit anything for the exercises that are not marked.
- In Table 1, you can find the marking scheme for each exercise.

Table 1: Marking scheme

Exercise	Marks
A	5 marks
B	45 marks
Total	50 marks

1.7 Complaints

- Your grades will be posted one week following the submission date, which means they will be accessible at the subsequent lab meeting.
- Normally, the grades for individual labs are assessed by a distinct TA for each lab and section. Kindly refrain from contacting all TAs. If you have any concerns regarding your grades, please direct an email to the TA responsible for that specific lab.

¹<https://perma.cc/UQS5-3BBP>

2 Exercise A (Creating a repository)

In this lab, we will be working with **Bitbucket** instead of GitHub. You will create a new repository on Bitbucket and clone it to your local machine. If you need a refresher on using SourceTree, refer to the instructions in Lab03.

2.1 Initialize the Repository

Here are the steps to get started with Bitbucket:

- First, create a Bitbucket account if you don't already have one. Visit [Bitbucket's website](#) and sign up.
- Once logged in, first you need to create a workspace (e.g., Name it `ENSF381`, but you need different ids like figure 2), then a project (i.e., `labs`) and finally create a new repository named `Lab7`. During the creation process:
 - Select the option to **Include a README** so that your repository is initialized with a README file.
 - Bitbucket does not have an option for adding predefined licenses, but you can add a `.gitignore` file, which helps to prevent some local files from being pushed into the remote repository.
 - Figure 3 shows the settings that I selected for my repository.
- After creating the repository, it will contain the `README.md` and the `.gitignore` files, providing an initial structure for your project.

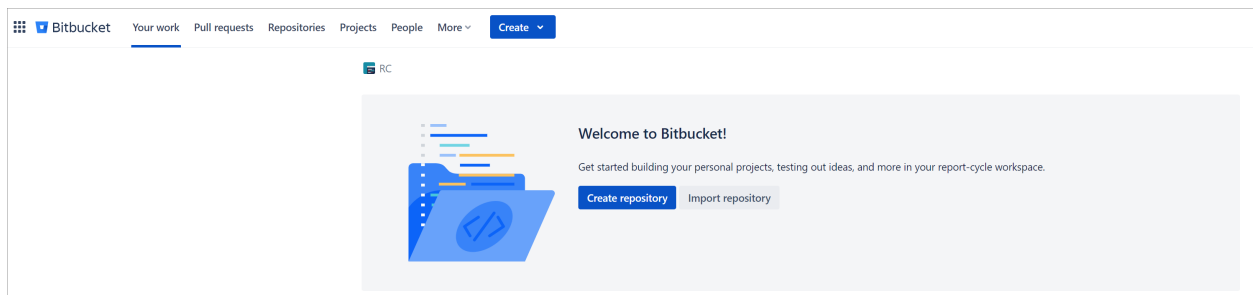


Figure 1: An empty Bitbucket account

2.2 Using SourceTree with Bitbucket

This section guides you through using SourceTree with Bitbucket repositories.

1. Connecting SourceTree to your Bitbucket account:

- Open SourceTree. If you haven't installed SourceTree, refer to Lab03 for installation instructions.
- [Connect SourceTree to your Bitbucket account](#). This may require generating an app password if you have two-factor authentication enabled. For guidance, see [Bitbucket's documentation on app passwords](#).

2. Cloning a Repository using SourceTree:

- Use the 'Clone' feature in SourceTree to clone your `Lab7` repository from Bitbucket to your local machine. Choose a suitable location on your local machine for the clone.
- Once cloned, you should see the `README.md` and `.gitignore` files in your local repository.

Create a workspace

A workspace is the place where you can store and share your code and content

Workspace name *

ENSF381

Workspace ID *

bitbucket.org/ ensf381-mjbza

This will be the URL for your workspace

☒ Keep this workspace private

Create Cancel

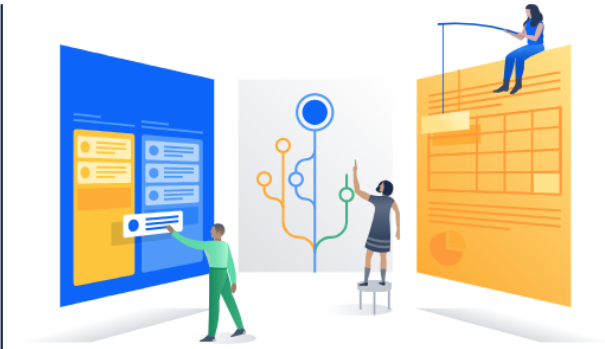


Figure 2: Creating a Bitbucket workspace

2.3 Inviting Collaborators to Your Bitbucket Repository

The process to add collaborators in Bitbucket is as follows:

- Log in to Bitbucket and navigate to your Lab7 repository.
- Go to the repository settings, then find and select **Repository permissions**. Figure 4 shows the form for adding new collaborators.
- Here, you can invite users by entering their Bitbucket username or email. Set the appropriate access level for them.
- Once you add them, they will receive an invitation to collaborate on your repository.
- Inform your collaborators to accept the invitation and clone the repository using SourceTree.

Create a new repository [Import repository](#)

Workspace ENSF381

Project name*

Repository name*

Access level ☒ Private repository
 Uncheck to make this repository public. Public repositories typically contain open-source code and can be viewed by anyone.

Include a README? Yes, with a template ▾

Default branch name

Include .gitignore? Yes (recommended) ▾

▾ Advanced settings

Description

Forking ▾

Language x ▾

[Create repository](#) [Cancel](#)

Figure 3: Creating a Bitbucket repository

Bitbucket Your work Pull requests Repositories Projects People More ▾ Create ▾

Q Search

JS Lab07

Back

GENERAL

Repository details

Repository permissions

Username aliases

SECURITY

ENSF381 / labs / Lab07 / Repository settings

Repository permissions [Add users or groups](#)

Repository permissions allow you to extend access beyond that already granted via [project permissions](#).

Q Permissions ▾ Access level ▾

Name	Permission	Access level	Actions
Administrators 1 member	Admin	Workspace	

Figure 4: Inviting users to a Bitbucket repository

2.4 Deliverable

1. Add the URL of your Bitbucket repository to your answer sheet. Ensure the repository is public and remains undeleted until the semester's end.
2. Include a screenshot from the *Repository permissions* table of your repository. The picture must show two collaborators and also include the address bar of your browser.

3 Exercise B (Creating React app)

You will build a small tic-tac-toe game in this lab. This lab is divided into several sections:

- [Setup for the Lab](#) will give you a **starting point** to follow the lab.
- [Overview](#) will teach you **the fundamentals** of React: components, props, and state.
- [Completing the Game](#) will teach you **the most common techniques** in React development.
- [Adding Time Travel](#) will give you a **deeper insight** into the unique strengths of React.

3.1 Setup for the Lab

3.1.1 Prerequisites

We will assume that you have some familiarity with HTML and JavaScript, but you should be able to follow along even if you are coming from a different programming language. We will also assume that you are familiar with programming concepts like functions, objects, arrays, and to a lesser extent, classes.

If you need to review JavaScript, we recommend reading [this guide](#). Note that we are also using some features from ES6 — a recent version of JavaScript. In this lab, we are using [arrow functions](#), [classes](#), [let](#), and [const](#) statements.

3.1.2 Initiating the React app

Here are the steps to follow:

1. Make sure you have a recent version of [Node.js](#) installed.
2. We normally use the following command to make a new react project; however, we do not use it now.

Listing 1: Command for creating new react app

```
1 npx create-react-app my-app
```

3. As the above command creates a new project folder and we have already initiated our project folder by cloning our remote repository (i.e., `lab7`), we use the following commands to create a React app inside the existing folder. In the following commands, we assumed your `lab7` folder is located in your user's home folder; change it based on your `lab7` location if it is needed.

Listing 2: Commands for creating new react app in an existing folder

```
1 cd ~/lab7
2 npx create-react-app ./
```

4. Run the original React app using `'npm start'`. It normally opens a new tab in your browser using the <http://localhost:3000/> address.
5. After confirming the React app works, stop the development server by using `'Ctrl + C'` twice.
6. Before changing the original generated code, commit and push the original files. After committing and pushing, compare the content of your remote repository with those in your local repository. There must be a `'node_modules'` in your local repository might not be pushed into your remote repository. Investigate why it is not pushed and why we must not push it into the remote repository. Reflect your answers into your answer sheet file.
7. Delete all files in the `src/` folder of the new project

Note:

Do not delete the src folder itself, just the original source files inside it. We will replace the default source files with examples for this project in the next step.

Listing 3: Commands for deleting contents of src folder

```
1 cd src
2
3 # If you are using a Mac or Linux:
4 rm -f *
5
6 # Or, if you are on Windows:
7 del *
8
9 # Then, switch back to the project/repository folder
10 cd ..
```

4. Add a file named `index.css` in the `src/` folder with the CSS code in the `style.css` file.
5. Add a file named `index.js` in the `src/` folder with the JS code in the `script.js` file.
6. Add these three lines to the top of `index.js` in the `src/` folder:

Listing 4: Can be found in Step1.txt File

```
1 import React from 'areact';
2 import ReactDOM from 'areact-dom/client';
3 import './index.css';
```

Now if you run `npm start` in the project folder and open <http://localhost:3000/> in the browser, you should see an empty tic-tac-toe field, similar to figure 5.

7. Commit and push changes to the repository.
8. Ask your colleague to pull your latest changes from the remote repository.
9. As you might guess, for running the code on the new machine, we need to install missing node modules that are necessary and have not been pushed to the repository. To install node modules, run the following commands:

Listing 5: Commands for running React project on new machine

```
1 # Change the directory to your project folder
2 cd ~/Lab7
3
4 # Install libraries
5 npm install
6
7 # Run the project
8 npm start
```

10. We do not repeat again that both members must participate in the coding activity and have some commits in the remote repository; otherwise, the group will not receive any marks.

3.1.3 Code Highlighter for JS codes (Optional)

We recommend following instructions to configure syntax highlighting for your editor.

Install the [vscode-language-babel](#) extension and follow the instructions.

There seems to be one other way to get the syntax highlighting working and you can learn more about it in the [Visual Studio Code docs](#).

3.2 Overview

Now that you are set up, let's get an overview of React!

3.2.1 What Is React?

React is a declarative, efficient, and flexible JavaScript library for building user interfaces. It lets you compose complex UIs from small and isolated pieces of code called “components”.

React has a few different kinds of components, but we will start with `React.Component` subclasses:

Listing 6: Sample React component definition

```
1 class ShoppingList extends React.Component {
2   render() {
3     return (
4       <div className="shopping-list">
5         <h1>Shopping List for {this.props.name}</h1>
6         <ul>
7           <li>Instagram</li>
8           <li>WhatsApp</li>
9           <li>Oculus</li>
10        </ul>
11      </div>
12    );
13  }
14 }
15
16 // Example usage: <ShoppingList name="Mark" />
```

We will get to the funny XML-like tags soon. We use components to tell React what we want to see on the screen. When our data changes, React will efficiently update and re-render our components.

Here, `ShoppingList` is a **React component class**, or **React component type**. A component takes in parameters, called **props** (short for “properties”), and returns a hierarchy of views to display via the `render` method.

The `render` method returns a *description* of what you want to see on the screen. React takes the description and displays the result. In particular, `render` returns a **React element**, which is a lightweight description of what to render. Most React developers use a special syntax called “JSX” which makes these structures easier to write. The `<div/>` syntax is transformed at build time to `React.createElement('div')`. The example above is equivalent to:

Listing 7: How React converts JSX

```
1 return React.createElement('div', {className: 'shopping-list'},
2   React.createElement('h1', /* ... h1 children ... */),
3   React.createElement('ul', /* ... ul children ... */)
4 );
```

If you are curious, `createElement()` is described in more detail in the [API reference](#), but we won't be using it in this lab. Instead, we will keep using JSX.

JSX comes with the full power of JavaScript. You can put *any* JavaScript expressions within braces inside JSX. Each React element is a JavaScript object that you can store in a variable or pass around in your program.

The `ShoppingList` component above only renders built-in DOM components like `<div/>` and ``. But you can compose and render custom React components too. For example, we can now refer to the whole shopping list by writing `<ShoppingList />`. Each React component is encapsulated and can operate independently; this allows you to build complex UIs from simple components.

3.2.2 Inspecting the Starter Code

Open `src/index.js` in your project folder (you have already touched this file during the setup).

This Starter Code is the base of what we are building. We have provided the CSS styling so that you only need to focus on learning React and programming the tic-tac-toe game.

By inspecting the code, you will notice that we have three React components:

- Square
- Board
- Game

The Square component renders a single `<button>` and the Board renders 9 squares. The Game component renders a board with placeholder values which we will modify later. There are currently no interactive components.

3.2.3 Passing Data Through Props

To get our feet wet, let's try passing some data from our Board component to our Square component.

We strongly recommend typing code by hand as you are working through the lab and not using copy and paste. This will help you develop muscle memory and a stronger understanding. However, if insist on copy-pasting, you can find the code of each code snippet in a different text file.

1. In Board's `renderSquare` method, change the code to pass a prop called `value` to the Square:

Listing 8: Can be found in Step2.txt File

```
1 class Board extends React.Component {
2   renderSquare(i) {
3     return <Square value={i} />;
4   }
5 }
```

2. Change Square's `render` method to show that value by replacing `/* TODO */` with `{this.props.value}`:

Listing 9: Can be found in Step3.txt File

```
1 class Square extends React.Component {
2   render() {
3     return (
4       <button className="square">
5         {this.props.value}
6       </button>
7     );
8   }
9 }
```

Before:

After: You should see a number in each square in the rendered output.

Congratulations! You have just “passed a prop” from a parent Board component to a child Square component. Passing props is how information flows in React apps, from parents to children.

3. Analyze how the numbers 0 to 8 are rendered in squeres. We expect you analyze the rendered HTML code, and reflect your understanding of the code up to this point.

Next player: X

Figure 5: Tictac board

Next player: X

0	1	2
3	4	5
6	7	8

Figure 6: Tictac Numbers

3.2.4 Making an Interactive Component

1. Let's fill the Square component with an "X" when we click it. First, change the button tag that is returned from the Square component's `render()` function to this:

Listing 10: Can be found in Step4.txt File

```
1 class Square extends React.Component {  
2   render() {  
3     return (  
4       <button className="square"  
5         onClick={function(e) { console.log('Clicked '+e.target.innerHTML); }}>  
6         {this.props.value}  
7       </button>  
8     );  
9   }  
10 }
```

2. If you click on a Square now, you should see 'Clicked i' in your browser's devtools console.
3. Make a screenshot from your devtools that includes some console logs of your clicks and the rendered output. Add the screenshot to your answer sheet file.

Note:

To save typing and avoid the confusing behavior of `this`, we will use the arrow function syntax for event handlers here and further below. Notice how with `onClick={() => console.log('xyz')}`, we

are passing a *function* as the `onClick` prop. React will only call this function after a click. Forgetting `() =>` and writing `onClick={console.log('xyz')}` is a common mistake, and would fire every time the component re-renders.

4. As a next step, we want the Square component to “remember” that it got clicked, and fill it with an “X” mark. To “remember” things, components use **state**. React components can have state by setting `this.state` in their constructors. `this.state` should be considered as private to a React component that it is defined in. Let’s store the current value of the Square in `this.state`, and change it when the Square is clicked. First, we will add a constructor to the class to initialize the state:

Listing 11: Can be found in Step5.txt File

```
1 class Square extends React.Component {
2   constructor(props) {
3     super(props);
4     this.state = {
5       value: null,
6     };
7   }
8
9   render() {
10    return (
11      <button className="square"
12        onClick={function(e) { console.log('Clicked ' + e.target.innerHTML); }}>
13        {this.props.value}
14      </button>
15    );
16  }
17 }
```

Note:

In [JavaScript classes](#), you need to always call `super` when defining the constructor of a subclass. All React component classes that have a `constructor` should start with a `super(props)` call.

5. Now we will change the Square’s `render` method to display the current state’s value when clicked:
 - Replace `this.props.value` with `this.state.value` inside the `<button>` tag.
 - Replace the `onClick={...}` event handler with `onClick={() => this.setState({value: 'X'})}`.
 - Put the `className` and `onClick` props on separate lines for better readability.

After these changes, the `<button>` tag that is returned by the Square’s `render` method looks like this:

Listing 12: Can be found in Step6.txt File

```
1 class Square extends React.Component {
2   constructor(props) {
3     super(props);
4     this.state = {
5       value: null,
6     };
7   }
8
9   render() {
10    return (
```

```

11     <button
12       className="square"
13       onClick={() => this.setState({value: 'X'})}
14     >
15       {this.state.value}
16     </button>
17   );
18 }
19 }

```

6. By calling `this.setState()` from an `onClick` handler in the Square's `render` method, we tell React to re-render that Square whenever its `<button>` is clicked. After the update, the Square's `this.state.value` will be 'X', so we will see the X on the game board. If you click on any Square, an X should show up.
7. Make a screenshot from the game board that all have 'X', and add the screenshot to your answer sheet file.
8. Commit and push your changes to the remote repository, use '*End of section 3.4.4*' as the comment, and change the roles with your colleague.

Note:

When you call `setState` in a component, React automatically updates the child components inside of it too.

3.2.5 Developer Tools

The React Devtools extension for [Chrome](#) and [Firefox](#) lets you inspect a React component tree with your browser's developer tools. Figure 7 illustrates the React Devtools tab in Chrome.

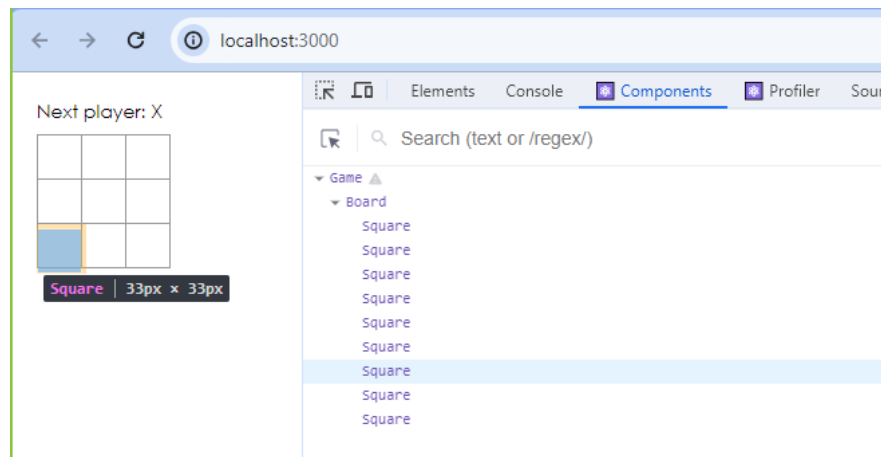


Figure 7: React Devtools in Chrome

The React DevTools let you check the props and the state of your React components. After installing React DevTools, you can right-click on any element on the page, click “Inspect” to open the developer tools, and the React tabs (“Components” and “Profiler”) will appear as the last tabs to the right. Use “Components” to inspect the component tree.

However, note there are a few extra steps to get it working with CodePen:

1. Log in or register and confirm your email (required to prevent spam).

2. Click the “Fork” button.
3. Click “Change View” and then choose “Debug mode”.
4. In the new tab that opens, the devtools should now have a React tab.

3.3 Completing the Game

We now have the basic building blocks for our tic-tac-toe game. To have a complete game, we now need to alternate placing “X”s and “O”s on the board, and we need a way to determine a winner.

3.3.1 Lifting State Up

Currently, each Square component maintains the game’s state. To check for a winner, we will maintain the value of each of the 9 squares in one location.

We may think that Board should just ask each Square for the Square’s state. Although this approach is possible in React, we discourage it because the code becomes difficult to understand, susceptible to bugs, and hard to refactor. Instead, the best approach is to store the game’s state in the parent Board component instead of in each Square. The Board component can tell each Square what to display by passing a prop, just like we did when we passed a number to each Square.

To collect data from multiple children, or to have two child components communicate with each other, you need to declare the shared state in their parent component instead. The parent component can pass the state back down to the children by using props; this keeps the child components in sync with each other and with the parent component.

Lifting state into a parent component is common when React components are refactored — let’s take this opportunity to try it out.

1. Add a constructor to the Board and set the Board’s initial state to contain an array of nine nulls corresponding to the nine squares:

Listing 13: Can be found in Step7.txt File

```
1 class Board extends React.Component {
2   constructor(props) {
3     super(props);
4     this.state = {
5       squares: Array(9).fill(null),
6     };
7   }
8
9   renderSquare(i) {
10    return <Square value={i} />;
11  }
12
13  // rest of the code
```

When we fill the board in later, the `this.state.squares` array will look something like this:

Listing 14: `this.state.squares` array

```
1 [
2   'O', null, 'X',
3   'X', 'X', 'O',
4   'O', null, null,
5 ]
```

2. The Board’s `renderSquare` method currently looks like this:

Listing 15: Current state of renderSquare method

```
1 renderSquare(i) {
2   return <Square value={i} />;
3 }
```

In the beginning, we passed the `value` prop down from the Board to show numbers from 0 to 8 in every Square. In a different previous step, we replaced the numbers with an “X” mark determined by Square’s own state. This is why Square currently ignores the `value` prop passed to it by the Board.

We will now use the prop passing mechanism again. We will modify the Board to instruct each individual Square about its current value (`'X'`, `'O'`, or `null`). We have already defined the `squares` array in the Board’s constructor, and we will modify the Board’s `renderSquare` method to read from it:

Listing 16: Can be found in Step8.txt File

```
1 renderSquare(i) {
2   return <Square value={this.state.squares[i]} />;
3 }
```

Each Square will now receive a `value` prop that will either be `'X'`, `'O'`, or `null` for empty squares.

- Next, we need to change what happens when a Square is clicked. The Board component now maintains which squares are filled. We need to create a way for the Square to update the Board’s state. Since state is considered to be private to a component that defines it, we cannot update the Board’s state directly from Square.

Instead, we will pass down a function from the Board to the Square, and we will have Square call that function when a square is clicked. We will change the `renderSquare` method in Board to:

Listing 17: Can be found in Step9.txt File

```
1 renderSquare(i) {
2   return (
3     <Square
4       value={this.state.squares[i]}
5       onClick={() => this.handleClick(i)}
6     />
7   );
8 }
```

Note:

We split the returned element into multiple lines for readability, and added parentheses so that JavaScript does not insert a semicolon after `return` and break our code.

- Now we are passing down two props from Board to Square: `value` and `onClick`. The `onClick` prop is a function that Square can call when clicked. We will make the following changes to Square:

- Replace `this.state.value` with `this.props.value` in Square’s `render` method
- Replace `this.setState()` with `this.props.onClick()` in Square’s `render` method
- Delete the `constructor` from Square because Square no longer keeps track of the game’s state

After these changes, the Square component looks like this:

Listing 18: Can be found in Step10.txt File

```
1 class Square extends React.Component {
2   render() {
```

```

3   return (
4     <button
5       className="square"
6       onClick={() => this.props.onClick()}
7     >
8       {this.props.value}
9     </button>
10  );
11  }
12 }

```

Note:

When a Square is clicked, the `onClick` function provided by the Board is called. Here's a review of how this is achieved:

1. The `onClick` prop on the built-in DOM `<button>` component tells React to set up a click event listener.
2. When the button is clicked, React will call the `onClick` event handler that is defined in Square's `render()` method.
3. This event handler calls `this.props.onClick()`. The Square's `onClick` prop was specified by the Board.
4. Since the Board passed `onClick={() => this.handleClick(i)}` to Square, the Square calls the Board's `handleClick(i)` when clicked.
5. We have not defined the `handleClick()` method yet, so our code crashes. If you click a square now, you should see a red error screen saying something like "this.handleClick is not a function".

Note:

The DOM `<button>` element's `onClick` attribute has a special meaning to React because it is a built-in component. For custom components like Square, the naming is up to you. We could give any name to the Square's `onClick` prop or Board's `handleClick` method, and the code would work the same. In React, it is conventional to use `on[Event]` names for props which represent events and `handle[Event]` for the methods which handle the events.

5. When we try to click a Square, we should get an error because we have not defined `handleClick` yet. We will now add `handleClick` to the Board class:

Listing 19: Can be found in Step11.txt File

```

1  class Board extends React.Component {
2    constructor(props) {
3      super(props);
4      this.state = {
5        squares: Array(9).fill(null),
6      };
7    }
8
9    handleClick(i) {
10     const squares = this.state.squares.slice();
11     squares[i] = 'X';
12     this.setState({squares: squares});
13   }

```

```

14
15   renderSquare(i) {
16     return (
17       <Square
18         value={this.state.squares[i]}
19         onClick={() => this.handleClick(i)}
20       />
21     );
22   }
23
24   render() {
25     const status = 'Next player: X';
26
27     return (
28       <div>
29         <div className="status">{status}</div>
30         <div className="board-row">
31           {this.renderSquare(0)}
32           {this.renderSquare(1)}
33           {this.renderSquare(2)}
34         </div>
35         <div className="board-row">
36           {this.renderSquare(3)}
37           {this.renderSquare(4)}
38           {this.renderSquare(5)}
39         </div>
40         <div className="board-row">
41           {this.renderSquare(6)}
42           {this.renderSquare(7)}
43           {this.renderSquare(8)}
44         </div>
45       </div>
46     );
47   }
48 }

```

After these changes, we are again able to click on the Squares to fill them, the same as we had before. However, now the state is stored in the Board component instead of the individual Square components. When the Board's state changes, the Square components re-render automatically. Keeping the state of all squares in the Board component will allow it to determine the winner in the future.

Since the Square components no longer maintain state, the Square components receive values from the Board component and inform the Board component when they are clicked. In React terms, the Square components are now **controlled components**. The Board has full control over them.

Note how in `handleClick`, we call `.slice()` to create a copy of the `squares` array to modify instead of modifying the existing array. We will explain why we create a copy of the `squares` array in the next section.

3.4 Why Immutability Is Important

In the previous code example, we suggested that you create a copy of the `squares` array using the `slice()` method instead of modifying the existing array. We will now discuss immutability and why immutability is important to learn.

There are generally two approaches to changing data. The first approach is to *mutate* the data by directly changing the data's values. The second approach is to replace the data with a new copy which has the desired changes.

3.4.1 Data Change with Mutation

Listing 20: Sample of Data Change with Mutation

```
1 var player = {score: 1, name: 'Jeff'};
2 player.score = 2;
3 // Now player is {score: 2, name: 'Jeff'}
```

3.4.2 Data Change without Mutation

Listing 21: Sample of Data Change without Mutation

```
1 var player = {score: 1, name: 'Jeff'};
2
3 var newPlayer = Object.assign({}, player, {score: 2});
4 // Now player is unchanged, but newPlayer is {score: 2, name: 'Jeff'}
5
6 // Or if you are using object spread syntax, you can write:
7 // var newPlayer = {...player, score: 2};
```

The end result is the same but by not mutating (or changing the underlying data) directly, we gain several benefits described below.

3.4.3 Complex Features Become Simple

Immutability makes complex features much easier to implement. Later in this lab, we will implement a “time travel” feature that allows us to review the tic-tac-toe game’s history and “jump back” to previous moves. This functionality is not specific to games — an ability to undo and redo certain actions is a common requirement in applications. Avoiding direct data mutation lets us keep previous versions of the game’s history intact, and reuse them later.

3.4.4 Detecting Changes

Detecting changes in mutable objects is difficult because they are modified directly. This detection requires the mutable object to be compared to previous copies of itself and the entire object tree to be traversed.

Detecting changes in immutable objects is considerably easier. If the immutable object that is being referenced is different than the previous one, then the object has changed.

3.4.5 Determining When to Re-Render in React

The main benefit of immutability is that it helps you build *pure components* in React. Immutable data can easily determine if changes have been made, which helps to determine when a component requires re-rendering.

You can learn more about `shouldComponentUpdate()` and how you can build *pure components* by reading [Optimizing Performance](#).

3.5 Function Components

We will now change the Square to be a **function component**.

In React, **function components** are a simpler way to write components that only contain a `render` method and do not have their own state. Instead of defining a class which extends `React.Component`, we can write a function that takes `props` as input and returns what should be rendered. Function components are less tedious to write than classes, and many components can be expressed this way.

1. Replace the Square class with this function. We have changed `this.props` to `props` both times it appears.

Listing 22: Can be found in Step12.txt File

```
1 function Square(props) {
2   return (
3     <button className="square" onClick={props.onClick}>
4       {props.value}
5     </button>
6   );
7 }
```

Note:

When we modified the Square to be a function component, we also changed `onClick={() => this.props.onClick()}` to a shorter `onClick={props.onClick}` (note the lack of parentheses on *both* sides).

3.5.1 Taking Turns

1. We now need to fix an obvious defect in our tic-tac-toe game: the “O”s cannot be marked on the board. We will set the first move to be “X” by default. We can set this default by modifying the initial state in our Board constructor:

Listing 23: Can be found in Step13.txt File

```
1 class Board extends React.Component {
2   constructor(props) {
3     super(props);
4     this.state = {
5       squares: Array(9).fill(null),
6       xIsNext: true,
7     };
8   }
9   // rest of the code
```

2. Each time a player moves, `xIsNext` (a boolean) will be flipped to determine which player goes next and the game’s state will be saved. We will update the Board’s `handleClick` function to flip the value of `xIsNext`:

Listing 24: Can be found in Step14.txt File

```
1 handleClick(i) {
2   const squares = this.state.squares.slice();
3   squares[i] = this.state.xIsNext ? 'X' : 'O';
4   this.setState({
5     squares: squares,
6     xIsNext: !this.state.xIsNext,
7   });
8 }
```

With this change, “X”s and “O”s can take turns. Try it!

2. Let’s also change the “status” text in Board’s `render` so that it displays which player has the next turn:

Listing 25: Can be found in Step15.txt File

```

1 render() {
2     const status = 'Next player: ' + (this.state.xIsNext ? 'X' : 'O');
3
4     return (
5         // the rest has not changed

```

After applying these changes, you should have this Board component:

Listing 26: Can be found in Step16.txt File

```

1 class Board extends React.Component {
2     constructor(props) {
3         super(props);
4         this.state = {
5             squares: Array(9).fill(null),
6             xIsNext: true,
7         };
8     }
9
10    handleClick(i) {
11        const squares = this.state.squares.slice();
12        squares[i] = this.state.xIsNext ? 'X' : 'O';
13        this.setState({
14            squares: squares,
15            xIsNext: !this.state.xIsNext,
16        });
17    }
18
19    renderSquare(i) {
20        return (
21            <Square
22                value={this.state.squares[i]}
23                onClick={() => this.handleClick(i)}
24            />
25        );
26    }
27
28    render() {
29        const status = 'Next player: ' + (this.state.xIsNext ? 'X' : 'O');
30
31        return (
32            <div>
33                <div className="status">{status}</div>
34                <div className="board-row">
35                    {this.renderSquare(0)}
36                    {this.renderSquare(1)}
37                    {this.renderSquare(2)}
38                </div>
39                <div className="board-row">
40                    {this.renderSquare(3)}
41                    {this.renderSquare(4)}
42                    {this.renderSquare(5)}
43                </div>
44                <div className="board-row">
45                    {this.renderSquare(6)}
46                    {this.renderSquare(7)}
47                    {this.renderSquare(8)}
48                </div>
49            </div>

```

```

50     );
51   }
52 }

```

3.5.2 Declaring a Winner

1. Now that we show which player's turn is next, we should also show when the game is won and there are no more turns to make. Copy this helper function and paste it at the end of the `index.js` file:

Listing 27: Can be found in Step17.txt File

```

1 function calculateWinner(squares) {
2   const lines = [
3     [0, 1, 2],
4     [3, 4, 5],
5     [6, 7, 8],
6     [0, 3, 6],
7     [1, 4, 7],
8     [2, 5, 8],
9     [0, 4, 8],
10    [2, 4, 6],
11  ];
12  for (let i = 0; i < lines.length; i++) {
13    const [a, b, c] = lines[i];
14    if (squares[a] && squares[a] === squares[b] && squares[a] === squares[c])
15      {
16        return squares[a];
17      }
18  }
19  return null;
20 }

```

Given an array of nine possible lines in a square board, this function will check for a winner and return 'X', 'O', or null as appropriate.

2. We will call `calculateWinner(squares)` in the Board's `render` function to check if a player has won. If a player has won, we can display text such as "Winner: X" or "Winner: O". We will replace the `status` declaration in Board's `render` function with this code:

Listing 28: Can be found in Step18.txt File

```

1 render() {
2   const winner = calculateWinner(this.state.squares);
3   let status;
4   if (winner) {
5     status = 'Winner: ' + winner;
6   } else {
7     status = 'Next player: ' + (this.state.xIsNext ? 'X' : 'O');
8   }
9
10  return (
11    // the rest has not changed

```

3. We can now change the Board's `handleClick` function to return early by ignoring a click if someone has won the game or if a Square is already filled:

Listing 29: Can be found in Step19.txt File

```

1 handleClick(i) {
2   const squares = this.state.squares.slice();
3   if (calculateWinner(squares) || squares[i]) {
4     return;
5   }
6   squares[i] = this.state.xIsNext ? 'X' : 'O';
7   this.setState({
8     squares: squares,
9     xIsNext: !this.state.xIsNext,
10  });
11 }

```

Congratulations! You now have a working tic-tac-toe game. And you have just learned the basics of React too. So *you are* probably the real winner here.

3.6 Adding Time Travel

As a final exercise, let's make it possible to “go back in time” to the previous moves in the game.

3.6.1 Storing a History of Moves

If we mutated the `squares` array, implementing time travel would be very difficult. However, we used `slice()` to create a new copy of the `squares` array after every move, and treated it as immutable. This will allow us to store every past version of the `squares` array, and navigate between the turns that have already happened. We will store the past `squares` arrays in another array called `history`. The `history` array represents all board states, from the first to the last move, and has a shape like this:

Listing 30: Sample data of history array

```

1 history = [
2   // Before first move
3   {
4     squares: [
5       null, null, null,
6       null, null, null,
7       null, null, null,
8     ]
9   },
10  // After first move
11  {
12    squares: [
13      null, null, null,
14      null, 'X', null,
15      null, null, null,
16    ]
17  },
18  // After second move
19  {
20    squares: [
21      null, null, null,
22      null, 'X', null,
23      null, null, 'O',
24    ]
25  },
26  // ...
27 ]

```

Now we need to decide which component should own the `history` state.

3.6.2 Lifting State Up, Again

We will want the top-level Game component to display a list of past moves. It will need access to the `history` to do that, so we will place the `history` state in the top-level Game component.

Placing the `history` state into the Game component lets us remove the `squares` state from its child Board component. Just like we “lifted state up” from the Square component into the Board component, we are now lifting it up from the Board into the top-level Game component. This gives the Game component full control over the Board’s data, and lets it instruct the Board to render previous turns from the `history`.

1. First, we will set up the initial state for the Game component within its constructor:

Listing 31: Can be found in Step20.txt File

```
1 class Game extends React.Component {
2   constructor(props) {
3     super(props);
4     this.state = {
5       history: [{
6         squares: Array(9).fill(null),
7       }],
8       xIsNext: true,
9     };
10  }
11
12  render() {
13    return (
14      <div className="game">
15        <div className="game-board">
16          <Board />
17        </div>
18        <div className="game-info">
19          <div>{/* status */}</div>
20          <ol>{/* TODO */}</ol>
21        </div>
22      </div>
23    );
24  }
25 }
```

2. Next, we will have the Board component receive `squares` and `onClick` props from the Game component. Since we now have a single click handler in Board for many Squares, we will need to pass the location of each Square into the `onClick` handler to indicate which Square was clicked. Here are the required steps to transform the Board component:

- (a) Delete the `constructor` in Board.
- (b) Replace `this.state.squares[i]` with `this.props.squares[i]` in Board’s `renderSquare`.
- (c) Replace `this.handleClick(i)` with `this.props.onClick(i)` in Board’s `renderSquare`.

The Board component now looks like this:

Listing 32: Can be found in Step21.txt File

```
1 class Board extends React.Component {
2   handleClick(i) {
3     const squares = this.state.squares.slice();
4     if (calculateWinner(squares) || squares[i]) {
5       return;
```

```

6     }
7     squares[i] = this.state.xIsNext ? 'X' : 'O';
8     this.setState({
9         squares: squares,
10        xIsNext: !this.state.xIsNext,
11    });
12 }
13
14 renderSquare(i) {
15     return (
16         <Square
17             value={this.props.squares[i]}
18             onClick={() => this.props.onClick(i)}
19         />
20     );
21 }
22
23 render() {
24     const winner = calculateWinner(this.state.squares);
25     let status;
26     if (winner) {
27         status = 'Winner: ' + winner;
28     } else {
29         status = 'Next player: ' + (this.state.xIsNext ? 'X' : 'O');
30     }
31
32     return (
33         <div>
34             <div className="status">{status}</div>
35             <div className="board-row">
36                 {this.renderSquare(0)}
37                 {this.renderSquare(1)}
38                 {this.renderSquare(2)}
39             </div>
40             <div className="board-row">
41                 {this.renderSquare(3)}
42                 {this.renderSquare(4)}
43                 {this.renderSquare(5)}
44             </div>
45             <div className="board-row">
46                 {this.renderSquare(6)}
47                 {this.renderSquare(7)}
48                 {this.renderSquare(8)}
49             </div>
50         </div>
51     );
52 }
53 }

```

3. We will update the Game component's `render` function to use the most recent history entry to determine and display the game's status:

Listing 33: Can be found in Step22.txt File

```

1 render() {
2     const history = this.state.history;
3     const current = history[history.length - 1];
4     const winner = calculateWinner(current.squares);

```

```

5   let status;
6   if (winner) {
7     status = 'Winner: ' + winner;
8   } else {
9     status = 'Next player: ' + (this.state.xIsNext ? 'X' : 'O');
10  }
11
12  return (
13    <div className="game">
14      <div className="game-board">
15        <Board
16          squares={current.squares}
17          onClick={i => this.handleClick(i)}
18        />
19      </div>
20      <div className="game-info">
21        <div>{status}</div>
22        <ol>{ /* TODO */}</ol>
23      </div>
24    </div>
25  );
26 }

```

4. Since the Game component is now rendering the game's status, we can remove the corresponding code from the Board's **render** method. After refactoring, the Board's **render** function looks like this:

Listing 34: Can be found in Step23.txt File

```

1  render() {
2    return (
3      <div>
4        <div className="board-row">
5          {this.renderSquare(0)}
6          {this.renderSquare(1)}
7          {this.renderSquare(2)}
8        </div>
9        <div className="board-row">
10         {this.renderSquare(3)}
11         {this.renderSquare(4)}
12         {this.renderSquare(5)}
13       </div>
14       <div className="board-row">
15         {this.renderSquare(6)}
16         {this.renderSquare(7)}
17         {this.renderSquare(8)}
18       </div>
19     </div>
20   );
21 }

```

5. Finally, we need to move the **handleClick** method from the Board component to the Game component. We also need to modify **handleClick** because the Game component's state is structured differently. Within the Game's **handleClick** method, we concatenate new history entries onto **history**.

Listing 35: Can be found in Step24.txt File

```

1  handleClick(i) {

```

```

2   const history = this.state.history;
3   const current = history[history.length - 1];
4   const squares = current.squares.slice();
5   if (calculateWinner(squares) || squares[i]) {
6     return;
7   }
8   squares[i] = this.state.xIsNext ? 'X' : 'O';
9   this.setState({
10     history: history.concat([
11       squares,
12     ]),
13     xIsNext: !this.state.xIsNext,
14   });
15 }

```

Note:

Unlike the array `push()` method you might be more familiar with, the `concat()` method does not mutate the original array, so we prefer it.

At this point, the Board component only needs the `renderSquare` and `render` methods. The game's state and the `handleClick` method should be in the Game component.

3.6.3 Showing the Past Moves

Since we are recording the tic-tac-toe game's history, we can now display it to the player as a list of past moves.

We learned earlier that React elements are first-class JavaScript objects; we can pass them around in our applications. To render multiple items in React, we can use an array of React elements.

In JavaScript, arrays have a `map()` method that is commonly used for mapping data to other data, for example:

Listing 36: Example of using map method on arrays

```

1 const numbers = [1, 2, 3];
2 const doubled = numbers.map(x => x * 2); // [2, 4, 6]

```

1. Using the `map` method, we can map our history of moves to React elements representing buttons on the screen, and display a list of buttons to “jump” to past moves. Let's `map` over the `history` in the Game's `render` method:

Listing 37: Can be found in Step25.txt File

```

1 render() {
2   const history = this.state.history;
3   const current = history[history.length - 1];
4   const winner = calculateWinner(current.squares);
5
6   const moves = history.map((step, move) => {
7     const desc = move ?
8       'Go to move #' + move :
9       'Go to game start';
10    return (
11      <li>
12        <button onClick={() => this.jumpTo(move)}>{desc}</button>
13      </li>
14    );
15  });

```

```

15   });
16
17   let status;
18   if (winner) {
19     status = 'Winner: ' + winner;
20   } else {
21     status = 'Next player: ' + (this.state.xIsNext ? 'X' : 'O');
22   }
23
24   return (
25     <div className="game">
26       <div className="game-board">
27         <Board
28           squares={current.squares}
29           onClick={(i) => this.handleClick(i)}
30         />
31       </div>
32       <div className="game-info">
33         <div>{status}</div>
34         <ol>{moves}</ol>
35       </div>
36     </div>
37   );
38 }

```

As we iterate through `history` array, `step` variable refers to the current `history` element value, and `move` refers to the current `history` element index. We are only interested in `move` here, hence `step` is not getting assigned to anything.

For each move in the tic-tac-toe game's history, we create a list item `` which contains a button `<button>`. The button has an `onClick` handler which calls a method called `this.jumpTo()`. We **have not** implemented the `jumpTo()` method yet. For now, we should see a list of the moves that have occurred in the game and a warning in the developer tools console that says:

Warning:

Each child in an array or iterator should have a unique "key" prop. Check the render method of "Game".

Let's discuss what the above warning means.

3.6.4 Picking a Key

When we render a list, React stores some information about each rendered list item. When we update a list, React needs to determine what has changed. We could have added, removed, re-arranged, or updated the list's items.

Imagine transitioning from:

Listing 38: Previous state

```

1 <li>Alexa: 7 tasks left</li>
2 <li>Ben: 5 tasks left</li>

```

to:

Listing 39: current state

```

1 <li>Ben: 9 tasks left</li>
2 <li>Claudia: 8 tasks left</li>
3 <li>Alexa: 5 tasks left</li>

```

In addition to the updated counts, a human reading this would probably say that we swapped Alexa and Ben's ordering and inserted Claudia between Alexa and Ben. However, React is a computer program and does not know what we intended. Because React cannot know our intentions, we need to specify a *key* property for each list item to differentiate each list item from its siblings. One option would be to use the strings `alexa`, `ben`, `claudia`. If we were displaying data from a database, Alexa, Ben, and Claudia's database IDs could be used as keys.

Listing 40: An example of making keys for React

```
1 <li key={user.id}>{user.name}: {user.taskCount} tasks left</li>
```

When a list is re-rendered, React takes each list item's key and searches the previous list's items for a matching key. If the current list has a key that did not exist before, React creates a component. If the current list is missing a key that existed in the previous list, React destroys the previous component. If two keys match, the corresponding component is moved. Keys tell React about the identity of each component which allows React to maintain state between re-renders. If a component's key changes, the component will be destroyed and re-created with a new state.

`key` is a special and reserved property in React (along with `ref`, a more advanced feature). When an element is created, React extracts the `key` property and stores the key directly on the returned element. Even though `key` may look like it belongs in `props`, `key` cannot be referenced using `this.props.key`. React automatically uses `key` to decide which components to update. A component cannot inquire about its `key`.

It is strongly recommended that you assign proper keys whenever you build dynamic lists. If you do not have an appropriate key, you may want to consider restructuring your data so that you do.

If no key is specified, React will present a warning and use the array index as a key by default. Using the array index as a key is problematic when trying to re-order a list's items or inserting/removing list items. Explicitly passing `key={i}` silences the warning but has the same problems as array indices and is not recommended in most cases.

Keys do not need to be globally unique; they only need to be unique between components and their siblings.

3.6.5 Implementing Time Travel

In the tic-tac-toe game's history, each past move has a unique ID associated with it: it is the sequential number of the move. The moves are never re-ordered, deleted, or inserted in the middle, so it is safe to use the move index as a key.

1. In the Game component's `render` method, we can add the key as `<li key={move}>` and React's warning about keys should disappear:

Listing 41: Can be found in Step26.txt File

```
1 const moves = history.map((step, move) => {
2   const desc = move ?
3     'Go to move #' + move :
4     'Go to game start';
5   return (
6     <li key={move}>
7       <button onClick={() => this.jumpTo(move)}>{desc}</button>
8     </li>
9   );
10 });
```

Clicking any of the list item's buttons throws an error because the `jumpTo` method is undefined. Before we implement `jumpTo`, we will add `stepNumber` to the Game component's state to indicate which step we are currently viewing.

2. First, add `stepNumber: 0` to the initial state in Game's constructor:

Listing 42: Can be found in Step27.txt File

```
1 class Game extends React.Component {
2   constructor(props) {
3     super(props);
4     this.state = {
5       history: [{
6         squares: Array(9).fill(null),
7       }],
8       stepNumber: 0,
9       xIsNext: true,
10    };
11  }
```

3. Next, we will define the `jumpTo` method in `Game` to update that `stepNumber`. We also set `xIsNext` to `true` if the number that we are changing `stepNumber` to is even:

Listing 43: Can be found in Step28.txt File

```
1 handleClick(i) {
2   // this method has not changed
3 }
4
5 jumpTo(step) {
6   this.setState({
7     stepNumber: step,
8     xIsNext: (step % 2) === 0,
9   });
10 }
11
12 render() {
13   // this method has not changed
14 }
```

Notice in `jumpTo` method, we have not updated `history` property of the state. That is because state updates are merged or in more simple words React will update only the properties mentioned in `setState` method leaving the remaining state as is. For more info [see the documentation](#).

4. We will now make a few changes to the `Game`'s `handleClick` method which fires when you click on a square.
 - (a) The `stepNumber` state we have added reflects the move displayed to the user now. After we make a new move, we need to update `stepNumber` by adding `stepNumber: history.length` as part of the `this.setState` argument. This ensures we do not get stuck showing the same move after a new one has been made.
 - (b) We will also replace reading `this.state.history` with `this.state.history.slice(0, this.state.stepNumber+1)`. This ensures that if we “go back in time” and then make a new move from that point, we throw away all the “future” history that would now be incorrect.

Listing 44: Can be found in Step29.txt File

```
1 handleClick(i) {
2   const history = this.state.history.slice(0, this.state.stepNumber + 1);
3   const current = history[history.length - 1];
4   const squares = current.squares.slice();
5   if (calculateWinner(squares) || squares[i]) {
```

```

6     return;
7   }
8   squares[i] = this.state.xIsNext ? 'X' : 'O';
9   this.setState({
10     history: history.concat([
11       {
12         squares: squares
13       }
14     ]),
15     stepNumber: history.length,
16     xIsNext: !this.state.xIsNext,
17   });
18 }

```

5. Finally, we will modify the Game component's `render` method from always rendering the last move to rendering the currently selected move according to `stepNumber`:

Listing 45: Can be found in Step30.txt File

```

1 render() {
2   const history = this.state.history;
3   const current = history[this.state.stepNumber];
4   const winner = calculateWinner(current.squares);
5
6   // the rest has not changed

```

If we click on any step in the game's history, the tic-tac-toe board should immediately update to show what the board looked like after that step occurred.

3.7 Deliverable

1. First, format your `index.js` file with the VSC formatter, then commit and push your final code to the repository. Make sure you have done all the steps exactly as we explained in this instruction.
2. Play with your game and record a gif movie. Commit and push your gif file into your repository. Make sure the gif file is accessible publicly and your repository is not private.
3. Copy and paste the link to your gif file into your answer sheet file.
4. Play the game to represent the following state, and make a screenshot from your game board and add it to your answer sheet file.

Listing 46: Expected state for the final screenshot

```

1 [
2   ['X', 'O', 'X',
3    'O', 'X', ' ',
4    'X', 'O', ' '],
5 ]

```

This is the end of Lab 7.

Note:

This lab has been modified for this course based on the [Intro to React](#) tutorial.