Welcome to the React Web Development Spec Trial Day

Introduction to the React Specialization

Background

React is a JavaScript library created and maintained by Meta (Facebook). It was originally released in 2013 and has grown to be one of the top JavaScript libraries for building web applications. React is a component-based library; components are isolated pieces of UI and related functionality which can be reused. This makes React very customizable and enjoyable to build with.

Apps that use React

- Facebook
- Twitter
- Pinterest
- UberEats
- AirBnB
- Netflix
- Nike
- DropBox
- Discord
- PayPal

Introduction to Specializations

During Specializations, you will take a huge step towards your career in tech by not only attaining skills in a particular technology area or stack, but also by getting accustomed to a new work style that resembles the way you will work and learn in your first job. Your coding bootcamp experience is technically over, and now you will begin your journey within the technology industry.

Throughout your career as a React Web Developer, you will need to continue to learn and gain new skills. Developers will do this by carefully researching new technologies, following online video tutorials, and practicing learned concepts. We designed the Specializations program to mimic that learning experience. To do this, we've leveraged a combination of video learning and carefully curated exercises. We also provide support through multiple avenues including: daily stand-ups with your Tech Lead, weekly review sessions, weekly one-on-one meetings with your Subject Matter Expert (SME), and queue support during your class hours.

While this new learning environment will help prepare you for your first role and continued learning, it will also present unique challenges for you to conquer. Some of these challenges include: time management, self-awareness, and video fatigue. Overcoming these challenges at Devmountain, while you have support, will help build the foundation for your growth as a web developer.

What You Will Learn

This course prepares students with the knowledge and skills to start a career as a React Web Developer, providing all requisite knowledge of front-end web component design, data flow, dynamic user experiences, server structure, and agile software development.

In this course, you will learn about the following technologies:

- JavaScript (ES6 and beyond)
- React
- Axios
- Node
- Express
- Sequelize
- React Router
- React Redux

Staff During Specializations

Tech leads provide guidance, industry knowledge, and leadership during Specializations. They are the primary leader throughout the day for students and ensure that students are progressing with the support that they need. Tech Leads manage 1-2 technical programs and communicate directly with Subject Matter Experts to escalate student questions. They run daily Scrum Meetings, check in with students frequently, and monitor student progress. They also help to maintain and update curriculum for their area(s).

Fun Fact: The role of the tech lead is modeled after the Team Lead in the tech industry. A Team Lead is traditionally someone who is both an individual contributor but also coordinating with team members and tracking and driving overall progress. While Team Leads are proactive in their communication, it's also important for the rest of the team to be proactive in keeping the Team Lead up to date on blockers, progress, and other general information. Tech leads do this and that…

Subject Matter Experts answer student questions and provide feedback to students on their comprehension of the material during weekly meetings. You can expect at minimum one hour of one-on-one time with a SME each week during specialization, in addition to any escalated questions you have the SMEs might answer either individually or with the group based on collaboration with the Tech Lead.

Fun Fact: The SME role is modeled after Senior Engineers and SMEs (same name) within the tech industry. In a real engineering org, SMEs and Senior Engineers are called upon to enable mid or junior level folks when they run into blockers. As part of Specializations, you have the oppportunity to work with real Subject Matter Experts with years of experience in the tech industry. Not only will you learn things from them, you will also learn how to advocate for yourself effectively. Working with SMEs on a less frequent basis than staff during Foundations will enable you to build independence and confidence that will be invaluable when you begin your first role within the tech industry.

Your Web Dev Team

Below are some of the amazing staff that you'll be working with during Specs!

Brady Bott - Web Dev Tech Lead

- Get to Know Brady: *I workout frequently, and do some personal training on the side. I also hike, and play the occasional video game. Reading is an all time favorite.*
- Advice from Brady: *To be good, it's equal parts logic and style. Come in with creativity.*
- Favorite thing about work: *Seeing students grow and understand concepts! It's SO cool to see them go from struggling, to understanding and even teaching other students.*
- View Linkedin

Scott Sutherland - Web Dev Tech Lead

- Get to Know Scott: *My favorite things are LOTR, reading, gaming, movies and motorcycles! I got into coding because of how many opportunities there are, and discovered a passion for technology and teaching along the way.*
- Advice from Scott: *Focus on completing the unit projects, and use what you learn from those in your own mini-projects. So much of coding is just using something until it makes sense.*
- Favorite thing about work: *My favorite thing about my job is the lightbulb moments. DM was difficult for me as a student, and those moments where things finally clicked after days or weeks of trying felt so good. It feels just as satisfying to see other people go through the same process and gain a solid understanding of core coding concepts.*
- View Linkedin

Jon White - Web Subject Matter Expert

- 4 years experience with Javascript
- Currently **Software Developer at Alta Ski Area**, previously with MX in Lehi, UT
- DM Alum!
- View Linkedin

Alex Crowe - Web Subject Matter Expert

- 2 years experience with Javascript
- Currently **Web Developer at 1-800-Plumber + Air**
- Works with React / Express / MongoDB / SQL / Stripe.js
- View Linkedin

Jason Jazzar - Web Subject Matter Expert

- Background: I started using JavaScript in 2013 when I began my Computer Science degree at Florida State University. I come from a Windows background and prior to JavaScript, I was using C# to build Windows desktop applications.
- How he got into tech: I found a passion for technology at a young age. As I was growing up, I enjoyed playing video games on the Xbox 360. I think the first “real” app I coded was at 10 years old, which was a bot used to play instant win games. My interest in technology has grown exponentially since then and I still enjoy learning about new technological advancements today.
- Career: I have been fortunate enough to work at many diverse places throughout my career. I have worked on development teams as an employee, contractor, and freelancer ranging from Fortune 500 companies to small startups in the fields of banking, professional services, government consulting, and healthcare. I am currently a contract developer and mentor at DevMountain. In fact, mentoring the next generation of web developers is one of the most rewarding aspects of the work that I do!
- View Linkedin

Trial

Structure

1. Watch content on Udemy to familiarize yourself with the basics of React.
2. Complete the exercises.
3. Reflect on your experience.

Watch

The course you'll be watching is made up of modules, each of which contain multiple videos. For this trial, watch every module but the last one - Augmenting Features and Tooling.

You can find the course here.

Overview

As an intro to React, you'll be working on two exercises today. In the first, you'll be creating an app from the ground up! The second project is a series of tasks where you'll be adding functionality to a provided app using React.

Recall that React is a front-end library built with JavaScript. Parts of React will feel very familiar, while others will be completely new. You'll also be working with JSX, which resembles HTML, but has it's own rules and practices. Even though you might feel more comfortable with this content than other Specs because of these similarities, make sure you give equal consideration to your other options. Also, don't put too much pressure on yourself while you work on these exercises. The code should be recognizable, but you're not a React wizard yet!

Part One

Intro

Welcome to your first React Application! Here we will guide you through starting a React application, creating new components, using state to manage variables, and passing info to other components via props! Let’s go ahead and get started!

Step One: Environment Setup

Let's go ahead and get you started! For this step, you will be creating the environment we will be using in the following steps.

1. Open up your zsh or bash terminal, and proceed to cd into your directory you will want this project's folder to be in, for example: ***/documents/foundations/projects***
2. Now that you are where you want this project to be located, run the following command:

```
npx create-react-app@latest intro-to-react
```

This command will create a React app for us! This is especially handy, as many React applications require the same boilerplate structure for them the run. The command is broken down like this:

a. ***npx*** stands for Node Package Execute, and is able to execute packages from the npm registry (such as creating a react app)

b. ***create-react-app@latest*** is the command indicating that we want the package that provides React Application boilerplate code, and we want the latest version of the package.

c. ***intro-to-react*** is simply the name we are giving to this project. You can replace it with anything you wish! If you are making a social media app for cats, you could call it ***kitten-twitter*** if you wanted. You cannot use capital letters, so hyphens are suggested.

Wait for the command to finish running. when you see the phrase "Happy Hacking!" then you know your project is ready!

1. Let's now run the command ***code intro-to-react*** as this will bring the project up into our VS Code.
2. After running the command, you should have your VS Code pop up with your new ***intro-to-react*** project! You will see MANY files in here. Here's a quick overview of what each of these files (and one folder) is for.

    - ***README.md****: explains what the project is*
    - ***node_modules/****: the folder that stores all your dependencies*
    - ***package.json****: config file*
    - ***public/index.html****: the main file that is served, all the JavaScript is tied back to this file*
    - ***public/favicon.ico****: the React icon that shows up in your browser's tab*
    - ***src/App.css****: the CSS that's imported into* ***App.js***
    - ***src/App.js****: the main component of your app!*
    - ***src/App.test.js****: a template test file*

- ***src/index.css**: global CSS for your app*
- ***src/index.js**: the JavaScript entry point, meaning that this is the main JS file that runs and it's what renders your App component to the page you see in the browser*
- ***src/logo.svg**: this is the React logo that you'll see displayed when you open the app*

3. Before we begin changing any code, we want to begin to see our application in the browser. Open up the VS Code terminal and run the command ***npm start*** which will start your application in the browser. You'll know it worked when you see the spinning React icon.

Note: App not opening?

Ensure that you are in the correct directory when you run ***npm start***. You should be in the root folder of your React app.

Run ***pwd*** and ***ls*** if you're not sure. When you run ***pwd***, you should see something like: ***/users/Mia/documents/foundations/projects/intro-to-react***, depending on where you're storing the code. And when you run ***ls***, you should see ***README.md***, ***node_modules***, ***package.json***, ***public***, and ***src***.

Step Two: Creating Your First Component

1. We will now start the process of making the is application our very own! Navigate to your ***/src/App.js*** file, and inside you will see a lot of JSX that constitutes what you see in the browser. We want to remove most of the code inside! Remove everything inside of the *** *** tag, and then add your own ***<h1>*** tag that tells us the current component. It should look like this when you are done:

```
return (
    <div className="App">
        <h1>App Component</h1>
    </div>
)
```

2. We will now create our first component! Create a new file called ***Child.jsx*** located inside of the ***src*** folder. Now open it up, and let's write our first component. On line 1, type ***import React from 'react'*** then hit enter twice so that we are on line 3. On line 3, declare a function called "Child" and inside of the curly brackets, write a return that returns a *** *** tag that contains an ***<h2>*** tag. The ***<h2>*** should read "Child Component". Finally, 2 lines AFTER the Child function ends (after the curly brackets) write the code ***export default Child***. It should look like this:

```
import React from 'react'

function Child() {
    return (
        <div>
            <h2>Child Component</h2>
        </div>
    )
}

export default Child
```

3. Next, navigate back to ***src/App.js*** and toward the top, we will now import the Child component by writing ***import Child from './Child.jsx'***. Now that we have the Child component inside, we can use it! Put the Child component directly under the ***<h1>*** tag, using this syntax: ***<Child />***. Your code should now look like this:

```
import React from 'react'
import Child from './Child.jsx'

...

return (
    <div className="App">
        <h1>App Component</h1>
        <Child />
    </div>
)
```

If performed correctly, you will now see both “App Component” and “Child Component” on the browser in front of you. You have successfully created your first component and displayed it on the screen!

Step Three: State

1. Let's interact with the useState hook now. Navigate to ***src/App.js*** and at the top we need to import ***useState***. Next, we will declare our state and setter. This is done in the lines directly under our function declaration. We will type ***const [input, setInput] = useState("")***. As a reminder, the useState hook creates two variables. One is a value, the other is a setter to set that value. Your code should now look like this:

```
import React, {useState} from 'react'
import Child from './Child.jsx'

function App() {
    const [input, setInput] = useState("")

    return (
        <div className="App">
            <h1>App Component</h1>
            <Child />
        </div>
    )
}

export default App
```

2. Let's add an ***<input>*** tag to out return statement. Underneath the ***<h1>*** tag, place an ***<input>*** tag with ***type="text"*** and ***placeholder="Type Something Cool"*** as attributes. Next, we will add something that is uniqe to React, the ***onChange*** attribute. In vanilla JS, we ahve to use document query selectors to grab items and add event listeners to them. In react, we can write them directly onto the element itself. So, write the following piece of code as an attribute onto your ***<input>*** tag: ***onChange={(e) => setInput(e.target.value)}***. Your code should look like this:

```
<div className="App">
    <h1>App Component</h1>
    <input
        type="text"
        placeholder="Type Something Cool"
        onChange={(e) => setInput(e.target.value)} />
    <Child />
</div>
```

*Let's back up and break this down: when using event attributes such as **onChange** or **onClick** you can either run a function, or write an arrow function directly inside of the curly brackets. Here, we are taking event (shorthanded to "e") and then setting our input state to the value of the event target, i.e. the input field value. Cool!*

3. But we aren't done yet. We may be changing our state, but we aren't displaying it anywhere. Above our ***<input>*** tag, create an ***<h3>*** tag that contains the following: ***User Input: {input}***. Your code should look like this:

```
<div className="App">
    <h1>App Component</h1>
    <h3>User Input: {input}</h3>
    <input
        type="text"
        placeholder="Type Something Cool"
        onChange={(e) => setInput(e.target.value)} />
    <Child />
</div>
```

Now we can try it out! You should be able to type something in the input field, and it now shows on the screen whenever you type! Pretty sweet!

Step Four: Passing Props

1. It's pretty cool that we have our input state displaying here, but what if we wanted our ***Child*** component to display that information? Well we can, thanks to props! Let's navigate to ***/src/App.js*** and give the ***<Child />*** component a prop. A prop is written similar to an html attribute. We will give it the prop of ***userInput={input}***. Your code should look like this:

```
<Child userInput={input} />
```

2. Now navigate to ***src/Child.jsx*** and we need to pass ***props*** in as a parameter to the Child function. Inside of the parenthesis write the word ***props*** . Now let's add an ***<h3>*** tag underneath the ***<h2>*** tag, the contents of it should be a JSX literal of ***props.userInput***. Your code should look like this now:

```
import React from 'react'

function Child(props) {
    return (
        <div>
            <h2>Child Component</h2>
            <h3>{props.userInput}</h3>
        </div>
    )
}

export default Child
```

Part Two

Intro

In this portion of the exercise, you'll be working with an app that displays a list of robots. Users can name their robots and a profile picture is randomly generated for each robot. Right now, however, the app is incomplete and it's your job to finish it!

In ***App.js***, we have two portions of state - ***input*** and ***robots***. The first is a variable that we will use to store what user's type; the latter is an array which will store robot objects. We will then loop that list and return individual JSX for each robot object.

In this simple app, we are storing all of state in the ***App*** component, which is the parent of our other components. This allows for us to pass data and functionality down to the children components, which is the only way that data can flow in React. With all of the logic stored in a parent, we know that data can get to where it needs to go. Otherwise, we would need to use a library to pass data from sibling to sibling.

Setup

1. Download the directory for this exercise here. (These are the same materials from the link on Frodo's home page or exercises page.)
2. In your terminal, ***cd*** into that folder and then into the ***robots*** folder inside. Open the project in VS Code with the ***code .*** command.
3. This project was originally made with ***create-react-app*** but doesn't currently have everything it needs to run. Since you downloaded the files, the ***node_modules*** were not included. Run ***npm install*** to get the needed dependencies.
4. Test that the app will show up in the browser by running ***npm start***.

Step One: Capturing User Input

Right now, there should be a form showing up on the left hand side of the page. But if you try to add a robot, all that happens is a page reload! Let's fix that.

1. Open up ***src/components/Form.js*** and ***src/App/.js*** in VS Code.
2. In ***App.js***, take a look at the ***handleInput*** and ***handleRobotAdd*** functions. These are both going to be very important in getting our form to work. Right now, all ***handleInput*** does is log a string to the console. Let's work on getting it hooked up to the right event and then we'll change what it does.
3. We want ***handleInput*** to execute when a ***change*** event occurs on the ***<input>*** in our form. But this function is stored in ***App***, not ***Form***, so we'll use props to pass it down. If you look in the ***return*** statement of the ***App*** function, you'll see that we are rendering the ***<Form/>*** component as a child. Inside ***Form***'s brackets, add a prop named ***handleInput*** and set its value to ***{handleInput}***. Your code should look like this now:

```
// App.js return statement

return (
    <div className='app'>
        <Header/>
        <main>
            <Form handleInput={handleInput}/>
            <RobotList />
        </main>
    </div>
)
```

   *The **handleInput** that's in curly braces is referring to the function above. The one to the left of the equals sign is the name that we are giving the prop.*

4. In ***Form.js***, we now need to set up this function to be able to read ***props***. We can do this by passing ***props*** in as an argument, go ahead and do that now. Your code should look like this:

```
const Form = (props) => {...
```

5. Now we want to call this function when the input field is typed in. This is similar to HTML, but with some syntactical changes for JSX. Let's add ***onChange={props.handleInput}*** onto our ***<input>*** element.
6. Make sure both ***App.js*** and ***Form.js*** are saved and that ***npm start*** is running. Look at your app in the browser and open the dev tools so that you can view console logs. Start typing in the field and you should see every time the ***handleInput*** function is firing! This is pretty cool, but we want the ***handleInput*** function to change the value of ***input*** that is part of state for the ***App*** component. Let's change that now.

7. In ***App.js***, change the ***handleInput*** function to use the ***setInput*** function to save the value of whatever is typed to the ***input*** variable. We can get that value by using the event object - ***event.target.value*** Recall that the ***setInput*** function is a setter function that is returned from the ***useState*** hook and is the only way that we should change the value of the ***input*** variable. Once you change your function, the code should look like this:

```
const handleInput = (evt) => {
    setInput(evt.target.value)
}
```

Step Two: Handling Form Submissions

Now that we're able to store the user's input on state, we want to be able to save it to our ***robots*** array that we are setting up with the ***useState*** hook. There is a function already set up for you that can do just that – all you need to do is connect it to the submit event on the form.

1. In ***App.js***, read over the ***handleRobotAdd*** function and try to figure out what it's doing. Click the arrow below to read an explanation of the function.

   ▼ *Click to hide*

   - ***e.preventDefault()****: prevents the default form behavior from happening*
   - ***let robot = {...}****: this is creating a new robot using the* ***input*** *value from state, and a URL from a website that returns random robot pictures*
   - ***setRobots([...robots, robot])****: using the* ***setRobots*** *setter function to set the value of* ***robots*** *to a copy of what it is currently with the new* ***robot*** *we just made added on to the end*
   - ***setInput('')****: sets the value of* ***input*** *back to an empty string*

2. Now let's pass this function down to ***Form*** so that we can use it there. In the ***return*** of ***App***, add ***handleRobotAdd={handleRobotAdd}*** to the ***Form*** component. This will add it as a prop that we can access in the ***Form*** component's code. Your code should look like this now:

```
// App.js return statement

return (
    <div className='app'>
        <Header/>
        <main>
            <Form handleInput={handleInput} handleRobotAdd={handleRobotAdd}/>
            <RobotList />
        </main>
    </div>
)
```

3. Back in ***Form.js***, we'll set up the submit event. Inside the opening ***<form>*** tag, add the following code: ***onSubmit={props.handleRobotAdd}***. We already set up the function to expect props, so now all we have to do is access the function (***handleRobotAdd***) that we assigned to be stored on the ***props*** object. Now your code should look like this:

```
// Form.js return statement

return (
    <form onSubmit={props.handleRobotAdd}>
        ...
```

```
    </form>
)
```

4. If you'd like to see how your code is working, add ***console.log(robots)*** in ***App.js*** just above the ***return*** and test your form in the browser. **The robots won't show up yet, but you'll be able to see the data in the console.**

5. Finally, let's pass one more prop down to ***Form***. In the ***return*** of ***App.js***, add one last prop to the ***Form*** component: ***input={input}***. And in ***Form.js***, add ***value={props.input}*** to the ***<input>*** tag. This will clear the input field when the form is submitted since the last thing that happens in ***handleRobotAdd*** is setting the state of ***input*** back to an empty string. Here's what your code should look like:

```
// App.js return statement

return (
    <div className='app'>
        <Header/>
        <main>
            <Form handleInput={handleInput} handleRobotAdd={handleRobotAdd} input=
{input}/>
            <RobotList />
        </main>
    </div>
)
```

```
// Form.js return statement

return (
    <form onSubmit={props.handleRobotAdd}>
        <p>What is your new robot's name?</p>
        <input
            type='text'
            placeholder='name'
            value={props.input}
            onChange={props.handleInput}/>
        <button>Create Robot</button>
    </form>
)
```

Step Three: Displaying Robots

The ***robots*** array can now have robots added to it from the form! Let's add the ability for the app to display those robots for us.

1. Take a look at ***Robot.js*** in the components folder. On both the ***<h3>*** and ***<img>*** tags, we can see that this component is expecting to have a ***robot*** value passed to it on ***props***. We can also see that ***robot*** has a ***name*** key and a ***picture*** key – the same setup as the objects that are stored in the ***robots*** array. That tells us that this component is meant for displaying information about one robot from our array. Now we just need to get the data there.
2. In ***App.js***, pass ***robots*** as a prop to the ***RobotList*** component. You can do this by typing ***robots={robots}*** in the brackets of the component in the ***return*** statement. Your code will look like this:

```
// App.js return statement

return (
    <div className='app'>
        <Header/>
        <main>
            <Form handleInput={handleInput} handleRobotAdd={handleRobotAdd} input=
{input}/>
            <RobotList robots={robots}/>
        </main>
    </div>
)
```

3. In ***RobotList***, we can see that the function returns different JSX conditionally. However, the condition is incomplete right now. We want to check the length of the array of robots. If it's larger than 0, then we will end up displaying a list of the robots. If it's not, then we'll display the message in the ***else*** block. Go ahead and add the necessary logic to the ***if*** condition in order for this to work. Once you've done that, your code should look something like this:

```
const RobotList = (props) => {
if (props.robots.length > 0) {...
```

4. Now inside the ***<ul>*** element, we are going to write some JavaScript! Inside the curly brackets used to escape the JSX, map over the ***robots*** array from ***props***. In the callback of the ***map*** method, return a ***<p>*** tag containing the name of the robot. This is pretty complicated – take a look at the code below and see if you can figure out what's happening. Then make sure your code matches and try adding a robot!

```
const RobotList = (props) => {
    if (props.robots.length > 0) {
        return (
            <div className='robot-list'>
                <h2>Your Robots</h2>
```

```
                <ul>
                    {
                        props.robots.map((robot) => {
                            return <p>{robot.name}</p>
                        })
                    }
                </ul>
            </div>
        )
    } else {
        return (
            <h2 className='robot-list'>You don't have any robots yet!</h2>
        )
    }
}
```

5. Back in the first step, we looked at the ***Robot*** component. Let's bring that into the ***RobotList*** now and we'll return that instead of the ***<p>*** tag. You also might have noticed a warning about not including a ***key*** prop on the elements in the array. These need to be distinct, so an easy way to do that is to use the index of the element. First, import the ***Robot*** component at the top of the ***RobotList*** file and then change the map callback function to contain both ***robot*** and ***i*** as arguments and return the ***Robot*** component. Add two props on that component: ***key={i}*** and ***robot={robot}***. You code should look like this:

```
// RobotList.js return statement

return (
    <div className='robot-list'>
        <h2>Your Robots</h2>
        <ul >
            {
                props.robots.map((robot, i) => {
                    return <Robot key={i} robot={robot}/>
                })
            }
        </ul>
    </div>
)
```

6. Make sure all your files are saved and look at your app in the browser - everything should be working now!
7. In a personal document, write down some of your thoughts on React.

- What did you like about React? What didn't you like?
- What would pursuing React look like for you?

Congrats!

You've completed the React Spec Trial Day. You should now have a better understanding of what Web Development with React is all about and if it's the specialization that you'd like to pursue. If you have questions, please reach out to a Web Dev Tech Lead.