CHAPTER 3

React Components

In the Hello World example, we created a very basic React native component, using pure JSX. However, in the real world, you will want to do much more than what a simple single-line JSX can do. That is where React components come in. React components can be composed using other components and basic HTML elements; they can respond to user input, change state, interact with other components, and much more.

But, before going into all that detail, let me first describe the application that we will build as part of this book. At every step that we take in this as well as the following chapters, we'll take features of the application or tasks that need to be performed one by one and address them. I like this approach because I've learned things the best when I put them to immediate use. This approach not only lets you appreciate and internalize the concepts because you put them to use, but also brings the more useful and practical concepts to the forefront.

The application I've come up with is something that most developers can relate to.

Issue Tracker

I'm sure that most of you are familiar with GitHub Issues or Jira. These applications help you create a bunch of issues or bugs, assign them to people, and track their statuses. These are essentially CRUD applications (Create, Read, Update, and Delete a record in a database) that manage a list of objects or entities. The CRUD pattern is so useful because pretty much all enterprise applications are built around the CRUD pattern on different entities or objects.

In the case of the Issue Tracker, we'll only deal with a single object or record, because that's good enough to depict the pattern. Once you grasp the fundamentals of how to implement the CRUD pattern in MERN, you'll be able to replicate the pattern and create a real-life application.

Here's the requirement list for the Issue Tracker application, a simplified or toned-down version of GitHub Issues or Jira:

- The user should be able to view a list of issues, with an ability to filter the list by various parameters.
- The user should be able to add new issues, by supplying the initial values of the issue's fields.
- The user should be able to edit and update an issue by changing its field values.
- The user should be able delete an issue.

An issue should have following attributes:

- A title that summarizes the issue (freeform long text)
- An owner to whom the issue is assigned (freeform short text)

V. Subramanian, *Pro MERN Stack*, https://doi.org/10.1007/978-1-4842-4391-6_3

- A status indicator (a list of possible status values)
- Creation date (a date, automatically assigned)
- Effort required to address the issue (number of days, a number)
- Estimated completion date or due date (a date, optional)

Note that I've included different types of fields (lists, date, number, text) to make sure you learn how to deal with different data types. We'll start simple, build one feature at a time, and learn about the MERN stack as we go along.

In this chapter, we'll create React classes and instantiate components. We'll also create bigger components by putting together smaller components. Finally, we'll pass data among these components and create components dynamically from data. In terms of features, the objective in this chapter is to lay out the main page of the Issue Tracker: a list of issues. We'll hard-code the data that is used to display the page and leave retrieval of the data from the server to a later chapter.

React Classes

In this section, the objective is to convert the single-line JSX into a simple React component instantiated from a React class, so that we can later use the full power of the first class React components.

React classes are used to create real components (as opposed to templated HTML where we created a Hello World message based on a variable, which we created in the previous chapter). These classes can then be reused within other components, handle events, and so much more. To start with, let's replace the Hello World example with a simple class forming the starting point for the Issue Tracker application.

React classes are created by extending `React.Component`, the base class from which all custom classes must be derived. Within the class definition, at the minimum, a `render()` method is needed. This method is what React calls when it needs to display the component in the UI.

There are other methods with special meaning to React that can be implemented, called the Lifecycle methods. These provide hooks into various stages of the component formation and other events. We'll discuss other Lifecycle functions in later chapters. But `render()` is one that *must* be present, otherwise the component will have no screen presence. The `render()` function is supposed to return an element (which can be either a native HTML element such as a `<div>` or an instance of another React component).

Let's change the Hello World example from a simple element to use a React class called `HelloWorld`, extended from `React.Component`:

```
...
class HelloWorld extends React.Component {
  ...
}
...
```

■ **Note** We used the ES2015 `class` keyword and the `extends` keyword to define a JavaScript class. React recommends the use of ES2015 classes. If you are not familiar with JavaScript classes, read up and learn about classes starting at https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Classes.

Now, within this class, a `render()` method is needed, which should return an element. We'll use the same JSX `<div>` with the message as the returned element.

```
...
  render() {
    return (
      <div title="Outer div">
        <h1>{message}</h1>
      </div>
    );
...
```

Let's also move all the code for message construction to within the `render()` function so that it remains encapsulated within the scope where it is needed rather than polluting the global namespace.

```
...
  render() {
    const continents = ['Africa','America','Asia','Australia','Europe'];
    const helloContinents = Array.from(continents, c => `Hello ${c}!`);
    const message = helloContinents.join(' ');

    return (
      ...
    );
...
```

In essence, the JSX element is now returned from the `render()` method of the component class called Hello World. The brackets around the JSX representation of the Hello World element are not necessary, but it is a convention that is normally used to make the code more readable, especially when the JSX spans multiple lines.

Just as an instance of a `div` element was created using JSX of the form `<div></div>`, an instance of the `HelloWorld` class can be created like this:

```
...
const element = <HelloWorld />;
...
```

Now, this element can be used in place of the `<div>` element to render inside the node called `contents`, as before. It's worth noting here that `div` and `h1` are built-in internal React components or elements that could be directly instantiated. Whereas `HelloWorld` is something that *we* defined and later instantiated. And within `HelloWorld`, we used React's built-in `div` component. The new, changed `App.jsx` is shown in Listing 3-1.

In the future, I may use component and component class interchangeably, like sometimes we tend to do with classes and objects. But it should be obvious by now that `HelloWorld` and `div` are actually React component classes, whereas `<HelloWorld />` and `<div />` are tangible components or instances of the component class. Needless to say, there is only one `HelloWorld` class, but many `HelloWorld` components can be instantiated based on this class.

Listing 3-1. App.jsx: A Simple React Class and Instance

```
class HelloWorld extends React.Component {
  render() {
    const continents = ['Africa','America','Asia','Australia','Europe'];
    const helloContinents = Array.from(continents, c => `Hello ${c}!`);
    const message = helloContinents.join(' ');

    return (
      <div title="Outer div">
        <h1>{message}</h1>
      </div>
    );
  }
}

const element = <HelloWorld />;

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

By now you should be running `npm run watch` in a console and have started the server using `npm start` in a separate console. Thus, any changes to `App.jsx` should have been automatically compiled. So, if you refresh your browser, you should see the greeting with all the continents, just as before.

EXERCISE: REACT CLASSES

1. In the `render` function, instead of returning one `<div>`, try returning two `<div>` elements placed one after the other. What happens? Why, and what's the solution? Ensure you look at the console running `npm run watch`.
2. Create a runtime error for the React library by changing the string `'contents'` to `'main'` or some other string that doesn't identify an element in the HTML. Where can one see the error? What about JavaScript runtime errors like undefined variable references?

Answers are available at the end of the chapter.

Composing Components

In the previous section, you saw how to build a component by putting together built-in React components that are HTML element equivalents. It's possible to build a component that uses other user-defined components as well, and that's what we will explore in this section.

Component composition is one of the most powerful features of React. This way, the UI can be split into smaller independent pieces so that each piece can be coded and reasoned in isolation, making it easier to build and understand a complex UI. Using components rather than building the UI in a monolithic fashion also encourages reuse. We'll see in a later chapter how one of the components that we built could easily be reused, even though we hadn't thought of reuse at the time of building the component.

A component takes inputs (called properties) and its output is the rendered UI of the component. In this section, we will not use inputs, but put together fine-grained components to build a larger UI. Here are a few things to remember when composing components:

- Larger components should be split into fine-grained components when there is a logical separation possible between the fine-grained components. In this section we'll create logically separated components.
- When there is an opportunity for reuse, components can be built which take in different inputs from different callers. When we build specialized widgets for user inputs in Chapter 10, we will be creating reusable components.
- React's philosophy prefers component composition in preference to inheritance. For example, a specialization of an existing component can be done by passing properties to the generic component rather than inheriting from it. You can read more about this at https://reactjs.org/docs/composition-vs-inheritance.html.
- In general, remember to keep coupling between components to a minimum (coupling is where one component needs to *know* about the details of another component, including parameters or properties passed between them).

Let's design the main page of the application to show a list of issues, with an ability to filter the issues and create a new issue. Thus, it will have three parts: a filter to select which issues to display, the list of issues, and finally an entry form for adding an issue. We're focusing on composing the components at this point in time, so we'll only use placeholders for these three parts. The structure and hierarchy of the user interface is shown in Figure 3-1.

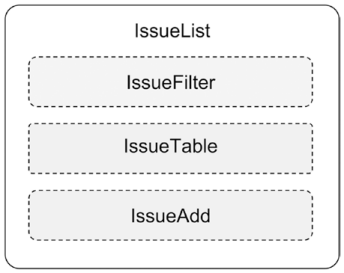

Figure 3-1. *Structure of the Issue List page*

Let's define three placeholder classes—`IssueFilter`, `IssueTable`, and `IssueAdd`—with just a placeholder text within a `<div>` in each. The `IssueFilter` component will look like this:

```
...
class IssueFilter extends React.Component {
  render() {
    return (
      <div>This is a placeholder for the issue filter.</div>
    );
  }
}
...
```


The other two classes—`IssueTable` and `IssueAdd`—will be similar, with a different placeholder message each:

```
...
class IssueTable extends React.Component {
    ...
      <div>This is a placeholder for a table of issues.</div>
...
class IssueAdd extends React.Component {
    ...
      <div>This is a placeholder for a form to add an issue.</div>
...
```

To put these together, let's remove the Hello World class and add a class called `IssueList`.

```
...
class IssueList extends React.Component {
}
...
```

Now let's add a `render()` method. Within this method, let's add an instance of each of the new placeholder classes separated by a `<hr>` or horizontal line. As you saw in the exercise of the earlier section, since the return of `render()` has to be a single element, these elements have to be enclosed within a `<div>` or a React `Fragment` component. The `Fragment` component is like an enclosing `<div>` but it has no effect on the DOM.

Let's use a `Fragment` component like this in the `IssueList`'s `render()` method:

```
...
  render() {
    return (
      <React.Fragment>
        <h1>Issue Tracker</h1>
        <IssueFilter />
        <hr />
        <IssueTable />
        <hr />
        <IssueAdd />
      </React.Fragment>
    );
  }
...
```

Finally, instead of instantiating a `HelloWorld` class, let's instantiate the `IssueList` class, which we will place under the `contents` div.

```
...
~~const element = <HelloWorld />;~~
const element = <IssueList />;
...
```


Ideally, each component should be written as an independent file. But at the moment, we only have placeholders, so for the sake of brevity, we'll keep all the classes in the same file. Also, you haven't learned how to put multiple class files together. At a later stage, when the classes are expanded to their actual content and we also have a way of building or referring to once class from another, we'll separate them out.

Listing 3-2 shows the new contents of the `App.jsx` file with all the component classes.

Listing 3-2. App.jsx: Composing Components

```
class IssueFilter extends React.Component {
  render() {
    return (
      <div>This is a placeholder for the issue filter.</div>
    );
  }
}

class IssueTable extends React.Component {
  render() {
    return (
      <div>This is a placeholder for a table of issues.</div>
    );
  }
}

class IssueAdd extends React.Component {
  render() {
    return (
      <div>This is a placeholder for a form to add an issue.</div>
    );
  }
}

class IssueList extends React.Component {
  render() {
    return (
      <React.Fragment>
        <h1>Issue Tracker</h1>
        <IssueFilter />
        <hr />
        <IssueTable />
        <hr />
        <IssueAdd />
      </React.Fragment>
    );
  }
}

const element = <IssueList />;

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


The effect of this code will be an uninteresting page, as shown in Figure 3-2.

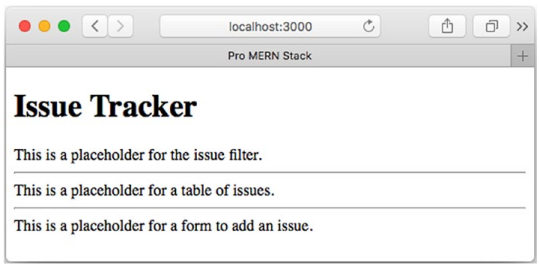

Figure 3-2. *Issue Tracker by composing components*

EXERCISE: COMPOSING COMPONENTS

1. Inspect the DOM in the developer console. Do you see any HTML element corresponding to the `React.Fragment` component? How do you think this can be useful as compared to using a `<div>` element to enclose the various elements?

Answers are available at the end of the chapter.

Passing Data Using Properties

Composing components without any variables is not so interesting. It should be possible to pass different input data from a parent component to a child component and make it render differently on different instances. In the Issue Tracker application, one such component that can be instantiated with different inputs is a table-row showing an individual issue. Depending on the inputs (an issue), the row can display different data. The new structure of the UI is shown in Figure 3-3.

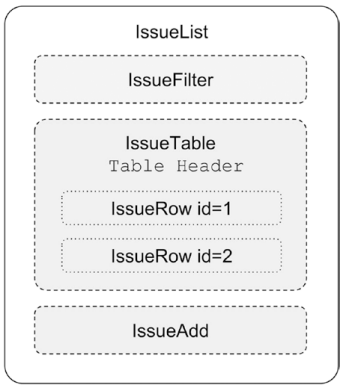

Figure 3-3. *Issue List UI hierarchy with issue rows*

So, let's create a component called `IssueRow`, and then use this multiple times within `IssueTable`, passing in different data to show different issues, like this:

```
...
class IssueTable extends React.Component {
  render() {
    return (
      <table>
        <thead>
          <tr>
            <th>ID</th>
            <th>Title</th>
          </tr>
        </thead>
        <tbody>
          <IssueRow /> {/* somehow pass Issue 1 data to this */}
          <IssueRow /> {/* somehow pass Issue 2 data to this */}
        </tbody>
      </table>
    );
  }
}
...
```


■ **Note** JSX does not support comments natively. In order to add comments, a JavaScript snippet has to be added that has JavaScript style comments. Thus, the form `{/* ... */}` can be used to place comments within JSX. Using HTML style comments like `<!-- ... -->` will not work.

In fact, the way to switch to the JavaScript world within any JSX snippet is to use curly braces. In the previous chapter, we used this to display the Hello World message, which was a JavaScript variable called `message` using the syntax `{message}`.

The easiest way to pass data to child components is using an attribute when instantiating a component. We used the `title` attribute in the previous chapter, but that was an attribute that affected the DOM element in the end. Any custom attribute can also be passed in a similar manner like this from `IssueTable`:

```
...
    <IssueRow issue_title="Title of the first issue" />
...
```

We used the name `issue_title` rather than simply `title` to avoid a confusion between this custom attribute and the HTML `title` attribute. Now, within the `render()` method of the child, the attribute's value can be accessed via a special object variable called `props`, which is available via the `this` accessor. For example, this is how the value of `issue_title` can be displayed within a cell in the `IssueRow` component:

```
...
    <td>{this.props.issue_title}</td>
...
```

In this case, we passed across a simple string. Other data types and even JavaScript objects can be passed this way. Any JavaScript expression can be passed along, by using curly braces ({}) instead of quotes, because the curly braces switches into the JavaScript world.

So, let's pass the issue's title (as a string), its ID (as a number), and the row style (as an object) from `IssueTable` to `IssueRow`. Within the `IssueRow` class, we'll use these passed-in properties to display the ID and title and set the style of the row, by accessing these properties through `this.props`.

The code for the complete `IssueRow` class is shown in Listing 3-3.

Listing 3-3. App.jsx: IssueRow Component, Accessing Passed-in Properties

```
class IssueRow extends React.Component {
  render() {
    const style = this.props.rowStyle;
    return (
      <tr>
        <td style={style}>{this.props.issue_id}</td>
        <td style={style}>{this.props.issue_title}</td>
      </tr>
    );
  }
}
```

We used the attribute `style` for the table-cell just as we would have used it in regular HTML. But note that this is not really an HTML attribute. Instead, it is a *property* being passed on to the built-in React component `<td>`. It's just that the `style` property in the `td` component is interpreted and set as the HTML

`style` attribute. In most cases, like `style`, the name of the attribute is the same as the HTML attribute, but for a few attributes causing conflict with JavaScript reserved words, naming requirements are different. Thus, the `class` HTML attribute needs to be `className` in JSX. Also, hyphens in the HTML attributes need to be replaced by camel cased names, for example, `max-length` becomes `maxLength` in JSX.

A complete list of DOM elements and how attributes for these need to be specified can be found in the React Documentation at https://reactjs.org/docs/dom-elements.html.

Now that we have an `IssueRow` component receiving the properties, let's pass them from the parent, `IssueTable`. The ID and the title are straightforward, but the style we need to pass on has a special convention of specification in React and JSX.

Rather than a CSS kind of string, React needs it to be specified as an object with a specific convention containing a series of JavaScript key-value pairs. The keys are the same as the CSS style name, except that instead of dashes (like `border-collapse`), they are camel cased (like `borderCollapse`). The values are CSS style values, just as in CSS. There is also a special shorthand for specifying pixel values; you can just use a number (like 4) instead of a string `"4px"`.

Let's give the rows a silver border of one pixel and some padding, say four pixels. The style object that would encapsulate this specification would be as follows:

```
...
    const rowStyle = {border: "1px solid silver", padding: 4};
...
```

This can be passed on to the `IssueRow` component using `rowStyle={rowStyle}` when instantiating it. This, and the other variables, can be passed to `IssueRow` while instantiating it like this:

```
...
<IssueRow rowStyle={rowStyle} issue_id={1}
  issue_title="Error in console when clicking Add" />
...
```

Note that we are not using string-like quotes for the Issue ID since it is a number or for `rowStyle` since it is an object. We instead use the curly braces, which makes it a JavaScript expression.

Now, let's construct the `IssueTable` component, which is essentially a `<table>`, with a header row and two columns (ID and title), and two hard-coded `IssueRow` components. Let's also specify an inline style for the table to indicate a collapsed border and use the same `rowStyle` variable to specify the header row styles, to make it look uniform.

Listing 3-4 shows the modified `IssueTable` component class.

Listing 3-4. App.jsx: IssueTable Passing Data to IssueRow

```
class IssueTable extends React.Component {
  render() {
    const rowStyle = {border: "1px solid silver", padding: 4};
    return (
      <table style={{borderCollapse: "collapse"}}>
        <thead>
          <tr>
            <th style={rowStyle}>ID</th>
            <th style={rowStyle}>Title</th>
          </tr>
        </thead>
```



```
        <tbody>
          <IssueRow rowStyle={rowStyle} issue_id={1}
            issue_title="Error in console when clicking Add" />
          <IssueRow rowStyle={rowStyle} issue_id={2}
            issue_title="Missing bottom border on panel" />
        </tbody>
      </table>
    );
  }
}
```

Figure 3-4 shows the effect of these changes in the code.

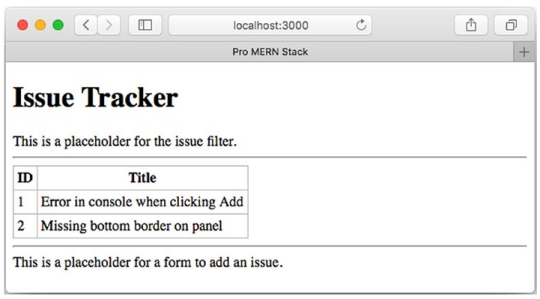

Figure 3-4. *Passing data to child components*

EXERCISE: PASSING DATA USING PROPERTIES

1. Try adding an attribute `border=1` for the table, as we would in regular HTML. What happens? Why? Hint: Read up the section titled "All Supported HTML Attributes" in the "DOM Elements" section of the React API Reference.
2. Why is there a double curly brace in the inline style for the table? Hint: Compare it to the other style, where we declared a variable and used that instead of specifying it inline.
3. The curly braces are a way to escape into JavaScript in the middle of JSX markup. Compare this to similar techniques in other templating languages such as PHP.

Answers are available at the end of the chapter.

Passing Data Using Children

There is another way to pass data to other components, using the contents of the HTML-like node of the component. In the child component, this can be accessed using a special field of `this.props` called `this.props.children`.

Just like in regular HTML, React components can be nested. In the Hello World example, we nested a `<h1>` element inside a `<div>`. When the components are converted to HTML elements, the elements nest in the same order. React components can act like the `<div>` and take in nested elements. In such cases, the JSX expression will need to include both the opening and closing tags, with *child* elements nested in them.

But then, when the parent React component renders, the children are not automatically under it because the structure of the parent React component needs to determine where exactly the children will appear. So, React lets the parent component access the children element using `this.props.children` and lets the parent component determine where it needs to be displayed. This works great when one needs to wrap other components within a parent component. For example, a wrapper `<div>` that adds a border and a padding can be defined like this:

```
...
class BorderWrap extends React.Component {
  render() {
    const borderedStyle = {border: "1px solid silver", padding: 6};
    return (
      <div style={borderedStyle}>
        {this.props.children}
      </div>
    );
  }
}
...
```

Then, during the rendering, *any* component could be wrapped with a padded border like this:

```
...
    <BorderWrap>
      <ExampleComponent />
    </BorderWrap>
...
```

Thus, instead of passing the issue title as a property to `IssueRow`, this technique could be used to embed it as the child contents of `<IssueRow>` like this:

```
...
  <IssueRow issue_id={1}>Error in console when clicking Add</IssueRow>
...
```

Now, within the `render()` method of `IssueRow`, instead of referring to `this.props.issue_title`, it will need to be referred to as `this.props.children`, like this:

```
...
   <td style={borderedStyle}>{this.props.children}</td>
...
```


Let's modify the application to use this method of passing data from `IssueTable` to `IssueRow`. Let's also pass in a nested title element as children, one that is a `<div>` and includes an emphasized piece of text. This change is shown in Listing 3-5.

Listing 3-5. App.jsx: Using Children Instead of Props

```
...
class IssueRow extends React.Component {
...
    return (
      <tr>
        <td style={style}>{this.props.issue_id}</td>
        <td style={style}>{this.props.issue_title}</td>
        <td style={style}>{this.props.children}</td>
      </tr>
    );
...
}
...
...
class IssueTable extends React.Component {
...
        <tbody>
          <IssueRow rowStyle={rowStyle} issue_id={1}
            issue_title="Error in console when clicking Add" />
          <IssueRow rowStyle={rowStyle} issue_id={2}
            issue_title="Missing bottom border on panel" />
          <IssueRow rowStyle={rowStyle} issue_id={1}>
            Error in console when clicking Add
          </IssueRow>
          <IssueRow rowStyle={rowStyle} issue_id={2}>
            <div>Missing <b>bottom</b> border on panel</div>
          </IssueRow>
        </tbody>
...
```

The result of these changes on the output will be minimal, just a little formatting in the title of the second issue will be visible. This is shown in Figure 3-5.

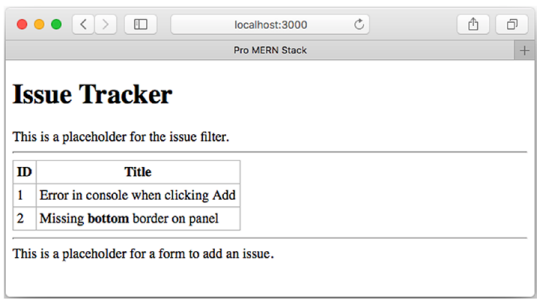

Figure 3-5. *Passing data to child components*

EXERCISE: PASSING DATA USING CHILDREN

1. When is it appropriate to pass data as `props` vis-à-vis `children`? Hint: Think about what is it that we want to pass.

Answers are available at the end of the chapter.

Dynamic Composition

In this section, we'll replace our hard-coded set of `IssueRow` components with a programmatically generated set of components from an array of issues. In later chapters we'll get more sophisticated by fetching the list of issues from a database, but for the moment, we'll use a simple in-memory JavaScript array to store a list of issues.

Let's also expand the scope of the issue from just an ID and a title to include as many fields of an issue as we can. Listing 3-6 shows this in-memory array, declared globally at the beginning of the file `App.jsx`. It has just two issues. The field due is left undefined in the first record, to ensure that we handle the fact that this is an optional field.

Listing 3-6. App.jsx: In-Memory Array of Issues

```
const issues = [
  {
    id: 1, status: New', owner: 'Ravan', effort: 5,
    created: new Date('2018-08-15'), due: undefined,
    title: 'Error in console when clicking Add',
  },
```



```
  {
    id: 2, status: 'Assigned', owner: 'Eddie', effort: 14,
    created: new Date('2018-08-16'), due: new Date('2018-08-30'),
    title: 'Missing bottom border on panel',
  },
];
```

You can add more example issues, but two issues are enough to demonstrate dynamic composition. Now, let's modify the `IssueTable` class to use this array of issues rather than the hard-coded list. Within the `IssueTable` class' `render()` method, let's iterate over the array of issues and generate an array of `IssueRows` from it.

The `map()` method of `Array` comes in handy for doing this, as we can map an issue object to an `IssueRow` instance. Also, instead of passing each field as a property, let's pass the `issue` object itself because there are many fields as part of the object. This is one way to do it, in-place within the table's body:

```
...
    <tbody>
      {issues.map(issue => <IssueRow rowStyle={rowStyle} issue={issue}/>)}
    </tbody>
...
```

If you wanted to use a `for` loop instead of the `map()` method, you can't do that within the JSX, as JSX is not really a templating language. It only can allow JavaScript expressions within the curly braces. We'll have to create a variable in the `render()` method and use that in the JSX. Let's create that variable for the set of issue rows like that anyway for readability:

```
...
  const issueRows = issues.map(issue => <IssueRow rowStyle={rowStyle} issue={issue}/>);
...
```

Now, we can replace the two hard-coded issue components inside `IssueTable` with this variable within the `<tbody>` element like this:

```
...
    <tbody>
      {issueRows}
    </tbody>
...
```

In other frameworks and templating languages, creating multiple elements using a template would have required a special `for` loop construct (e.g., `ng-repeat` in AngularJS) within that templating language. But in React, regular JavaScript can be used for all programmatic constructs. This not only gives you the full power of JavaScript to manipulate templates, but also a lesser number of constructs to learn and remember.

The header row in the `IssueTable` class will now need to have one column for each of the issue fields, so let's do that as well. But by now, specifying the style for each cell is becoming tedious, so let's create a class for the table, name it `table-bordered`, and use CSS to style the table and each table-cell instead. This style will need to be part of `index.html`, and Listing 3-7 shows the changes to that file.

Listing 3-7. index.html: Styles for Table Borders

```
...
  <script src="https://unpkg.com/@babel/polyfill@7/dist/polyfill.min.js"></script>
  <style>
    table.bordered-table th, td {border: 1px solid silver; padding: 4px;}
    table.bordered-table {border-collapse: collapse;}
  </style>
</head>
...
```

Now, we can remove `rowStyle` from all the table-cells and table-headers. One last thing that needs to be done is to identify each instance of `IssueRow` with an attribute called key. The value of this key can be anything, but it has to uniquely identify a row. React needs this key so that it can optimize the calculation of differences when things change, for example, when a new row is inserted. We can use the ID of the issue as the key, as it uniquely identifies the row.

The final `IssueTable` class with a dynamically generated set of `IssueRow` components and the modified header is shown in Listing 3-8.

Listing 3-8. App.jsx: IssueTable Class with IssueRows Dynamically Generated and Modified Header

```
class IssueTable extends React.Component {
  render() {
    const issueRows = issues.map(issue =>
      <IssueRow key={issue.id} issue={issue} />
    );

    return (
      <table className="bordered-table">
        <thead>
          <tr>
            <th>ID</th>
            <th>Status</th>
            <th>Owner</th>
            <th>Created</th>
            <th>Effort</th>
            <th>Due Date</th>
            <th>Title</th>
          </tr>
        </thead>
        <tbody>
          {issueRows}
        </tbody>
      </table>
    );
  }
}
```

The changes in `IssueRow` are quite simple. The inline styles have to be removed, and a few more columns need to be added, one for each of the added fields. Since React does not automatically call `toString()` on objects that are to be displayed, the dates have to be explicitly converted to strings. The `toString()` method results in a long string, so let's use `toDateString()` instead. Since the field due is

optional, we need to also check for its presence before calling `toDateString()` on it. An easy way to do this is to use the ternary `? - :` operator in an expression like this:

```
...
  issue.due ? issue.due.toDateString() : ''
...
```

The ternary operator is handy because it is a JavaScript expression, and it can be used directly in place of the display string. Otherwise, to use an `if-then-else` statement, the code would have to be outside the JSX part, in the beginning of the `render()` method implementation. The new `IssueRow` class is shown in Listing 3-9.

Listing 3-9. App.jsx: New IssueRow Class Using Issue Object Property

```
class IssueRow extends React.Component {
  render() {
    const issue = this.props.issue;
    return (
      <tr>
        <td>{issue.id}</td>
        <td>{issue.status}</td>
        <td>{issue.owner}</td>
        <td>{issue.created.toDateString()}</td>
        <td>{issue.effort}</td>
        <td>{issue.due ? issue.due.toDateString() : ''}</td>
        <td>{issue.title}</td>
      </tr>
    );
  }
}
```

After these changes, the screen should look like Figure 3-6.

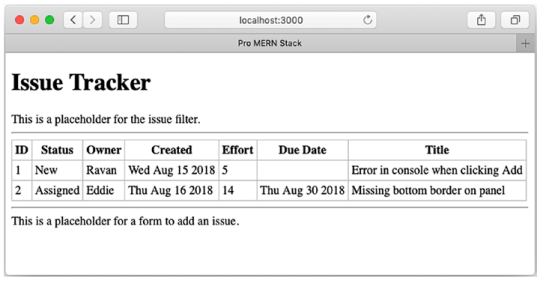

Figure 3-6. *Issue Rows constructed programmatically from an array*

EXERCISE: DYNAMIC COMPOSITION

1. We used the issue's `id` field as the value of key. What other keys could have been used? Which one would you choose?
2. In the previous section, we passed every field of an issue as a separate property to `IssueRow`. In this section, we passed the entire issue object. Why?
3. Instead of using a local variable `issueRows`, try using the map expression directly inside the `<tbody>`. Does it work? What does it tell us?

Answers are available at the end of the chapter.

Summary

In this chapter, we created a barebones version of the main page of the Issue Tracker. We started using React classes instead of simple elements, some of which were just placeholders to depict components that we have not yet developed. We did this by writing fine-grained individual components and putting them together (composing) in an enclosing component. We also passed parameters or data from an enclosing component to its children, and thus reused a component class and rendered it differently with different data, dynamically using a `map()` to generate components based on an array of input data.

The components didn't do much apart from rendering themselves based on the input data. In the next chapter, we'll see how user interaction can affect the data and change the appearance of a component.

Answers to Exercises

Exercise: React Classes

1. Compilation will fail with an error, "Adjacent JSX elements must be wrapped in an enclosing tag". The `render()` method can only have a single return value, thus, it can return only one element. Enclosing the two `<div>`s in another `<div>` is one solution, or as the error message suggests, using a `Fragment` component is another solution, which we will discuss in later sections.
2. React prints an error in the browser's JavaScript console when it is a React error. Regular JavaScript errors are also shown in the console, but the code displayed is not the original code; it is the compiled code. We'll learn how to debug using the original source in later chapters.

Exercise: Composing Components

1. No, there is no enclosing element. All the elements returned by `IssueList` appear directly under the `contents` div. In this case, we could have easily used a `<div>` to enclose the elements.

   But imagine a situation where a list of table-rows needs to be returned, like this:

   ```
   ...
     <tr> {/* contents of row 1 */} </tr>
     <tr> {/* contents of row 2 */} </tr>
   ...
   ```


   Then, the calling component would place these rows under a `<tbody>` element. Adding a `<div>` to enclose these rows would result in an invalid DOM tree, as `<tbody>` cannot have a `<div>` within it. A fragment is the only option in this case.

Exercise: Passing Data Using Properties

1. A border will not be displayed. How React interprets each element's attribute is different from how an HTML parser does it. The border attribute is not one of the supported attributes. React completely ignores the border attribute.
2. The outer braces denote that the attribute value is a JavaScript expression. The inner braces specify an object, which is the attribute's value.
3. The curly braces of React are similar to `<?php ... ?>` of PHP, with a slight difference. The contents within a `<?php ... ?>` tag are full-fledged programs, whereas in JSX, you can only have JavaScript expressions. All programming constructs like `for` loops are written outside the JSX in plain JavaScript.

Exercise: Passing Data Using Children

1. `props` are flexible and useful for passing in any kind of data. On the other hand, `children` can only be an *element*, which can also be deeply nested. Thus, if you have simple data, pass it as `props`. If you have a component to pass, you could use `children` if it is deeply nested and naturally appears within the child component. Components can also be passed as `props`, typically when you want to pass multiple components or when the component is not a natural child content of the parent.

Exercise: Dynamic Composition

1. Another choice for the key property is the array index, as it is also unique. If the key is a large value like a UUID, you may think that it is more efficient to use the array index, but in reality it is not. React uses the key to *identify* the row. If it finds the same key, it assumes it is the same row. If the row has not changed, it does not re-render the row.

   Thus, if you insert a row, React will be more efficient in shuffling existing rows rather than re-rendering the entire table if the rows' keys were the ID of the object. If you used the array index instead, it would think that every row after the inserted row has changed and re-render each of them.
2. Passing the entire object is obviously more concise. I would choose to pass individual properties only if the number of properties being passed is a small subset of the full set of properties of the object.
3. It works, despite the fact that we have JSX within the expression. Anything within the curly braces is parsed as a JavaScript expression. But since we are using a JSX transform on JavaScript expressions, these snippets will also go through the transform. It is possible to nest this deeper and use another set of curly braces within the nested piece of JSX and so on.

CHAPTER 4

React State

Until now, we only saw static components, that is, components that did not change. To make components that respond to user input and other events, React uses a data structure called *state* in the component. In this chapter, we will explore how to use React State and how it can be manipulated to change how the component looks and what it shows on screen.

The state essentially holds the data, something that can change, as opposed to the immutable properties in the form of `props` that you saw earlier. This state needs to be used in the `render()` method to build the view. It is only the change of state that can change the view. When data or the state changes, React automatically rerenders the view to show the new changed data.

For this chapter, the goal is to add a button and append a row to the initial list of issues on click of that button. We'll add this button in place of the placeholder text in the `IssueAdd` component. By doing that, you'll learn about a component's state, how to manipulate it, how to handle events, and how to communicate between components.

We'll start by appending a row without user interaction. We'll do this using a timer rather than a button so that we focus on the state and modifications and not deal with things like user input. Toward the end of the chapter, we will replace the timer with an actual button and a form for user input.

Initial State

The state of a component is captured in a variable called `this.state` in the component's class, which should be an object consisting of one or more key-value pairs, where each key is a state variable name and the value is the current value of that variable. React does not specify what needs to go into the state, but it is useful to store in the state anything that affects the rendered view and can change due to any event. These are typically events generated due to user interaction.

For the `IssueTable` component, the list of issues being displayed is definitely one such piece of data that both affects the rendered view and can also change when an issue is added, edited, or deleted. The array of issues is therefore an ideal state variable.

Other things, such as the size of the window, also can change, but this is not something that affects the DOM. Even though the display changes (for example, a line may wrap because the window is narrower), the change is handled by the browser directly based on the same DOM. So, we don't need to capture this in the state of the component. There may be cases where it does affect the DOM; for example, if the height of the window determines *how many* issues we display, we may store the height of the window in a state variable and restrict the number of `IssueRow` components being constructed. In those cases, the height of the window, or a derived value, for example, the number of issues being shown, could also be stored in the state.

Things that don't change, for example, the border style of the table, also don't need to go into the state. That's because user interaction or other events do not affect the style of borders.

V. Subramanian, *Pro MERN Stack*, https://doi.org/10.1007/978-1-4842-4391-6_4

For now, let's just use an array of issues as the one and only state of the component and use that array to construct the table of issues. Thus, in the `render()` method of `IssueTable`, let's change the loop that creates the set of `IssueRows` to use the state variable called `issues` rather than the global array like this:

```
...
  const issueRows = this.state.issues.map(issue =>
     <IssueRow key={issue.id} issue={issue} />
...
```

As for the initial state, let's use a hard-coded set of issues and set it to the initial state. We already have a global array of issues; let's rename this array to `initialIssues`, just to make it explicit that it is only an initial set.

```
...
const initialIssues = [
  ...
];
...
```

Setting the initial state needs to be done in the constructor of the component. This can be done by simply assigning the variable `this.state` to the set of state variables and their values. Let's use the variable `initialIssues` to initialize the value of the state variable `issues` like this:

```
...
    this.state = { issues: initialIssues };
...
```

Note that we used only one state variable called `issues`. We can have other state variables, for instance if we were showing the issue list in multiple pages, and we wanted to also keep the page number currently being shown as another state variable, we could have done that by adding another key to the object like `page: 0`.

The set of all changes to use the state to render the view of `IssueTable` is shown in Listing 4-1.

Listing 4-1. App.jsx: Initializing and Using State

```
...
~~const issues = [~~
const initialIssues = [
  {
    id: 1, status: 'New', owner: 'Ravan', effort: 5,
    created: new Date('2018-08-15'), due: undefined,
  },
...
class IssueTable extends React.Component {
  constructor() {
    super();
    this.state = { issues: initialIssues };
  }

  render() {
    ~~const issueRows = issues.map(issue =>~~
```



```
    const issueRows = this.state.issues.map(issue =>
      <IssueRow key={issue.id} issue={issue} />
    );
...
```

Running and testing this piece of code should show no change in the application; you will still see a table containing two rows of issues, just as before.

EXERCISE: INITIAL STATE

1. If you needed to display each row in a different background color based on the status of the issue, how would you go about it? Would you have a list of colors corresponding to each issue also stored in the state? Why or why not?

Answers are available at the end of the chapter.

Async State Initialization

Although we set the initial state in the constructor, it is highly unlikely that regular SPA components will have the initial state available to them statically. These will typically be fetched from the server. In the case of the Issue Tracker application, even the initial list issues to be displayed would have to be fetched via an API call.

The state can only be assigned a value in the constructor. After that, the state can be modified, but only via a call to `React.Component`'s `this.setState()` method. This method takes in one argument, which is an object containing all the changed state variables and their values. The only state variable that we have is the one called `issues`, which can be set to any list of issues in a call to `this.setState()` like this:

```
...
  this.setState({ issues: newIssues });
...
```

If there were additional state variables, setting only one variable (`issues`) will cause it to get *merged* with the existing state. For example, if we had stored the current page as another state variable, the new value of the state variable `issues` will be merged into the state, keeping the value of the current page unchanged.

Since at the time of constructing the component, we don't have the initial data, we will have to assign an empty array to the `issues` state variable in the constructor.

```
...
  constructor() {
    this.state = { issues: [] };
...
```

We will not fetch the data from the server just yet, but to explore the changes to the state initialization, let's simulate such a call. The key difference between a global array of issues and a call to the server is that the latter needs an asynchronous call. Let's add a method to the `IssueTable` class that asynchronously

returns an array of issues. Eventually, we'll replace this with an API call to the server, but for the moment, we'll use a `setTimeout()` call to make it asynchronous. In the callback to the `setTimeout()` call (which will eventually be an Ajax call), let's call `this.setState()` with the static array of initial issues like this:

```
...
  loadData() {
    setTimeout(() => {
      this.setState({ issues: initialIssues });
    }, 500);
  }
...
```

The timeout value of 500 milliseconds is somewhat arbitrary: it's reasonable to expect a real API call to fetch the initial list of issues within this time.

Now, it is very tempting to call `loadData()` within the constructor of `IssueTable`. It may even seem to work, but the fact is that the constructor only constructs the *component* (i.e., does all the initialization of the object in memory) and does not render the UI. The rendering happens later, when the component needs to be shown on the screen. If `this.setState()` gets called before the component is ready to be rendered, things will go awry. You may not see this happening in simple pages, but if the initial page is complex and takes time to render, and if the Ajax call returns before rendering is finished, you will get an error.

React provides many other methods called *lifecycle methods* to cater to this and other situations where something needs to be done depending on the stage, or changes in the status of the component. Apart from the constructor and the `render()` methods, the following lifecycle methods of a component could be of interest:

- `componentDidMount()`: This method is called as soon as the component's representation has been converted and inserted into the DOM. A `setState()` can be called within this method.
- `componentDidUpdate()`: This method is invoked immediately after an update occurs, but it is not called for the initial render. `this.setState()` can be called within this method. The method is also supplied the previous props and previous state as arguments, so that the function has a chance to check the differences between the previous props and state and the current props and state before taking an action.
- `componentWillUnmount()`: This method is useful for cleanup such as cancelling timers and pending network requests.
- `shouldComponentUpdate()`: This method can be used to optimize and prevent a rerender in case there is a change in the props or state that really doesn't affect the output or the view. This method is rarely used because, when the state and props are designed well, there will rarely be a case when the state or props change but an update is not required for the view.

The best place to initiate the loading of data in this case is the `componentDidMount()` method. At this point in time, the DOM is guaranteed to be ready, and `setState()` can be called to rerender the component. `componentDidUpdate()` is an option as well, but since it may not be called for the initial render, let's not use it. Let's add the `componentDidMount()` method in `IssueTable` and load the data within this method:

```
...
  componentDidMount() {
    this.loadData();
  }
...
```

The complete set of changes in the `IssueTable` class is shown in Listing 4-2.

Listing 4-2. App.jsx, IssueTable: Loading State Asynchronously

```
...
class IssueTable extends React.Component {
  constructor() {
    super();
    this.state = { issues: initialIssues };
    this.state = { issues: [] };
  }

  componentDidMount() {
    this.loadData();
  }

  loadData() {
    setTimeout(() => {
      this.setState({ issues: initialIssues });
    }, 500);
  }
...
```

If you refresh the browser (assuming you're still running `npm run watch` and `npm start` on two different consoles), you will find that the list of issues is displayed as it used to be in the previous steps. But, you will also see that for a fraction of a second after the page is loaded, the table is empty, as shown in Figure 4-1.

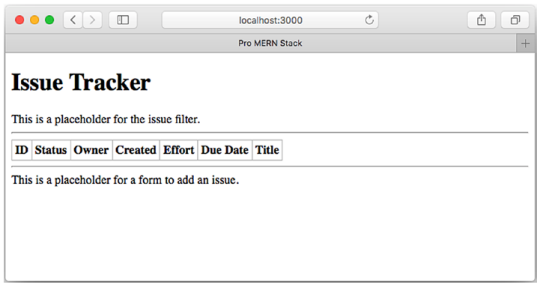

Figure 4-1. *Empty table shown for a fraction of a second*

It gets filled soon after, but still, there is a flicker. When we explore server-side rendering in later chapters, we will get rid of this ungainly flicker. For the moment, let's live with this minor UI unpleasantness.

Updating State

In the previous sections, you saw how to set the initial state, using a direct assignment in the constructor as well as setting a value in other lifecycle methods using `this.setState()`. In this section, let's make a minor change to the state rather than set a completely new value to it. Let's add a new issue and thus change, not the complete state, but only a portion of it.

To start, let's add a method in `IssueTable` to add a new issue. This can take in as an argument an issue object, to which we'll assign a new ID and set the creation date. The new ID can be calculated from the existing length of the array.

```
...
  createIssue(issue) {
    issue.id = this.state.issues.length + 1;
    issue.created = new Date();
  }
...
```

Note that the state variable cannot be set directly, nor can it be mutated directly. That is, setting `this.state.issues` to a new value or modifying its elements is not allowed. The variable `this.state` in the component should always be treated as immutable. For example, the following should not be done:

```
...
    this.state.issues.push(issue);    // incorrect!
...
```

The reason is that React does not automatically identify such changes to the state because it is a plain JavaScript variable. The only way to let React know something has changed, and to cause a rerender, is to call `this.setState()`. Further, `this.setState()` may cause the changes that are done directly to the state variable to be overwritten. So, the following should not be done either:

```
...
    issues = this.state.issues;
    issues.push(issue);         // same as this.state.issues.push()!
    this.setState({ issues: issues });
...
```

It may seem to work, but it will have unexpected consequences in some of the lifecycle methods within this as well as descendent components. Especially in those methods that compare the old and new properties, the difference between the old state and the new one will not be detected.

What is needed in the `setState()` call is a fresh array of issues, say a copy of the state variable. If any existing array element, say an issue itself, is changing, not only is the copy of the array needed, but also the copy of the object that is being changed is needed. There are libraries called *immutability helpers,* such as `immutable.js` (`http://facebook.github.io/immutable-js/`), which can be used to construct the new state object. When a property of the object is modified, the library creates a copy optimally.

But we will only append an issue, and not change an existing issue. It's fairly straightforward to make a shallow copy of the array, and this will suffice for the moment. So, we won't be using the library—there isn't much extra code we need to write to handle it. If, in your application, you find that you have to make lots of copies because of deep nesting of objects in the state, you could consider using `immutable.js`.

The simple way to make a copy of an array is using the `slice()` method. So let's create a copy of the `issues` array like this:

```
...
    issues = this.state.issues.slice();
...
```

Later in the chapter, we will create a user interface to add a new issue. But for now, rather than dealing with the complexity of UI and event handling, let's just add a timer, on the expiry of which, a hard-coded sample issue will be appended to the list of issues. Let's first declare this hard-coded sample issue object globally, right after the global `initialIssues`:

```
...
const sampleIssue = {
  status: 'New', owner: 'Pieta',
  title: 'Completion date should be optional',
};
...
```

Let's use this object in a call to `createIssue()`, after the expiry of a timer of two seconds, in the constructor of `IssueTable`:

```
...
    setTimeout(() => {
      this.createIssue(sampleIssue);
    }, 2000);
...
```

This should automatically add the sample issue to the list of issues after the page is loaded. The final set of changes—for using a timer to append a sample issue to the list of issues—is shown in Listing 4-3.

Listing 4-3. App.jsx: Appending an Issue on a Timer

```
...
const initialIssues = [
  ...
];

const sampleIssue = {
  status: 'New', owner: 'Pieta',
  title: 'Completion date should be optional',
};

...

class IssueTable extends React.Component {
  constructor() {
    super();
    this.state = { issues: [] };
    setTimeout(() => {
      this.createIssue(sampleIssue);
    }, 2000);
  }
```



```
  ...

  createIssue(issue) {
    issue.id = this.state.issues.length + 1;
    issue.created = new Date();
    const newIssueList = this.state.issues.slice();
    newIssueList.push(issue);
    this.setState({ issues: newIssueList });
  }
}
...
```

On running this set of changes and refreshing the browser, you'll see that there are two rows of issues to start with. After two seconds, a third row is added with a newly generated ID and the contents of the sample issue. A screenshot of the three-row table is shown in Figure 4-2.

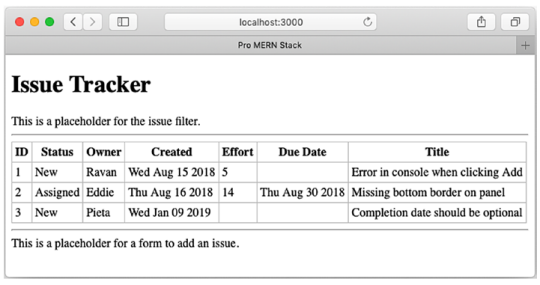

Figure 4-2. *Appended row to initial set of issues*

Note that we did not explicitly call a `setState()` on the `IssueRow` components. React automatically propagates any changes to child components that depend on the parent component's state. Further, we did not have to write any code for inserting a row into the DOM. React calculated the changes to the virtual DOM and inserted a new row.

At this point, the hierarchy of the components and the data flow can be visually depicted, as shown in Figure 4-3.

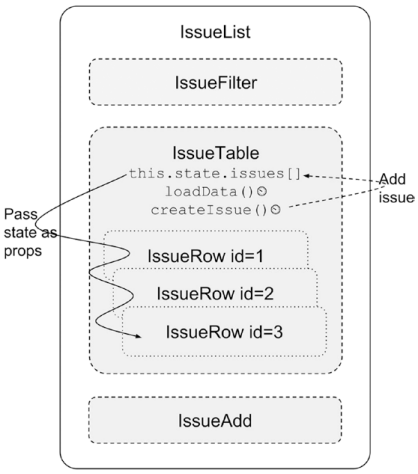

Figure 4-3. *Setting state and passing data as props*

EXERCISE: UPDATING STATE

1. Set up another timer at say, three seconds, right after the first timer to add yet another issue based on `sampleIssue`. Do you notice something going wrong when the second new issue is added? Hint: Look at the ID of the first new issue. Why do you think this is happening? How can you correct it?
2. Add a `console.log` in the `IssueRow`'s `render()` method. How many times do you expect `render()` to be called? How many console logs do you see? (Make sure you undo the changes you did in the previous exercise!)

Answers are available at the end of the chapter.

Lifting State Up

Before we add user interface elements to create new issues, let's move the initiation of the creation to where it really belongs: in the `IssueAdd` component. This will allow us to deal with the changes one step at a time, because moving the timer for adding a new issue from the `IssueTable` component to the `IssueAdd` component is not as trivial as it first appears.

If you do try to move it, you will immediately realize that the `createIssue()` method will also have to move, or we need to have a variant within `IssueAdd` that can communicate back to `IssueTable` and call the `createIssue()` method, which continues to remain there. But there is no straightforward way to communicate between siblings in React. Only parents can pass information down to children; horizontal communication seems hard, if not impossible.

The way around this is to have the *common parent* contain the state and all the methods that deal with this state. By lifting the state up on level to `IssueList`, information can be propagated down to `IssueAdd` as well as to `IssueTable`.

Let's start by moving the state to `IssueList` and the methods to load the initial state. The constructor of `IssueTable` had both the state initialization as well as the timer, of which only the state initialization needs to move (the timer will move to `IssueAdd`):

```
...
class IssueList extends React.Component {
  constructor() {
    super();
    this.state = { issues: [] };
  }
...
```

The other methods that deal with the state are `componentDidMount()`, `loadData()`, and `createIssue()`. Let's move these also to the `IssueList` class:

```
...
class IssueList extends React.Component {
  ...
  componentDidMount() {
    ...
  }
  loadData() {
    ...
  }
  createIssue(issue) {
    ...
  }
  ...
}
```

Now, `IssueTable` doesn't have a state to construct the `IssueRow` components from. But you have already seen how data can be passed in from a parent to a child in the form of `props`. Let's use that strategy and pass the array of issues from the state within `IssueList` to `IssueTable` via props:

```
...
        <IssueTable issues={this.state.issues} />
...
```

And, within `IssueTable`, instead of referring to the state variable `issues`, we'll need to get the same data from props:

```
...
    const issueRows = this.state.issues.map(issue =>
    const issueRows = this.props.issues.map(issue =>
...
```


As for `IssueAdd`, we need to move the timer into the constructor of this class and trigger the addition of a new issue from within this component. But we don't have the `createIssue()` method available here. Fortunately, since a parent component can pass information down to a child component, we'll pass the *method* itself as part of the props to `IssueAdd` from `IssueList`, so that it can be called from `IssueAdd`. Here's the changed instantiation of the `IssueAdd` component within `IssueList`:

```
...
        <IssueAdd createIssue={this.createIssue} />
...
```

This lets us make a call to `createIssue()` from `IssueAdd` using `this.props.createIssue()` as part of the timer callback. So let's create a constructor in `IssueAdd` and move the timer set up with a minor change to use the `createIssue` callback passed in via props like this:

```
...
    setTimeout(() => {
      this.props.createIssue(sampleIssue);
    }, 2000);
...
```

We still have to take care of one more thing before we can say we are done with this set of changes. All this while, we have been using the arrow function syntax to set up timers. In ES2015, the arrow function has the effect of setting the context (the value of `this`) to the *lexical scope*. This means that `this` within the callback will refer to whatever `this` was in the lexical scope, that is, outside of the anonymous function, where the code is present.

That worked as long as the called function was within the same class as the timer callback. It still works, in the `loadData()` method, because `this` refers to the `IssueList` component where the timer was fired, and therefore, `this.state` refers to the state within `IssueList` itself.

But, when `createIssue` is called from a timer within `IssueAdd`, `this` will refer to the `IssueAdd` component. But what we really want is for `createIssue` to be always called with `this` referring to the `IssueList` component. Otherwise, `this.state.issues` will be undefined.

The way to make this work is to *bind* the method to the `IssueList` component before passing it around. We could make this change when instantiating `IssueAdd` like this:

```
...
        <IssueAdd createIssue={this.createIssue.bind(this)} />
...
```

But then, if we need to ever refer to the same method again and pass it to some other child component, we'd have to repeat this code. Also, there is never going to be a case where we will need the method to be not bound, so it is best to replace the definition of `createIssue` with a bound version of itself. The recommended way to do this is in the constructor of the class where this method is implemented.

So, instead of binding during the instantiation of `IssueAdd`, let's bind it in the constructor of `IssueList`.

```
...
    this.createIssue = this.createIssue.bind(this);
...
```

The new versions of each of these classes, after making all these changes, are shown in the following listings. Listing 4-4 shows the new `IssueTable` class; Listing 4-5 shows the new `IssueAdd` class; and Listing 4-6 shows the new `IssueList` class.

Listing 4-4. App.jsx: New IssueTable Class

```
class IssueTable extends React.Component {
  render() {
    const issueRows = this.props.issues.map(issue =>
      <IssueRow key={issue.id} issue={issue} />
    );

    return (
      <table className="bordered-table">
        <thead>
          <tr>
            <th>ID</th>
            <th>Status</th>
            <th>Owner</th>
            <th>Created</th>
            <th>Effort</th>
            <th>Due Date</th>
            <th>Title</th>
          </tr>
        </thead>
        <tbody>
          {issueRows}
        </tbody>
      </table>
    );
  }
}
```

Listing 4-5. App.jsx, IssueAdd: New IssueAdd Class

```
class IssueAdd extends React.Component {
  constructor() {
    super();
    setTimeout(() => {
      this.props.createIssue(sampleIssue);
    }, 2000);
  }
  render() {
    return (
      <div>This is a placeholder for a form to add an issue.</div>
    );
  }
}
```

Listing 4-6. App.jsx, IssueList: New IssueList Class

```
class IssueList extends React.Component {
  constructor() {
    super();
    this.state = { issues: [] };
```

```
    this.createIssue = this.createIssue.bind(this);
  }

  componentDidMount() {
    this.loadData();
  }

  loadData() {
    setTimeout(() => {
      this.setState({ issues: initialIssues });
    }, 500);
  }

  createIssue(issue) {
    issue.id = this.state.issues.length + 1;
    issue.created = new Date();
    const newIssueList = this.state.issues.slice();
    newIssueList.push(issue);
    this.setState({ issues: newIssueList });
  }

  render() {
    return (
      <React.Fragment>
        <h1>Issue Tracker</h1>
        <IssueFilter />
        <hr />
        <IssueTable issues={this.state.issues} />
        <hr />
        <IssueAdd createIssue={this.createIssue} />
      </React.Fragment>
    );
  }
}
```

The effect of these changes will not be seen in the user interface. The application will behave as it used to. On refreshing the browser, you will see an empty table to start with, which will soon be populated with two issues and after two seconds, another issue will be added.

But this sets us up nicely for the change where we can replace the timer in `IssueAdd` with a button that the user can click to add a new issue.

EXERCISE: LIFTING STATE UP

1. Remove the binding of the method `createIssue()`. What error do you see in the console? What does it tell you?

Answers are available at the end of the chapter.

Event Handling

Let's now add an issue interactively, on the click of a button rather than use a timer to do this. We'll create a form with two text inputs and use the values that the user enters in them to add a new issue. An Add button will trigger the addition.

Let's start by creating the form with two text inputs in the `render()` method of `IssueAdd` in place of the placeholder `div`.

```
...
      <div>This is a placeholder for a form to add an issue.</div>
      <form>
        <input type="text" name="owner" placeholder="Owner" />
        <input type="text" name="title" placeholder="Title" />
        <button>Add</button>
      </form>
...
```

At this point, we can remove the timer that creates an issue from the constructor.

```
...
  constructor() {
    super();
    setTimeout(() => {
      this.props.createIssue(sampleIssue);
    }, 2000);
  }
...
```

If you run the code, you'll see a form being displayed in place of the placeholder in `IssueAdd`. The screenshot of how this looks is shown in Figure 4-4.

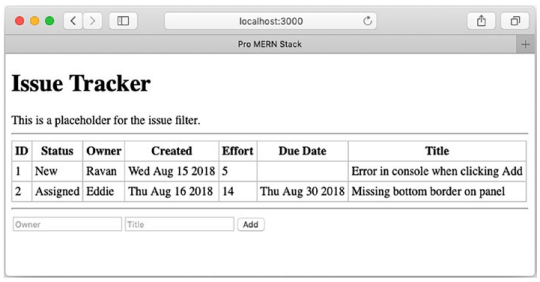

Figure 4-4. *IssueAdd placeholder replaced with a form*

At this point, clicking Add will submit the form and fetch the same screen again. That's not what we want. Firstly, we want it to call `createIssue()` using the values in the owner and title fields. Secondly, we want to prevent the form from being submitted because we will handle the event ourselves.

To handle events such as `onclick` and `onsubmit`, the properties that we need to supply to the elements are, simply, `onClick` and `onSubmit`. As in plain HTML and JavaScript, these properties take functions as values. We'll create a class method called `handleSubmit()` to receive the submit event from the form when the Add button is clicked. Within this method, we'll need a handle to the form, so as in regular HTML, let's give the form a name, say, `issueAdd` which can then be referred to in JavaScript using `document.forms.issueAdd`.

So, let's rewrite the form declaration with a name and an `onSubmit` handler like this.

```
...
          <form name="issueAdd" onSubmit={this.handleSubmit}>
...
```

Now, we can implement the method `handleSubmit()` in `IssueAdd`. This method receives the event that triggered the submit as an argument. In order to prevent the form from being submitted when the Add button is clicked, we need to call the `preventDefault()` function on the event. Then, using the form handle via `documents.forms.issueAdd`, we can get the values of the text input fields. Using these, we'll create a new issue by calling `createIssue()`. After the call to `createIssue()`, let's keep the form ready for the next set of inputs by clearing the text input fields.

```
...
  handleSubmit(e) {
    e.preventDefault();
    const form = document.forms.issueAdd;
    const issue = {
      owner: form.owner.value, title: form.title.value, status: 'New',
    }
    this.props.createIssue(issue);
    form.owner.value = ""; form.title.value = "";
  }
...
```

■ **Note** At this point, we are using the conventional way of taking user input, using named inputs and getting their value using the `value` property of the DOM element. React has another way of dealing with user input by way of *controlled* components, where the value of the input is tied to a state variable. We'll explore this in later chapters.

Since `handleSubmit` will be called from an event, the context, or `this` will be set to the object generating the event, which is typically the `window` object. As you saw in the previous section, to let this method have access to the object variables via `this`, we need to bind it to `this` in the constructor:

```
...
  constructor() {
    super();
    this.handleSubmit = this.handleSubmit.bind(this);
  }
...
```


The new full code of the `IssueAdd` class, after these changes, is shown in Listing 4-7.

Listing 4-7. App.jsx, IssueList: New IssueAdd Class

```
class IssueAdd extends React.Component {
  constructor() {
    super();
    this.handleSubmit = this.handleSubmit.bind(this);
  }

  handleSubmit(e) {
    e.preventDefault();
    const form = document.forms.issueAdd;
    const issue = {
      owner: form.owner.value, title: form.title.value, status: 'New',
    }
    this.props.createIssue(issue);
    form.owner.value = ""; form.title.value = "";
  }

  render() {
    return (
      <form name="issueAdd" onSubmit={this.handleSubmit}>
        <input type="text" name="owner" placeholder="Owner" />
        <input type="text" name="title" placeholder="Title" />
        <button>Add</button>
      </form>
    );
  }
}
```

The global object `sampleIssue` is no longer required, so we can get rid of it. This change is shown in Listing 4-8.

Listing 4-8. App.jsx, Removal of sampleIssue

```
...
const sampleIssue = {
  status: 'New', owner: 'Pieta',
  title: 'Completion date should be optional',
};
...
```

You can now test the changes by entering some values in the owner and title fields and clicking Add. You can add as many rows as you like. If you add two issues, you'll get a screen like the one in Figure 4-5.

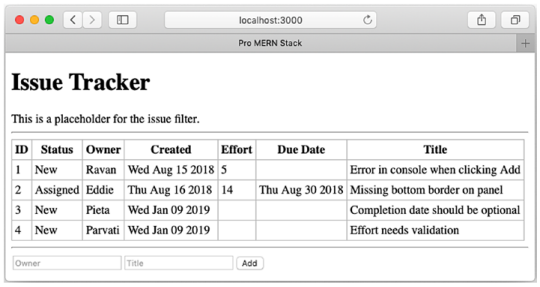

Figure 4-5. *Adding new issues using the IssueAdd form*

At the end of all this, we have been able to encapsulate and initiate the creation of a new issue from the `IssueAdd` component itself. To do this, we "lifted the state up" to the least common ancestor, so that all children have access to it directly via passed-in props or via callbacks that can modify the state. This new UI hierarchy data and function flow is depicted in Figure 4-6. Compare this with the situation where the state was maintained in `IssueTable`, as in Figure 4-3.

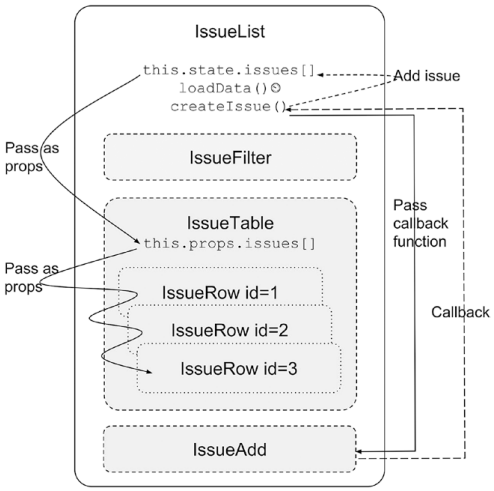

Figure 4-6. *Component hierarchy and data flow after lifting state up*

EXERCISE: EVENT HANDLING

1. Refresh the browser; you'll see that the added issues are gone. How does one persist the changes?
2. Remove `e.preventDefault()`. Click the Add button with some values for owner and title. What happens? What do you see in the URL bar? Can you explain this?
3. Use the developer console to inspect the table and add a breakpoint on the `<tbody>` element as "break on subtree modification". Now, add a new issue. How many times is the subtree being modified? Compare this with exercise #2 in "Updating State," where you traced the number of `render()` calls in a `IssueRow`.

Answers are available at the end of the chapter.

Stateless Components

We have three functioning React components (`IssueAdd`, `IssueRow` and `IssueTable`) composed hierarchically into `IssueList` (another one, the `IssueFilter`, is still a placeholder). But there is a difference among these functioning component classes.

`IssueList` has lots of methods, a state, initialization of the state, and functions that modify the state. In comparison, `IssueAdd` has some interactivity, but no state[1]. But, if you notice, `IssueRow` and `IssueTable` have nothing but a `render()` method. For performance reasons and for clarity of code, it is recommended that such components are written as functions rather than classes: a function that takes in `props` and just renders based on it. It's as if the component's view is a pure function of its `props`, and it is stateless. The `render()` function itself can be the component.

If a component does not depend on props, it can be written as a simple function whose name is the component name. For example, consider the following Hello World class we wrote in the beginning of Chapter 2 (React Components):

```
...
class HelloWorld extends React.Component {
  render() {
    return (
      <div title="Outer div">
        <h1>Hello World!</h1>
      </div>
    );
  }
}
...
```

This can be rewritten as a pure function like this:

```
...
function HelloWorld() {
  return (
    <div title="Outer div">
      <h1>Hello World!</h1>
    </div>
  );
}
...
```

If the rendering depends on the props alone (more often than not, this will indeed be the case), the function can be written with one argument as the props, which can be accessed within the function's JSX body. Say the Hello World component takes in a message as part of the props. The component can be rewritten as follows:

```
...
function HelloWorld(props) {
  return (
    <div title="Outer div">
```

[1]This is not entirely true. There is, in fact, state in this component: the state of the input fields as the user is typing. But we have not captured them as React State and have let the browser's native handlers maintain it. So we don't really treat it as a regular state.


```
      <h1>{props.message}</h1>
    </div>
  );
}
...
```

An even more concise form using an arrow function can be used when the rendered output can be represented as a JavaScript expression, that is, a function with no other statement than just the return statement:

```
...
const HelloWorld = (props) => (
  <div title="Outer div">
    <h1>{props.message}</h1>
  </div>
);
...
```

This `HelloWorld` component could have been instantiated like this:

```
...
  <HelloWorld message="Hello World" />
...
```

Since `IssueRow` and `IssueTable` are stateless components, let's change them to pure functions. The new components are shown in Listing 4-9 and Listing 4-10, respectively.

Listing 4-9. App.jsx, IssueRow as a Stateless Component

```
function IssueRow(props) {
  const issue = props.issue;
  return (
    <tr>
      <td>{issue.id}</td>
      <td>{issue.status}</td>
      <td>{issue.owner}</td>
      <td>{issue.created.toDateString()}</td>
      <td>{issue.effort}</td>
      <td>{issue.due ? issue.due.toDateString() : ''}</td>
      <td>{issue.title}</td>
    </tr>
  );
}
```

Listing 4-10. App.jsx, IssueTable as a Stateless Component

```
function IssueTable(props) {
  const issueRows = props.issues.map(issue =>
    <IssueRow key={issue.id} issue={issue} />
  );
```

```
  return (
    <table className="bordered-table">
      <thead>
        <tr>
          <th>ID</th>
          <th>Status</th>
          <th>Owner</th>
          <th>Created</th>
          <th>Effort</th>
          <th>Due Date</th>
          <th>Title</th>
        </tr>
      </thead>
      <tbody>
        {issueRows}
      </tbody>
    </table>
  );
}
```

Designing Components

Most beginners will have a bit of confusion between state and `props`, when to use which, what granularity of components should one choose, and how to go about it all. This section is devoted to discussing some principles and best practices.

State vs. Props

Both state and `props` hold model information, but they are different. The `props` are immutable, whereas state is not. Typically, state variables are passed down to child components as `props` because the children don't maintain or modify them. They take in a read-only copy and use it only to render the view of the component. If any event in the child affects the parent's state, the child calls a method defined in the parent. Access to this method should have been explicitly given by passing it as a callback via `props`.

Anything that *can* change due to an event anywhere in the component hierarchy qualifies as being part of the state. Avoid keeping computed values in the state; instead, simply compute them when needed, typically inside the `render()` method.

Do not copy `props` into state, just because `props` are immutable. If you feel the need to do this, think of modifying the original state from which these `props` were derived. One exception is when `props` are used as *initial* values to the state, and the state is truly disjointed from the original state after the initialization.

You can use Table 4-1 as a quick reference to the differences.

Table 4-1. *State vs. Props*

Attribute	State	Props
Mutability	Can be changed using `this.setState()`	Cannot be changed
Ownership	Belongs to the component	Belongs to an ancestor, the component gets a read-only copy
Information	Model information	Model information
Affects	Rendering of the component	Rendering of the component

Component Hierarchy

Split the application into components and subcomponents. Typically, this will reflect the data model itself. For example, in the Issue Tracker, the issues array was represented by the `IssueTable` component, and each issue was represented by the `IssueRow` component.

Decide on the granularity just as you would for splitting functions and objects. The component should be self-contained with minimal and logical interfaces to the parent. If you find it doing too many things, just like in functions, it should probably be split into multiple components, so that it follows the Single Responsibility principle (that is, every component should be responsible for one and only one thing). If you are passing in too many `props` to a component, it is an indication that either the component needs to be split, or it need not exist: the parent itself could do the job.

Communication

Communication between components depends on the direction. Parents communicate to children via `props`; when state changes, the `props` automatically change. Children communicate to parents via callbacks.

Siblings and cousins can't communicate with each other, so if there is a need, the information has to go up the hierarchy and then back down. This is called *lifting the state up*. This is what we did when we dealt with adding a new issue. The `IssueAdd` component had to insert a row in `IssueTable`. It was achieved by keeping the state in the least common ancestor, `IssueList`. The addition was initiated by `IssueAdd` and a new array element added in `IssueList`'s state via a callback. The result was seen in `IssueTable` by passing the `issues` array down as `props` from `IssueList`.

If there is a need to know the state of a child in a parent, you're probably doing it wrong. Although React does offer a way using `refs`, you shouldn't feel the need if you follow the one-way data flow strictly: state flows as props into children, events cause state changes, which flows back as props.

Stateless Components

In a well-designed application, most components would be stateless functions of their properties. All states would be captured in a few components at the top of the hierarchy, from where the props of all the descendants are derived.

We did just that with the `IssueList`, where we kept the state. We converted all descendent components to stateless components, relying only on props passed down the hierarchy to render themselves. We kept the state in `IssueList` because that was the least common component above all the descendants that depended on that state. Sometimes, you may find that there is no logical common ancestor. In such cases, you may have to invent a new component just to hold the state, even though visually the component has nothing.

Summary

In this chapter, you learned how to use state and make changes to it on user interactions or other events. The more interesting aspect was how state values are propagated down the component hierarchy as `props`. You also had a glimpse of user interaction: the click of a button to add a new issue, and how that causes the state to change, and in turn, how the props in the descendant components changed, causing them to rerender as well. Further, you learned how a child can communicate with its parent via callbacks.

We used simulated asynchronous calls and data local to the browser to achieve all this. In the next chapter, instead of using local data, we'll fetch the data from the server. When an issue is added, we'll send the data to the server to persist it.

Answers to Exercises

Exercise: Initial State

1. You could store the background color of each row as part of the state, but then, the values will have to be calculated at some point in time. When is a good time to do this? Just before setting the state? How about when setting the initial state?

   Since this is a *derived* value, it is better and more efficient to just calculate these values within the `render()` method and use them then and there rather than save them in the state.

Exercise: Updating State

1. When the second timer fires and another issue is added, you will find that it gets an ID of 4 but the ID of the third row also changes to 4. Further, in the console, you will see an error to the effect that two children with the same key were found.

   This happens because we are using the same object as the first to create the second issue, and setting the ID to 4 sets it in the one and only object: `sampleIssue`. To avoid this, you have to create a copy of the object before using it to create a new issue, say, using `Object.assign()`.

2. Each row is rendered once when initialized (two renders, one for each row). After the new row is inserted, each row is rendered once again (three renders, one for each row). Although a render is called, this does not mean that the DOM is updated. Only the virtual DOM is recreated on each render. Real DOM update happens only where there are differences.

Exercise: Lifting State Up

1. On removing the `bind()` call, you'll see an error that says *undefined is not an object*, while evaluating `this.state.issues`. This should tell you that `this.state` is undefined, and lead you to think about whether `this` is the correct `this` in this call sequence.

   In future, if you see a similar error, it should trigger off a thought that maybe a `bind()` call is missing somewhere.

Exercise: Event Handling

1. To persist the changes, we could either save the issues in local storage on the browser, or save it in the server. Modifying the global `initialIssues` variable will not work because when the page is refreshed, this variable is recreated.

2. The page is refreshed as if a new request to `/` has been made. In the URL bar, you can see URL query parameters for owner and title like `?owner=&title=`. This is because the default action of a form is a GET HTTP request with the values of the form, and what you see in the URL bar is just the result of this call. (The values in the URL parameters are blank because they were assigned to empty strings in `handleSubmit()`).

3. You will see that the subtree under `<tbody>` is being modified only once. In the details of the modification, you can see that a child is being added, but none of the other existing children are being modified. If you compare it with the number of `render()` method calls, you will find that even though `render()` was being called for each row, only the new row is being added to the DOM.

CHAPTER 5

Express and GraphQL

Now that you have learned about creating components and building a workable user interface using React, in this chapter, we'll spend some time integrating with the back-end server for the data.

Until now, the only resource the Express and Node.js server was serving was static content in the form of `index.html`. In this chapter, we'll start fetching and storing the data using APIs from the Express and Node.js server in addition to the static HTML file. This will replace the hard-coded array of issues in the browser's memory. We will be making changes to both front-end and back-end code, as we'll be implementing as well as consuming the APIs.

We will not persist the data on disk; instead, we'll just use a simulated database in the server's memory. We will leave actual persistence to the next chapter.

Express

I briefly touched upon Express and how to serve static files using Express in the Hello World chapter. But Express can do much more than just serve static files. Express is a minimal, yet, flexible web application framework. It's minimal in the sense that by itself, Express does very little. It relies on other modules called *middleware* to provide the functionality that most applications will need.

Routing

The first concept is that of routing. At the heart of Express is a router, which essentially takes a client request, matches it against any routes that are present, and executes the handler function that is associated with that route. The handler function is expected to generate the appropriate response.

A route specification consists of an HTTP method (GET, POST, etc.), a path specification that matches the request URI, and the route handler. The handler is passed in a request object and a response object. The request object can be inspected to get the various details of the request, and the response object's methods can be used to send the response to the client. All this may seem a little overwhelming, so let's just start with a simple example and explore the details.

We already have an Express application in the which we created using the `express()` function. We also installed a middleware for handling static files. A middleware function deals with *any* request matching the path specification, regardless of the HTTP method. In contrast, a route can match a request with a specific

V. Subramanian, *Pro MERN Stack*, https://doi.org/10.1007/978-1-4842-4391-6_5

HTTP method. So, instead of `app.use()`, `app.get()` has to be used in order to match the GET HTTP method. Further, the handler function, the second argument that the routing function takes, can set the response to be sent back to the caller like this:

```
...
app.get('/hello', (req, res) => {
  res.send('Hello World!');
});
...
```

Request Matching

When a request is received, the first thing that Express does is match the request to one of the routes. The request method is matched against the route's method. In the previous example, the route's method is `get()` so any HTTP request using the GET method will match it. Further, the request URL is matched with the path specification, the first argument in the route, which is `/hello`. When a HTTP request matches this specification, the handler function is called. In the previous example, we just responded with a text message.

The route's method and path need not be specific. If you want to match all HTTP methods, you could write `app.all()`. If you needed to match multiple paths, you could pass in an array of paths, or even a regular expression like `'/*.do'` will match any request ending with the extension `.do`. Regular expressions are rarely used, but route parameters are often used, so I'll discuss that in a little more detail.

Route Parameters

Route parameters are named segments in the path specification that match a part of the URL. If a match occurs, the value in that part of the URL is supplied as a variable in the request object.

This is used in the following form:

```
app.get('/customers/:customerId', ...)
```

The URL `/customers/1234` will match the route specification, and so will `/customers/4567`. In either case, the customer ID will be captured and supplied to the handler function as part of the request in `req.params`, with the name of the parameter as the key. Thus, `req.params.customerId` will have the value `1234` or `4567` for each of these URLs, respectively.

■ **Note** The query string is not part of the path specification, so you cannot have different handlers for different parameters or values of the query string.

Route Lookup

Multiple routes can be set up to match different URLs and patterns. The router does not try to find a best match; instead, it tries to match all routes in the order in which they are installed. The first match is used. So, if two routes are possible matches to a request, it will use the first defined one. So, the routes have to be defined in the order of priority.

Thus, if you add patterns rather than very specific paths, you should be careful to add the more generic pattern after the specific paths in case a request can match both. For example, if you want to match everything that goes under `/api/`, that is, a pattern like `/api/*`, you should add this route only after all the more specific routes that handle paths such as `/api/issues`.

Handler Function

Once a route is matched, the handler function is called, which in the previous example was an anonymous function supplied to the route setup function. The parameters passed to the handler are a request object and a response object. The handler function is not expected to return any value. But it can inspect the request object and send out a response as part of the response object based on the request parameters.

Let's briefly look at the important properties and methods of the request and response objects.

Request Object

Any aspect of the request can be inspected using the request object's properties and methods. A few important and useful properties and methods are listed here:

- `req.params`: This is an object containing properties mapped to the named route parameters as you saw in the example that used `:customerId`. The property's key will be the name of the route parameter (`customerId` in this case) and the value will be the actual string sent as part of the HTTP request.
- `req.query`: This holds a parsed query string. It's an object with keys as the query string parameters and values as the query string values. Multiple keys with the same name are converted to arrays, and keys with a square bracket notation result in nested objects (e.g., `order[status]=closed` can be accessed as `req.query.order.status`).
- `req.header, req.get(header)`: The `get` method gives access to any header in the request. The header property is an object with all headers stored as key-value pairs. Some headers are treated specially (like Accept) and have dedicated methods in the request object for them. That's because common tasks that depend on these headers can be easily handled.
- `req.path`: This contains the path part of the URL, that is, everything up to any `?` that starts the query string. Usually, the path is part of the route specification, but if the path is a pattern that can match different URLs, you can use this property to get the actual path that was received in the request.
- `req.url, req.originalURL`: These properties contain the complete URL, including the query string. Note that if you have any middleware that modifies the request URL, `originalURL` will hold the URL as it was received, before the modification.
- `req.body`: This contains the body of the request, valid for POST, PUT, and PATCH requests. Note that the body is not available (`req.body` will be undefined) unless a middleware is installed to read and optionally interpret or parse the body.

There are many other methods and properties; for a complete list, refer to the Request documentation of Express at http://expressjs.com/en/api.html#req as well as Node.js' request object at https://nodejs.org/api/http.html#http_class_http_incomingmessage, from which the Express Request is extended.

Response Object

The response object is used to construct and send a response. Note that if no response is sent, the client is left waiting.

- `res.send(body)`: You already saw the `res.send()` method briefly, which responded with a string. This method can also accept a buffer (in which case the content type is set as `application/octet-stream` as opposed to `text/html` in case of a string). If the body is an object or an array, it is automatically converted to a JSON string with an appropriate content type.
- `res.status(code)`: This sets the response status code. If not set, it is defaulted to `200 OK`. One common way of sending an error is by combining the `status()` and `send()` methods in a single call like `res.status(403).send("Access Denied")`.
- `res.json(object)`: This is the same as `res.send()`, except that this method forces conversion of the parameter passed into a JSON, whereas `res.send()` may treat some parameters like `null` differently. It also makes the code readable and explicit, stating that you are indeed sending out a JSON.
- `res.sendFile(path)`: This responds with the contents of the file at `path`. The content type of the response is guessed using the extension of the file.

There are many other methods and properties in the response object; you can look at the complete list in the Express documentation for Response at http://expressjs.com/en/api.html#res and Node.js' Response object in the HTTP module at https://nodejs.org/api/http.html#http_class_http_serverresponse. But for most common use, the previous methods should suffice.

Middleware

Express is a web framework that has minimal functionality of its own. An Express application is essentially a series of middleware function calls. In fact, the Router itself is nothing but a middleware function. The distinction is that middleware usually works on generic handling of requests and/or things that need to be done for all or most requests, but not necessarily be the last in the chain, that sends out a response. A route, on the other hand, is meant to be used for a specific path+method combination and is expected to send out a response.

Middleware functions are those that have access to the request object (`req`), the response object (`res`), and the next middleware function in the application's request-response cycle. The next middleware function is commonly denoted by a variable named `next`. I won't go into the details of how to write your own middleware function, since we will not be writing new middleware in the application. But we will use some middleware for sure, so it's handy to understand how any middleware works at a high level.

We already used one middleware called `express.static` in the Hello World example, to serve static files. This is the only built-in middleware (other than the router) available as part of Express. But there are other very useful middleware supported by the Express team, of which we will be using body-parser in this chapter, though indirectly. Third-party middleware is available via npm.

Middleware can be at the application level (applies to all requests) or at a specific path level (applies to specific request path patterns). The way to use a middleware at the application level is to simply supply the function to the application, like this:

```
app.use(middlewareFunction);
```

In the case of the `static` middleware, we constructed a middleware function by calling `express.static()` method. This not only returned a middleware function, but also configured it to use the directory called `public` to look for the static files.

In order to use the same middleware for only requests matching a certain URL path, say, `/public`, the `app.use()` method would have to be called with two arguments, the first one being the path, like this:

```
app.use('/public', express.static('public'));
```

This would have *mounted* the static middleware on the path `/public` and all static files would have to be accessed with the prefix `/public`, for example, `/public/index.html`.

REST API

REST (short for representational state transfer) is an architectural pattern for application programming interfaces (APIs). There are other older patterns such as SOAP and XMLRPC, but of late, the REST pattern has gained popularity.

Since the APIs in the Issue Tracker application are only for internal consumption, we could use any API pattern or even invent our own. But let's not do that because using an existing pattern forces you to think and organize the APIs and schema better and encourages some good practices.

Although we won't be using the REST pattern, I'll discuss it briefly since it is one of the more popular choices due to its simplicity and small number of constructs. It'll let you appreciate the differences and the logic for the choice that I'll make eventually, to use GraphQL.

Resource Based

The APIs are resource based (as opposed to action based). Thus, API names like `getSomething` or `saveSomething` are not normal in REST APIs. In fact, there are no API *names* in the conventional sense, because APIs are formed by a combination of resources and actions. There are really only resource names called *endpoints*.

Resources are accessed based on a Uniform Resource Identifier (URI), also known as an endpoint. Resources are nouns (not verbs). You typically use two URIs per resource: one for the collection (like `/customers`) and one for an individual object (like `/customers/1234`), where `1234` uniquely identifies a customer.

Resources can also form a hierarchy. For example, the collection of orders of a customer is identified by `/customers/1234/orders`, and an order of that customer is identified by `/customers/1234/orders/43`.

HTTP Methods as Actions

To access and manipulate the resources, you use HTTP methods. While resources were nouns, the HTTP methods are verbs that operate on them. They map to CRUD (Create, Read, Update, Delete) operations on the resource. Table 5-1 shows commonly used mapping of CRUD operations to HTTP methods and resources.

Table 5-1. *CRUD Mapping for HTTP Methods*

Operation	Method	Resource	Example	Remarks
Read – List	GET	Collection	`GET /customers`	Lists objects (additional query string can be used for filtering and sorting)
Read	GET	Object	`GET /customers/1234`	Returns a single object (query string may be used to specify which fields)
Create	POST	Collection	`POST /customers`	Creates an object with the values specified in the body
Update	PUT	Object	`PUT /customers/1234`	Replaces the object with the one specified in the body
Update	PATCH	Object	`PATCH /customers/1234`	Modifies some properties of the object, as specified in the body
Delete	DELETE	Object	`DELETE /customers/1234`	Deletes the object

Some other operations such as DELETE and PUT in the collection may also be used to delete and modify the entire collection in one shot, but this is not common usage. HEAD and OPTIONS are also valid verbs that give out information about the resources rather than actual data. They are used mainly for APIs that are externally exposed and consumed by many different clients.

Although the HTTP method and operation mapping are well mapped and specified, REST by itself lays down no rules for the following:

- Filtering, sorting, and paginating a list of objects. The query string is commonly used in an implementation-specific way to specify these.
- Specifying which fields to return in a READ operation.
- If there are embedded objects, specifying which of those to expand in a READ operation.
- Specifying which fields to modify in a PATCH operation.
- Representation of objects. You are free to use JSON, XML, or any other representation for the objects in both READ and WRITE operations.

Given the fact that different API sets use different ways of dealing with these issues, most REST API implementations are more REST-*like* than strict REST. This has affected common adoption and there are, therefore, a lack of tools that can help do much of the common things that need to be done to implement a REST-based API.

GraphQL

Although the REST paradigm is quite useful in making APIs predictable, the shortcomings discussed previously have made it hard to use it when different clients access the same set of APIs. For example, how an object is displayed in a mobile application and the same is displayed in a desktop browser can be quite different, and therefore, a more granular control as well as aggregation of different resources may work better.

GraphQL was developed to address just these concerns. As a result, GraphQL is a far more elaborate specification, with the following salient features.

CHAPTER 6

MongoDB

In this chapter, I'll take up MongoDB, the database layer and the M in the MERN stack. Until now, we had an array of issues in the Express server's memory that we used as the database. We'll replace this with real persistence and read and write the list of issues from a MongoDB database.

To achieve this, we'll need to install or use MongoDB on the cloud, get used to its shell commands, install a Node.js driver to access it from Node.js, and finally modify the server code to replace the API calls to read and write from a MongoDB database instead of the in-memory array of issues.

MongoDB Basics

This is an introductory section, where we will not be modifying the application. We'll look at these core concepts in this section: MongoDB, documents, and collections. Then, we'll set up MongoDB and explore these concepts with examples using the mongo shell to read and write to the database.

Documents

MongoDB is a document database, which means that the equivalent of a record is a document, or an object. In a relational database, you organize data in terms of rows and columns, whereas in a document database, an entire object can be written as a document.

For simple objects, this may seem no different from a relational database. But let's say you have objects with nested objects (called *embedded* documents) and arrays. Now, when using a relational database, this will typically need multiple tables. For example, in a relational database, an Invoice object may be stored in a combination of an `invoice` table (to store the invoice details such as the customer address and delivery details) and an `invoice_items` table (to store the details of each item that is part of the shipment). In MongoDB, the entire Invoice object would be stored as one document. That's because a document can contain arrays and other objects in a nested manner and the contained objects don't have to be separated out into other documents.

A document is a data structure composed of field and value pairs. The values of fields may include objects, arrays, and arrays of objects and so on, as deeply nested as you want it to be. MongoDB documents are similar to JSON objects, so it is easy to think of them as JavaScript objects. Compared to a JSON object, a MongoDB document has support not only for the primitive data types—Boolean, numbers, and strings—but also other common data types such as dates, timestamps, regular expressions, and binary data.

An Invoice object may look like this:

```
{
  "invoiceNumber" : 1234,
  "invoiceDate" : ISODate("2018-10-12T05:17:15.737Z"),
  "billingAddress" : {
```


V. Subramanian, *Pro MERN Stack*, https://doi.org/10.1007/978-1-4842-4391-6_6


```
    "name" : "Acme Inc.",
    "line1" : "106 High Street",
    "city" : "New York City",
    "zip" : "110001-1234"
  },
  "items" : [
    {
      "description" : "Compact Flourescent Lamp",
      "quantity" : 4,
      "price" : 12.48
    },
    {
      "description" : "Whiteboard",
      "quantity" : 1,
      "price" : 5.44
    }
  ]
}
```

In this document, there are numbers, strings, and a date data type. Further, there is a nested object (`billingAddress`) and an array of objects (`items`).

Collections

A collection is like a table in a relational database: it is a set of documents. Just like in a relational database, the collection can have a primary key and indexes. But there are a few differences compared to a relational database.

A primary key is mandated in MongoDB, and it has the reserved field name `_id`. Even if `_id` field is not supplied when creating a document, MongoDB creates this field and auto-generates a unique key for every document. More often than not, the auto-generated ID can be used as is, since it is convenient and guaranteed to produce unique keys even when multiple clients are writing to the database simultaneously. MongoDB uses a special data type called the `ObjectId` for the primary key.

The `_id` field is automatically indexed. Apart from this, indexes can be created on other fields, and this includes fields within embedded documents and array fields. Indexes are used to efficiently access a subset of documents in a collection.

Unlike a relational database, MongoDB does not require you to define a schema for a collection. The only requirement is that all documents in a collection must have a unique `_id`, but the actual documents may have completely different fields. In practice, though, all documents in a collection do have the same fields. Although a flexible schema may seem very convenient for schema changes during the initial stages of an application, this can cause problems if some kind of schema checking is not added in the application code.

As of version 3.6, MongoDB has supported a concept of schema, even though it is optional. You can read all about MongoDB schemas at `https://docs.mongodb.com/manual/core/schema-validation/index.html`. A schema can enforce allowed and required fields and their data types, just like GraphQL can. But it can also validate other things like string length and minimum and maximum values for integers.

But the errors generated because of schema violations do not give enough details as to which of the validation checks fail as of version 3.6. This may improve in future versions of MongoDB, at which point in time it is worth considering adding full-fledged schema checks. For the Issue Tracker application, we'll not use the schema validation feature of MongoDB, instead, we'll implement all necessary validations in the back-end code.

Databases

A database is a logical grouping of many collections. Since there are no foreign keys like in a SQL database, the concept of a database is nothing but a logical partitioning namespace. Most database operations read or write from a single collection, but `$lookup`, which is a stage in an aggregation pipeline, is equivalent to a join in SQL databases. This stage can combine documents within the same database.

Further, taking backups and other administrative tasks work on the database as a unit. A database connection is restricted to accessing only one database, so to access multiple databases, multiple connections are required. Thus, it is useful to keep all the collections of an application in one database, though a database server can host multiple databases.

Query Language

Unlike the universal English-like SQL in a relational database, the MongoDB query language is made up of *methods* to achieve various operations. The main methods for read and write operations are the CRUD methods. Other methods include aggregation, text search, and geospatial queries.

All methods operate on a collection and take parameters as JavaScript objects that specify the details of the operation. Each method has its own specification. For example, to insert a document, the only argument needed is the document itself. For querying, the parameters are a query filter and a list of fields to return (also called the *projection*).

The query filter is a JavaScript object consisting of zero or more properties, where the property name is the name of the field to match on and the property value consists of another object with an operator and a value. For example, to match all documents with the field `invoiceNumber` that are greater than 1,000, the following query filter can be used:

```
{ "invoiceNumber": { $gt: 1000 } }
```

Since there is no "language" for querying or updating, the query filters can be very easily constructed programmatically.

Unlike relational databases, MongoDB encourages denormalization, that is, storing related parts of a document as embedded subdocuments rather than as separate collections (tables) in a relational database. Take an example of people (name, gender, etc.) and their contact information (primary address, secondary address etc.). In a relational database, this would require separate tables for People and Contacts, and then a join on the two tables when all of the information is needed together. In MongoDB, on the other hand, it can be stored as a list of contacts *within* the same People document. That's because a join of collections is not natural to most methods in MongoDB: the most convenient `find()` method can operate only on one collection at a time.

Installation

Before you try to install MongoDB on your computer, you may want to try out one of the hosted services that give you access to MongoDB. There are many services, but the following are popular and have a free version that you can use for a small test or sandbox application. Any of these will do quite well for the purpose of the Issue Tracker application that we'll build as part of this book.

- MongoDB Atlas (https://www.mongodb.com/cloud/atlas): I refer to this as Atlas for short. A small database (shared RAM, 512 MB storage) is available for free.
- mLab (previously MongoLab) (https://mlab.com/): mLab has announced an acquisition by MongoDB Inc. and may eventually be merged into Atlas itself. A sandbox environment is available for free, limited to 500 MB storage.

- Compose (https://www.compose.com): Among many other services, Compose offers MongoDB as a service. A 30-day trial period is available, but a permanently free sandbox kind of option is not available.

Of these three, I find Atlas the most convenient because there are many options for the location of the host. When connecting to the database, it lets me choose one closest to my location, and that minimizes the latency. mLab does not give a cluster—a database can be created individually. Compose is not permanently free, and it is likely that you may need more than 30 days to complete this book.

The downside of any of the hosted options is that, apart from the small extra latency when accessing the database, you need an Internet connection. Which means that you may not be able to test your code where Internet access is not available, for example, on a flight. In comparison, installing MongoDB on your computer may work better, but the installation takes a bit more work than signing up for one of the cloud-based options.

Even when using one of the cloud options, you will need to download and install the mongo shell to be able to access the database remotely. Each of the services come with instructions on this step as well. Choose version 3.6 or higher of MongoDB when signing up for any of these services. Test the signup by connecting to the cluster or database using the mongo shell, by following instructions given by the service provider.

If you choose to install MongoDB on your computer (it can be installed easily on OS X, Windows, and most distributions based on Linux), look up the installation instructions, which are different for each operating system. You may install MongoDB by following the instructions at the MongoDB website (https://docs.mongodb.com/manual/installation/ or search for "mongodb installation" in your search engine).

Choose MongoDB version 3.6 or higher, preferably the latest, as some of the examples use features introduced only in version 3.6. Most local installation options let you install the server, the shell, and tools all in one. Check that this is the case; if not, you may have to install them separately.

After a local installation, ensure that you have started MongoDB server (the name of the daemon or service is mongod), if it is not already started by the installation process. Test the installation by running the mongo shell like this:

```
$ mongo
```

On a Windows system, you may need to append `.exe` to the command. The command may require a path depending on your installation method. If the shell starts successfully, it will also connect to the local MongoDB server instance. You should see the version of MongoDB printed on the console, the database it is connecting to (the default is test), and a command prompt, like this, if you had installed MongoDB version 4.0.2 locally:

```
MongoDB shell version v4.0.2
connecting to: mongodb://127.0.0.1:27017
MongoDB server version: 4.0.2
>
```

The message you see can be slightly different from this, especially if you have installed a different version of MongoDB. But you do need to see the prompt `>` where you can type further commands. If, instead, you see an error message, revisit the installation and the server starting procedure.

The Mongo Shell

The mongo shell is an interactive JavaScript shell, very much like the Node.js shell. In the interactive shell, a few non-JavaScript conveniences are available over and above the full power of JavaScript. In this section, we'll discuss the basic operations that are possible via the shell, those that are most commonly used. For a full reference of all the capabilities of the shell, you can take a look at the mongo shell documentation at https://docs.mongodb.com/manual/mongo/.

The commands that we will be typing in the mongo shell have been collected together in a file called `mongo_commands.txt`. These commands have been tested to work as is on Atlas or a local installation, but you may find variations in the other options. For example, mLab lets you connect only to a database (as opposed to a cluster), so it does not allow of switching between databases in mLab.

■ **Note** If you find that something is not working as expected when typing a command, cross-check the commands with the same in the GitHub repository (https://github.com/vasansr/pro-mern-stack-2). This is because typos may have been introduced during the production of the book, or last-minute corrections may have missed making it to the book. The GitHub repository, on the other hand, reflects the most up-to-date and tested set of code and commands.

To work with MongoDB, you need to connect to a database. Let's start with finding which databases are available. The command to show the current databases is:

```
> show databases
```

This will list the databases and the storage occupied by them. For example, in a fresh local installation of MongoDB, this is what you will see:

```
admin          0.000GB
config         0.000GB
local          0.000GB
```

These are system databases that MongoDB uses for its internal book keeping, etc. We will not be using any of these to create our collections, so we'd better change the current database. To identify the current database, the command is:

```
> db
```

The default database a mongo shell connects to is called `test` and that is what you are likely to see as the output to this command. Let's now see what collections exist in this database.

```
> show collections
```

You will find that there are no collections in this database, since it is a fresh installation. Further, you will also find that the database `test` was not listed when we listed the available databases. That's because databases and collections are really created only on the first write operation to any of these.

Let's switch to a database called `issuetracker` instead of using the default database:

```
> use issuetracker
```


This should result in output that confirms that the new database is `issuetracker`:

```
switched to db issuetracker
```

Let's confirm that there are no collections in this database either:

```
> show collections
```

This command should return nothing. Now, let's create a new collection. This is done by creating one document in a collection. A collection is referenced as a property of the global object `db`, with the same name as the collection. The collection called `employees` can be referred to as `db.employees`. Let's insert a new document in the `employees` collection using the `insertOne()` method. This method takes in the document to be inserted as an argument:

```
> db.employees.insertOne({ name: { first: 'John', last: 'Doe' }, age: 44 })
```

The result of this command will show you the result of the operation and the ID of the new document that was created, something like this:

```
{
    "acknowledged" : true,
    "insertedId" : ObjectId("5bbc487a69d13abf04edf857")
}
```

Apart from the `insertOne()` method, many methods are available on any collection. You can see the list of available methods by pressing the Tab character twice after typing "`db.employees.`" (the period at the end is required before pressing Tab). You may find an output like the following:

```
db.employees.addIdIfNeeded(              db.employees.getWriteConcern(
db.employees.aggregate(                  db.employees.group(
db.employees.bulkWrite(                  db.employees.groupcmd(
db.employees.constructor                 db.employees.hasOwnProperty
db.employees.convertToCapped(            db.employees.hashAllDocs(
db.employees.convertToSingleObject(      db.employees.help(
db.employees.copyTo(                     db.employees.initializeOrderedBulkOp(
db.employees.count(                      db.employees.initializeUnorderedBulkOp(
db.employees.createIndex(                db.employees.insert(
db.employees.createIndexes(              db.employees.insertMany(
db.employees.dataSize(                   db.employees.insertOne(
...
```

This is the auto-completion feature of the mongo shell at work. Note that you can let the mongo shell auto-complete the name of any method by pressing the Tab character after entering the beginning few characters of the method.

Let's now check if the document has been created in the collection. To do that, we can use the `find()` method on the collection. Without any arguments, this method just lists all the documents in the collection:

```
> db.employees.find()
```

This should result in displaying the document we just created, but it is not "pretty" formatted. It will be printed all in one line and may wrap around to the next line inconveniently. To get a more legible output, we can use the `pretty()` method on the result of the `find()` method:

```
> db.employees.find().pretty()
```

That should show a much more legible output, like this:

```
{
     "_id" : ObjectId("5bbc487a69d13abf04edf857"),
     "name" : {
            "first" : "John",
            "last" : "Doe"
     },
     "age" : 44
}
```

At this point in time, if you execute `show collections` and `show databases`, you will find that the `employees` collection and the `issuetracker` database have indeed been created and are listed in the output of their respective commands. Let's insert another document in the same collection and try to deal with multiple documents in the collection:

```
> db.employees.insertOne({ name: { first: 'Jane', last: 'Doe' }, age: 54 })
```

Now, since we have the full power of JavaScript in the shell, let's try to exercise some of it to get a taste. Instead of printing the results onscreen, let's collect the results into a JavaScript array variable. The result of the `find()` method was a cursor that could be iterated. In the `cursor` object, there are methods other than `pretty()`, one of which is `toArray()`. This method reads all the documents from the query and places them in an array. So, let's use this method and assign its result to an array variable.

```
> let result = db.employees.find().toArray()
```

Now, the variable `result` should be an array with two elements, each an `employee` document. Let's use the JavaScript array method `forEach()` to iterate through them and print the first names of each employee:

```
> result.forEach((e) => print('First Name:', e.name.first))
```

This should give an output like this:

```
First Name: John
First Name: Jane
```

In Node.js, the `console.log` method is available for printing objects on the console. The mongo shell, on the other hand, provides the `print()` method for the same purpose, but this prints only strings. Objects need to be converted to strings before printing, using the utility function `tojson()`. There is also another method, called `printjson()`, which prints objects as JSON. Let's use that to inspect the contents of the nested document `name` instead of only the first name:

```
> result.forEach((e) => printjson(e.name))
```


Now, you should see the `name` object expanded into first and last names, like the following:

```
{ "first" : "John", "last" : "Doe" }
{ "first" : "Jane", "last" : "Doe" }
```

The shell by itself does very little apart from providing a mechanism to access methods of the database and collections. It is the JavaScript engine, which forms the basis of the shell and gives a lot of flexibility and power to the shell.

In the next section, we will discuss more methods on the collection, such as `insertOne()` that you just learned about. These methods are accessible from many programming languages via a driver. The mongo shell is just another tool that can access these methods. You will find that the methods and arguments available in other programming languages are very similar to those in the mongo shell.

EXERCISE: MONGODB BASICS

1. Using the shell, display a list of methods available on the `cursor` object. Hint: Look up the mongo shell documentation for mongo Shell Help at https://docs.mongodb.com/manual/tutorial/access-mongo-shell-help/.

Answers are available at the end of the chapter.

MongoDB CRUD Operations

Since the mongo shell is the easiest to try out, let's explore the CRUD operations available in MongoDB using the shell itself. We will continue to use the `issuetracker` database we created in the previous section. But let's clear the database so that we can start fresh. The collection object provides a convenient method for erasing itself called `drop()`:

```
> db.employees.drop()
```

This should result in an output like this:

```
true
```

This is different from removing all the documents in the collection, because it also removes any indexes that are part of the collection.

Create

In the previous section, you briefly saw how to insert a document, and as part of that, you found how MongoDB automatically created the primary key, which was a special data type called `ObjectID`. Let's now use our own ID instead of letting MongoDB auto-generate one.

```
> db.employees.insertOne({
  _id: 1,
  name: { first: 'John', last: 'Doe' },
  age: 44
})
```


This will result in the following output:

```
{ "acknowledged" : true, "insertedId" : 1 }
```

Note that the value of `insertedId` reflected the value that we supplied for `_id`. Which means that, instead of an `ObjectID` type of value, we were able to supply our own value. Let's try to create a new identical document (you can use the Up Arrow key to repeat the previous command in the mongo shell). It will fail with the following error:

```
WriteError({
     "index" : 0,
     "code" : 11000,
     "errmsg" : "E11000 duplicate key error collection: issuetracker.employees index:
     _id_ dup key: { : 1.0 }",
     "op" : {
            "_id" : 1,
            "name" : {
                  "first" : "John",
                  "last" : "Doe"
            },
            "age" : 44
     }
})
```

This shows that the `_id` field continues to be a primary key and it is expected to be unique, regardless of whether it is auto-generated or supplied in the document. Now, let's add another document, but with a new field as part of the name, say, the middle name:

```
> db.employees.insertOne({
  name: {first: 'John', middle: 'H', last: 'Doe'},
  age: 22
})
```

This works just fine, and using `find()`, you can see that two documents exist in the collection, but they are not necessarily the same schema. This is the advantage of a flexible schema: the schema can be enhanced whenever a new data element that needs to be stored is discovered, without having to explicitly modify the schema.

In this case, it is implicit that any employee document where the `middle` field under `name` is missing indicates an employee without a middle name. If, on the other hand, a field was added that didn't have an implicit meaning when absent, its absence would have to be handled in the code. Or a migration script would have to be run that defaults the field's value to something.

You will also find that the format of the `_id` field is different for the two documents, and even the data type is different. For the first document, the data type is an integer. For the second, it is of type `ObjectID` (which is why it is shown as `ObjectID(...)`. Thus, it's not just the presence of fields that can differ between two documents in the same collection, even the data types of the same field can be different.

In most cases, leaving the creation of the primary key to MongoDB works just great, because you don't have to worry about keeping it unique: MongoDB does that automatically. But, this identifier is not human-readable. In the Issue Tracker application, we want the identifier to be a number so that it can be easily

remembered and talked about. But instead of using the `_id` field to store the human-readable identifier, let's use a new field called `id` and let MongoDB auto-generate `_id`.

So, let's drop the collection and start creating new documents with a new field called `id`.

```
> db.employees.drop()

> db.employees.insertOne({
  id: 1,
  name: { first: 'John', last: 'Doe' },
  age: 48
})

> db.employees.insertOne({
  id: 2,
  name: { first: 'Jane', last: 'Doe'} ,
  age: 16
})
```

The collection has a method that can take in multiple documents in one go. This method is called `insertMany()`. Let's use that to create a few more documents in a single command:

```
> db.employees.insertMany([
  { id: 3, name: { first: 'Alice', last: 'A' }, age: 32 },
  { id: 4, name: { first: 'Bob', last: 'B' }, age: 64 },
])
```

The response to this would have shown that multiple `insertedIds` that were created as opposed to a single `insertedId` for the `insertOne()` method, like this:

```
{
     "acknowledged" : true,
     "insertedIds" : [
            ObjectId("5bc6d80005fb87b8f2f5cf6f"),
            ObjectId("5bc6d80005fb87b8f2f5cf70")
     ]
}
```

Read

Now that there are multiple documents in the collection, let's see how to retrieve a subset of the documents as opposed to the full list. The `find()` method takes in two more arguments. The first is a filter to apply to the list, and the second is a projection, a specification of which fields to retrieve.

The filter specification is an object where the property name is the field to filter on, and the value is its value that it needs to match. Let's fetch one employee's document, identified by the `id` being equal to 1. Since we know that there can only be one employee for the given ID, let's use `findOne()` rather than `find()`. The method `findOne()` is a variation of the method `find()`, and it returns a single object rather than a cursor.

```
> db.employees.findOne({ id: 1 })
```

This should return the first employee document that we created, and the output will look like this:

```
{
     "_id" : ObjectId("5bc6d7e505fb87b8f2f5cf6d"),
     "id" : 1,
     "name" : {
            "first" : "John",
            "last" : "Doe"
     },
     "age" : 48
}
```

Note that we did not use `pretty()` here, yet, the output is prettified. This is because `findOne()` returns a single object and the mongo shell prettifies objects by default.

The filter is actually a shorthand for `{ id: { $eq: 1 } }`, where `$eq` is the operator signifying that the value of the field `id` has to be *equal* to 1. In the generic sense, the format of a single element in the filter is `fieldname: { operator: value }`. Other operators for comparison are available, such as `$gt` for greater than, etc. Let's try the `$gte` (greater than or equal to) operator for fetching a list of employees aged 30 or older:

```
> db.employees.find({ age: { $gte: 30 } })
```

That command should return three documents, because we inserted those many whose age was more than 30. If multiple fields are specified, then all of them have to match, which is the same as combining them with an *and* operator:

```
> db.employees.find({ age: { $gte: 30 }, 'name.last': 'Doe'  })
```

The number of documents returned now should be reduced to only one, since there is only one document that matched both the criteria, the last name being equal to `'Doe'` as well as age being greater than 30. Note that we used the dot notation for specifying a field embedded in a nested document. And this also made us use quotes around the field name, since it is a regular JavaScript object property.

To match multiple values of the same field—for example, to match age being greater than 30 *and* age being less than 60—the same strategy cannot be used. That's because the filter is a regular JavaScript object, and two properties of the same name cannot exist in a document. Thus, a filter like `{ age: { $gte: 30 }, age: { $lte: 60 } }` will not work (JavaScript will not throw an error, instead, it will pick just one of the values for the property age). An explicit `$and` operator has to be used, which takes in an array of objects specifying multiple field-value criteria. You can read all about the `$and` operator and many more operators in the operators section of the reference manual of MongoDB at https://docs.mongodb.com/manual/reference/operator/query/.

When filtering on a field is a common occurrence, it's typically a good idea to create an index on that field. The `createIndex()` method on the collection is meant for this purpose. It takes in an argument specifying the fields that form the index (multiple fields will form a composite index). Let's create an index on the age field:

```
> db.employees.createIndex({ age: 1 })
```

With this index, any query that uses a filter that has the field age in it will be significantly faster because MongoDB will use this index instead of scanning through all documents in the collection. But this was not a unique index, as many people can be the same age.

The age field is probably not a frequently used filter, but fetching a document based on its identifier is going to be very frequent. MongoDB automatically creates an index on the `_id` field, but we have used our own identifier called `id`, and this field is what is more likely to be used to fetch individual employees. So let's create an index on this field. Further, it has to be unique since it identifies the employee: no two employees should have the same value for `id`. The second argument to `createIndex()` is an object that contains various attributes of the index, one of them specifying whether the index is unique. Let's use that to create a unique index on `id`:

```
> db.employees.createIndex({ id: 1 }, { unique: true })
```

Now, not only will the `find()` method perform much better when a filter with `id` is supplied, but creation of a document with a duplicate `id` will be prevented by MongoDB. Let's try that by re-running the `insert` command for the first employee:

```
> db.employees.insertOne({
  id: 1,
  name: { first: 'John', last: 'Doe' },
  age: 48
})
```

Now, you should see an error in the mongo shell like this (the `ObjectID` will be different for you):

```
WriteError({
     "index" : 0,
     "code" : 11000,
     "errmsg" : "E11000 duplicate key error collection: issuetracker.employees index:
     id_1 dup key: { : 1.0 }",
     "op" : {
            "_id" : ObjectId("5bc04b8569334c5ff5bb7e8c"),
            "id" : 1
               ...
     }
})
```

Projection

All this while, we retrieved the entire document that matched the filter. In the previous section, when we had to print only a subset of the fields of the document, we did it using a `forEach()` loop. But this means that the entire document is fetched from the server even when we needed only some parts of it for printing. When the documents are large, this can use up a lot of network bandwidth. To restrict the fetch to only some fields, the `find()` method takes a second argument called the *projection*. A projection specifies which fields to include or exclude in the result.

The format of this specification is an object with one or more field names as the key and the value as 0 or 1, to indicate exclusion or inclusion. But 0s and 1s cannot be combined. You can either start with nothing and include fields using 1s, or start with everything and exclude fields using 0s. The `_id` field is an exception; it is always included unless you specify a 0. The following will fetch all employees but only their first names and age:

```
> db.employees.find({}, { 'name.first': 1, age: 1 })
```


Note that we specified an empty filter, to say that all documents have to be fetched. This had to be done since the projection is the second argument. The previous request would have printed something like this:

```
{ "_id" : ObjectId("5bbc...797855"), "name" : { "first" : "John" }, "age" : 48 }
{ "_id" : ObjectId("5bbc...797856"), "name" : { "first" : "Jane" }, "age" : 16 }
{ "_id" : ObjectId("5bbc...797857"), "name" : { "first" : "Alice" }, "age" : 32 }
{ "_id" : ObjectId("5bbc...797858"), "name" : { "first" : "Bob" }, "age" : 64 }
```

Even though we specified only the first name and age, the field `_id` was automatically included. To suppress the inclusion of this field, it needs to be explicitly excluded, like this:

```
> db.employees.find({}, { _id: 0, 'name.first': 1, age: 1 })
```

Now, the output will exclude the ID and look like this:

```
{ "name" : { "first" : "John" }, "age" : 48 }
{ "name" : { "first" : "Jane" }, "age" : 16 }
{ "name" : { "first" : "Alice" }, "age" : 32 }
{ "name" : { "first" : "Bob" }, "age" : 64 }
```

Update

There are two methods—`updateOne()` and `updateMany()`—available for modifying a document. The arguments to both methods are the same, except that `updateOne()` stops after finding and updating the first matching document. The first argument is a query filter, the same as the filter that `find()` takes. The second argument is an update specification if only some fields of the object need to be changed.

When using `updateOne()`, the primary key or any unique identifier is what is normally used in the filter, because the filter can match only one document. The update specification is an object with a series of `$set` properties whose values indicate another object, which specifies the field and its new value. Let's update the age of the employee identified by the `id` 2:

```
> db.employees.updateOne({ id: 2 }, { $set: {age: 23 } })
```

This should result in the following output:

```
{ "acknowledged" : true, "matchedCount" : 1, "modifiedCount" : 1 }
```

The `matchedCount` returned how many documents matched the filter. If the filter had matched more than one, that number would have been returned. But since the method is supposed to modify only one document, the modified count should always be 1, unless the modification had no effect. If you run the command again, you will find that `modifiedCount` will be 0, since the age was already 23 for the employee with ID 2.

To modify multiple documents in one shot, the `updateMany()` method has to be used. The format is the same as the `updateOne()` method, but the effect is that *all* documents that match will be modified. Let's add an `organization` field to all employees using the `updateMany()` method:

```
> db.employees.updateMany({}, { $set: { organization: 'MyCompany' } })
```


Note that even though the field `organization` did not exist in the documents, the new value `MyCompany` would have been applied to all of them. If you execute the command `find()` to show the companies alone in the projection, this fact will be confirmed.

There is also a method to *replace* the complete document called `replaceOne()`. Instead of specifying which fields to modify, if the complete modified document is available, it can be used to just replace the existing document with the new one. Here's an example:

```
> db.employees.replaceOne({ id: 4 }, {
  id: 4,
  name : { first : "Bobby" },
  age : 66
});
```

This command will replace the existing document with ID 4, with the new one. The fact that the `organization` and `name.last` fields were not specified will have the effect that these fields will not *exist* in the replaced document, as opposed to not *changed* using `updateOne()`. Getting the replaced object should show proof of that:

```
> db.employees.find({ id: 4 })
```

This should result in a document that looks as follows:

```
{ "_id" : ObjectId("5c38ae3da7dc439456c0281b"), "id" : 4, "name" : { "first" : "Bobby" },
"age" : 66 }
```

You can see that it no longer has the fields `name.last` and `organization`, because these were not specified in the document that was supplied to the command `replaceOne()`. It just replaces the document with the one supplied, except for the field `ObjectId`. Being the primary key, this field cannot be changed via an `updateOne()` or a `replaceOne()`.

Delete

The `delete` operation takes a filter and removes the document from the collection. The filter format is the same, and the variations `deleteOne()` and `deleteMany()` are both available, just as in the `update` operation.

Let's delete the last document, with ID 4:

```
> db.employees.deleteOne({ id: 4 })
```

This should result in the following output, confirming that the deletion affected only one document:

```
{ "acknowledged" : true, "deletedCount" : 1 }
```

Let's also cross-check the deletion by looking at the size of the collection. The `count()` method on the collection tells us how many documents it contains. Executing that now should return the value 3, because we originally inserted four documents.

```
> db.employees.count()
```

Aggregate

The `find()` method is used to return all the documents or a subset of the documents in a collection. Many a time, instead of the list of documents, we need a summary or an aggregate, for example, the count of documents that match a certain criterion.

The `count()` method can surely take a filter. But what about other aggregate functions, such as sum? That is where the `aggregate()` comes into play. When compared to relational databases supporting SQL, the `aggregate()` method performs the function of the GROUP BY clause. But it can also perform other functions such as a join, or even an unwind (expand the documents based on arrays within), and much more.

You can look up the advanced features that the `aggregate()` function supports in the MongoDB documentation at https://docs.mongodb.com/manual/reference/operator/aggregation-pipeline/ but for now, let's look at the real aggregation and grouping construct that it provides.

The `aggregate()` method works in a pipeline. Every *stage* in the pipeline takes the input from the result of the previous stage and operates as per its specification to result in a new modified set of documents. The initial input to the pipeline is, of course, the entire collection. The pipeline specification is in the form of an array of objects, each element being an object with one property that identifies the pipeline stage type and the value specifying the pipeline's effect.

For example, the `find()` method can be replicated using `aggregate()` by using the stages `$match` (the filter) and `$project` (the projection). To perform an actual aggregation, the `$group` stage needs to be used. The stage's specification includes the grouping key identified by the property `_id` and other fields as keys, whose values are aggregation specifications and fields on which the aggregation needs to be performed. The `_id` can be null to group on the entire collection.

Let's try this by getting the total age of all employees in the entire collection. There will be only one element in the array of pipelines, an object with a single property `$group`. In the value, `_id` will be set to `null` because we don't want to group by any field. We'll need to sum (using the aggregate function `$sum`) the field `age` into a new field called `total_age` like this:

```
> db.employees.aggregate([
  { $group: { _id: null, total_age: { $sum: '$age' } } }
])
```

This should result in an output like this:

```
{ "_id" : null, "total_age" : 103 }
```

The same function, `$sum`, can be used to get a count of the records by simply summing the value 1:

```
> db.employees.aggregate([
  { $group: { _id: null, count: { $sum: 1 } } }
])
```

To group the aggregate by a field, we'll need to specify the name of the field (prefixed by a $) as the value of `_id`. Let's use the `organization` field, but before that, let's insert a new document with an organization different from the rest of the documents (which were all set to `MyCompany`):

```
> db.employees.insertOne({
  id: 4,
  name: { first: 'Bob', last: 'B' },
  age: 64,
  organization: 'OtherCompany'
})
```


Now, here's the command that aggregates the age using sum across different organizations:

```
> db.employees.aggregate([
  { $group: { _id: '$organization', total_age: { $sum: '$age' } } }
])
```

This should result in an output like this:

```
{ "_id" : "OtherCompany", "total_age" : 64 }
{ "_id" : "MyCompany", "total_age" : 103 }
```

Let's also try another aggregate function, say average, using `$avg`:

```
> db.employees.aggregate([
  { $group: { _id: '$organization', average_age: { $avg: '$age' } } }
])
```

This should now result in an output like this:

```
{ "_id" : "OtherCompany", "average_age" : 64 }
{ "_id" : "MyCompany", "average_age" : 34.333333333333336 }
```

There are other aggregation functions, including minimum and maximum. For the complete set, refer to the documentation at https://docs.mongodb.com/manual/reference/operator/aggregation/group/#accumulator-operator.

EXERCISE: MONGODB CRUD OPERATIONS

1. Write a simple statement to retrieve all employees who have middle names. Hint: Look up the MongoDB documentation for query operators at https://docs.mongodb.com/manual/reference/operator/query/.
2. Is the filter specification a JSON? Hint: Think about date objects and quotes around field names.
3. Say an employee's middle name was set mistakenly, and you need to remove it. Write a statement to do this. Hint: Look up the MongoDB documentation for update operators at https://docs.mongodb.com/manual/reference/operator/update/.
4. During index creation, what did the 1 indicate? What other valid values are allowed? Hint: Look up the MongoDB indexes documentation at https://docs.mongodb.com/manual/indexes/.

Answers are available at the end of the chapter.

MongoDB Node.js Driver

This is the Node.js driver that lets you connect and interact with the MongoDB server. It provides methods very similar to what you saw in the mongo shell, but not exactly the same. Instead of the low-level MongoDB driver, we could use an Object Document Mapper called Mongoose, which has a higher level of abstraction and more convenient methods. But learning about the lower-level MongoDB driver may give you a better handle on the actual working of MongoDB itself, so I've chosen to use the low-level driver for the Issue Tracker application.

To start, let's install the driver:

```
$ npm install mongodb@3
```

Let's also start a new Node.js program just to try out the different ways that the driver's methods can be used. In the next section, we'll use some code from this trial to integrate the driver into the Issue Tracker application. Let's call this sample Node.js program `trymongo.js` and place it in a new directory called `scripts`, to distinguish it from other files that are part of the application.

The first thing to do is make a connection to the database server. This can be done by first importing the object `MongoClient` from the driver, then creating a new client object from it using a URL that identifies a database to connect to, and finally calling the `connect` method on it, like this:

```
...
const { MongoClient } = require('mongodb');

const client = new MongoClient(url);
client.connect();
...
```

The URL should start with `mongodb://` followed by the hostname or the IP address of the server to connect to. An optional port can be added using `:` as the separator, but it's not required if the MongoDB server is running on the default port, 27017. It's good practice to separate the connection parameters into a configuration file rather than keep them in a checked-in file, but we'll do this in the next chapter. For the moment, let's hard code this. If you have used one of the cloud providers, the URL can be obtained from the corresponding connection instructions. For the local installation, the URL will be `mongodb://localhost/issuetracker`. Note that the MongoDB Node.js driver accepts the database name as part of the URL itself, and it is best to specify it this way, even though a cloud provider may not show this explicitly.

Let's add the local installation URL to `trymongo.js` and a commented version of cloud providers' URLs.

```
...
const url = 'mongodb://localhost/issuetracker';

// Atlas URL - replace UUU with user, PPP with password, XXX with hostname
// const url = 'mongodb+srv://UUU:PPP@cluster0-XXX.mongodb.net/issuetracker?retryWrites=true';

// mLab URL - replace UUU with user, PPP with password, XXX with hostname
// const url = 'mongodb://UUU:PPP@XXX.mlab.com:33533/issuetracker';
...
```


Further, the client constructor takes in another argument with more settings for the client, one of which is whether to use the new style parser. Let's change the constructor to pass this also, to avoid a warning in the latest Node.js driver (version 3.1).

```
...
const client = new MongoClient(url, { useNewUrlParser: true });
...
```

The `connect()` method is an asynchronous method and needs a callback to receive the result of the connection. The callback takes in two arguments: an error and the result. The result is the client object itself. Within the callback, a connection to the database (as opposed a connection to the server) can be obtained by calling the `db` method of the `client` object. Thus, the callback and connection to the database can be written like this:

```
...
client.connect(function(err, client) {
  const db = client.db();
...
```

The connection to the database, `db`, is similar to the `db` variable we used in the mongo shell. In particular, it is the one that we can use to get a handle to a collection and its methods. Let's get a handle to the collection called `employees` that we were using in the previous section using the mongo shell.

```
...
  const collection = db.collection('employees');
...
```

With this collection, we can do the same things we did with the mongo shell's equivalent `db.employees` in the previous section. The methods are also very similar, except that they are all asynchronous. This means that the methods take in the regular arguments, but also a callback function that's called when the operation completes. The convention in the callback functions is to pass the error as the first argument and the result of the operation as the second argument. You already saw this pattern of callback in the previous connection method.

Let's insert a document and read it back to see how these methods work within the Node.js driver. The insertion can be written using the `insertOne` method, passing in an employee document and a callback. Within the callback, let's print the new `_id` that was created. Just as in the mongo shell `insertOne` command, the created ID is returned as part of the result object, in the property called `insertedId`.

```
...
  const employee = { id: 1, name: 'A. Callback', age: 23 };
  collection.insertOne(employee, function(err, result) {
    console.log('Result of insert:\n', result.insertedId);
...
```

Note that accessing the collection and the `insert` operation can only be called within the callback of the connection operation, because only then do we know that the connection has succeeded. There also needs to be some amount of error handling, but let's deal with this a little later.

Now, within the callback of the `insert` operation, let's read back the inserted document, using the ID of the result. We could use either the ID we supplied (`id`) or the auto-generated MongoDB ID (`_id`). Let's use `_id` just to make sure that we are able to use the result values.

```
...
    collection.find({ _id: result.insertedId})
      .toArray(function(err, docs) {
        console.log('Result of find:\n', docs);
      }
...
```

Now that we are done inserting and reading back the document, we can close the connection to the server. If we don't do this, the Node.js program will not exit, because the connection object is waiting to be used and listening to a socket.

```
...
      client.close();
...
```

Let's put all this together in a function called `testWithCallbacks()`. We will soon also use a different method of using the Node.js driver using `async/await`. Also, as is customary, let's pass a callback function to this function, which we will call from the `testWithCallbacks()` function once all the operations are completed. Then, if there are any errors, these can be passed to the callback function.

Let's first declare this function:

```
...
function testWithCallbacks(callback) {
  console.log('\n--- testWithCallbacks ---');
  ...
}
...
```

And within each callback as a result of each of the operations, on an error, we need to do the following:

- Close the connection to the server
- Call the callback
- Return from the call, so that no more operations are performed

We also need to do the same when all operations are completed. The pattern of the error handling is like this:

```
...
    if (err) {
      client.close();
      callback(err);
      return;
    }
...
```

Let's also introduce a call to the `testWithCallbacks()` function from the main section, supply it a callback to receive any error, and print it if any.

```
...
testWithCallbacks(function(err) {
  if (err) {
```



```
    console.log(err);
  }
});
...
```

With all the error handling and callbacks introduced, the final code in the `trymongo.js` file is shown in Listing 6-1.

■ **Note** Although no effort has been spared to ensure that all code listings are accurate, there may be typos or even corrections that did not make it to the book before it went to press. So, always rely on the GitHub repository (https://github.com/vasansr/pro-mern-stack-2) as the tested and up-to-date source for all code listings, especially if something does not work as expected.

Listing 6-1. trymongo.js: Using Node.js driver, Using the Callbacks Paradigm

```
const { MongoClient } = require('mongodb');

const url = 'mongodb://localhost/issuetracker';

// Atlas URL  - replace UUU with user, PPP with password, XXX with hostname
// const url = 'mongodb+srv://UUU:PPP@cluster0-XXX.mongodb.net/issuetracker?retryWrites=true';

// mLab URL - replace UUU with user, PPP with password, XXX with hostname
// const url = 'mongodb://UUU:PPP@XXX.mlab.com:33533/issuetracker';

function testWithCallbacks(callback) {
  console.log('\n--- testWithCallbacks ---');
  const client = new MongoClient(url, { useNewUrlParser: true });
  client.connect(function(err, client) {
    if (err) {
      callback(err);
      return;
    }
    console.log('Connected to MongoDB');

    const db = client.db();
    const collection = db.collection('employees');

    const employee = { id: 1, name: 'A. Callback', age: 23 };
    collection.insertOne(employee, function(err, result) {
      if (err) {
        client.close();
        callback(err);
        return;
      }
      console.log('Result of insert:\n', result.insertedId);
      collection.find({ _id: result.insertedId})
        .toArray(function(err, docs) {
```



```
        if (err) {
          client.close();
          callback(err);
          return;
        }
        console.log('Result of find:\n', docs);
        client.close();
        callback(err);
      });
    });
  });
}

testWithCallbacks(function(err) {
  if (err) {
    console.log(err);
  }
});
```

Let's clean up the collection before we test this. We could open another command shell, run the mongo shell in it, and execute `db.employees.remove({})`. But the mongo shell has a command line way of executing a simple command using the `--eval` command line option. Let's do that instead and pass the database name to connect to; otherwise, the command will be executed on the default database `test`. For the local installation, the command will look like this:

```
$ mongo issuetracker --eval "db.employees.remove({})"
```

If you are using a remote server from one of the hosting providers, instead of the database name, use the connection string including the database name as suggested by the hosting provider. For example, the Atlas command may look like this (replace the hostname, user, and password with your own):

```
$ mongo "mongodb+srv://cluster0-xxxxx.mongodb.net/issuetracker" --username atlasUser
--password atlasPassword --eval "db.employees.remove({})"
```

Now, we are ready to test the trial program we just created. It can be executed like this:

```
$ node scripts/trymongo.js
```

This should result in output like this (you will see a different `ObjectID`, otherwise the output should be the same):

```
--- testWithCallbacks ---
Connected to MongoDB
Result of insert:
 5bbef955580a2c313d4052f6
Result of find:
 [ { _id: 5bbef955580a2c313d4052f6,
    id: 1,
    name: 'A. Callback',
    age: 23 } ]
```


As you probably felt yourself, the callback paradigm is a bit unwieldy. But the advantage is that it works in the older JavaScript version (ES5), and therefore, older versions of Node.js. The callbacks are bit too deeply nested and the error handling makes for repetitive code. ES2015 started supporting Promises, which is supported by the Node.js MongoDB driver as well, and this was an improvement over callbacks. But in ES2017 and Node.js from version 7.6, full support for the async/await paradigm appeared, and this is the recommended and most convenient way to use the driver.

Let's implement another function called `testWithAsync()` within `trymongo.js` that uses the async/await paradigm. All asynchronous calls with a callback can now be replaced by a call to the same method, but without supplying a callback. Using `await` before the method call will simulate a synchronous call by *waiting* for the call to complete and return the results. For example, instead of passing a callback to the `connect()` method, we can just wait for it to complete like this:

```
...
    await client.connect();
...
```

Right in the next line, we can do whatever needs to be done after the operation is completed, in this case, get a connection to the database:

```
...
    await client.connect();
    const db = client.db();
...
```

The same pattern can be used for the other asynchronous calls, with one change: the result of the call, which was originally the second argument of the callback, can directly be assigned to a variable like a return value from the function call. So, the result of `insertOne()` can be captured like this:

```
...
    const result = await collection.insertOne(employee);
...
```

Errors will be thrown and can be caught. We can place all the operations in a single `try` block and catch any error in one place (the `catch` block) rather than after each call. There is no need for the function to take a callback, because if the caller needs to wait for the result, an `await` can be added before the call to this function, and errors can be thrown.

The new function using `await` before each of the operations—`connect()`, `insertOne()`, and `find()`—is shown in Listing 6-2.

Listing 6-2. trymongo.js, testWithAsync Function

```
async function testWithAsync() {
  console.log('\n--- testWithAsync ---');
  const client = new MongoClient(url, { useNewUrlParser: true });
  try {
    await client.connect();
    console.log('Connected to MongoDB');
    const db = client.db();
    const collection = db.collection('employees');

    const employee = { id: 2, name: 'B. Async', age: 16 };
    const result = await collection.insertOne(employee);
    console.log('Result of insert:\n', result.insertedId);
```

```
    const docs = await collection.find({ _id: result.insertedId })
      .toArray();
    console.log('Result of find:\n', docs);
  } catch(err) {
    console.log(err);
  } finally {
    client.close();
  }
}
```

Finally, let's modify the main part of the program to call `testWithAsync()` within the callback that handles the return value from `testWithCallbacks()`:

```
...
testWithCallbacks(function(err) {
  if (err) {
    console.log(err);
  }
  testWithAsync();
});
...
```

If you clear the collection using `remove()` as described previously and test these changes, you will see this result (the `ObjectID`s that you see will be different than the ones shown here):

```
--- testWithCallbacks ---
Connected to MongoDB
Result of insert:
 5bbf25dcf50e97340be0f01f
Result of find:
 [ { _id: 5bbf25dcf50e97340be0f01f,
    id: 1,
    name: 'A. Callback',
    age: 23 } ]

--- testWithAsync ---
Connected to MongoDB
Result of insert:
 5bbf25dcf50e97340be0f020
Result of find:
 [ { _id: 5bbf25dcf50e97340be0f020,
    id: 2,
    name: 'B. Async',
    age: 16 } ]
```

A good way to test whether errors are being caught and displayed is by running the program again. There will be errors because we have a unique index on the field `id`, so MongoDB will throw a duplicate key violation. If you have dropped the collection after creating the index, you could run the `createIndex()` command to reinstate this index.

As you can see, the async/await paradigm is much smaller in terms of code, as well as a lot clearer and easier to read. In fact, although we caught the error within this function, we didn't have to do it. We could as well have let the caller handle it.

Given the benefits of the async/await paradigm, let's use this in the Issue Tracker application when interacting with the database.

Schema Initialization

The mongo shell is not only an interactive shell, but is also a scripting environment. Using this, scripts can be written to perform various tasks such as schema initialization and migration. Because the mongo shell is in fact built on top of a JavaScript engine, the power of JavaScript is available in the scripts, just as in the shell itself.

One difference between the interactive and the non-interactive mode of working is that the non-interactive shell does not support non-JavaScript shortcuts, such as `use <db>` and `show collections` commands. The script has to be a regular JavaScript program adhering to the proper syntax.

Let's create a schema initialization script called `init.mongo.js` within the `script` directory. Since MongoDB does not enforce a schema, there is really no such thing as a schema initialization as you may do in relational databases, like creation of tables. The only thing that is really useful is the creation of indexes, which are one-time tasks. While we're at it, let's also initialize the database with some sample documents to ease testing. We will use the same database called `issuetracker` that we used to try out the mongo shell, to store all the collections relevant to the Issue Tracker application.

Let's copy the array of issues from `server.js` and use the same array to initialize the collection using `insertMany()` on a collection called `issues`. But before that, let's clear existing issues it by calling a `remove()` with an empty filter (which will match all documents) on the same collection. Then, let's create a few indexes on useful fields that we will be using to search the collection with.

Listing 6-3 shows the complete contents of the initialization script, `init.mongo.js`. There are comments in the beginning of the file that indicate how to run this script for different types of databases—local, Atlas, and mLab.

Listing 6-3. init.mongo.js: Schema Initialization

```
/*
 * Run using the mongo shell. For remote databases, ensure that the
 * connection string is supplied in the command line. For example:
 * localhost:
 *   mongo issuetracker scripts/init.mongo.js
 * Atlas:
 *   mongo mongodb+srv://user:pwd@xxx.mongodb.net/issuetracker ➥
     scripts/init.mongo.js
 * MLab:
 *   mongo mongodb://user:pwd@xxx.mlab.com:33533/issuetracker ➥
     scripts/init.mongo.js
 */

db.issues.remove({});

const issuesDB = [
  {
    id: 1, status: 'New', owner: 'Ravan', effort: 5,
    created: new Date('2019-01-15'), due: undefined,
    title: 'Error in console when clicking Add',
  },
```



```
  {
    id: 2, status: 'Assigned', owner: 'Eddie', effort: 14,
    created: new Date('2019-01-16'), due: new Date('2019-02-01'),
    title: 'Missing bottom border on panel',
  },
];

db.issues.insertMany(issuesDB);
const count = db.issues.count();
print('Inserted', count, 'issues');

db.issues.createIndex({ id: 1 }, { unique: true });
db.issues.createIndex({ status: 1 });
db.issues.createIndex({ owner: 1 });
db.issues.createIndex({ created: 1 });
```

You should be able to run this script using the mongo shell, with the name of the file as an argument in the command line, if you are using the local installation of MongoDB like this:

```
$ mongo issuetracker scripts/init.mongo.js
```

For the other methods of using MongoDB, there are instructions as comments on the top of the script. In essence, the entire connection string has to be specified in the command line, including the username and password that you use to connect to the hosted service. Following the connection string, you can type the name of the script, `scripts/init.mongo.js`.

You can run this any time you wish to reset the database to its pristine state. You should see an output that indicates that two issues were inserted, among other things such as the MongoDB version and the shell version. Note that creating an index when one already exists has no effect, so it is safe to create the index multiple times.

EXERCISE: SCHEMA INITIALIZATION

1. The same schema initialization could have been done using a Node.js script and the MongoDB driver. What are the pros and cons of each of these methods: using the mongo shell vs. the Node.js MongoDB driver?
2. Are there any other indexes that may be useful? Hint: What if we needed a search bar in the application? Read about MongoDB index types at https://docs.mongodb.com/manual/indexes/#index-types.

Answers are available at the end of the chapter.

Reading from MongoDB

In the previous section, you saw how to use the Node.js driver to perform basic CRUD tasks. With this knowledge, let's now change the List API to read from the MongoDB database rather than the in-memory array of issues in the server. Since we've initialized the database with the same initial set of issues, while testing, you should see the same set of issues in the UI.

In the trial that we did for the driver, we used the connection to the database in a sequence of operations and closed it. In the application, instead, we will maintain the connection so that we can reuse it for many operations, which will be triggered from within API calls. So, we'll need to store the connection to

the database in a global variable. Let's do that in addition to the `import` statement and other global variable declarations and call the global database connection variable `db`:

```
...
const url = 'mongodb://localhost/issuetracker';

// Atlas URL  - replace UUU with user, PPP with password, XXX with hostname
// const url = 'mongodb+srv://UUU:PPP@cluster0-XXX.mongodb.net/issuetracker?retryWrites=true';

// mLab URL - replace UUU with user, PPP with password, XXX with hostname
// const url = 'mongodb://UUU:PPP@XXX.mlab.com:33533/issuetracker';

let db;
...
```

Next, let's write a function to connect to the database, which initializes this global variable. This is a minor variation of what we did in `trymongo.js`. Let's not catch any errors in this function, instead, let the caller deal with them.

```
...
async function connectToDb() {
  const client = new MongoClient(url, { useNewUrlParser: true });
  await client.connect();
  console.log('Connected to MongoDB at', url);
  db = client.db();
}
...
```

Now, we have to change the setup of the server to first connect to the database and then start the Express application. Since `connectToDb()` is an async function, we can use `await` to wait for it to finish, then call `app.listen()`. But since `await` cannot be used in the main section of the program, we have to enclose it within an `async` function and execute that function immediately.

```
...
(async function () {
  await connectToDb();
  app.listen(3000, function () {
    console.log('App started on port 3000');
  });
})();
...
```

But we also have to deal with errors. So, let's enclose the contents of this anonymous function within a `try` block and print any errors on the console in the `catch` block:

```
...
(async function () {
  try {
    ...
  } catch (err) {
    console.log('ERROR:', err);
  }
})();
...
```

Now that we have a connection to the database set up in the global variable called db, we can use it in the List API resolver `issueList()` to retrieve a list of issues by calling the `find()` method on the `issues` collection. We need to return an array of issues from this function, so let's just use the `toArray()` function on the results of `find()` like this:

```
...
  const issues = await db.collection('issues').find({}).toArray();
...
```

The changes to `server.js` are shown in Listing 6-4.

Listing 6-4. server.js: Changes for Reading the Issue List from MongoDB

```
...
const { Kind } = require('graphql/language');
const { MongoClient } = require('mongodb');

const url = 'mongodb://localhost/issuetracker';

// Atlas URL  - replace UUU with user, PPP with password, XXX with hostname
// const url = 'mongodb+srv://UUU:PPP@cluster0-XXX.mongodb.net/issuetracker?retryWrites=true';

// mLab URL - replace UUU with user, PPP with password, XXX with hostname
// const url = 'mongodb://UUU:PPP@XXX.mlab.com:33533/issuetracker';

let db;

let aboutMessage = "Issue Tracker API v1.0";
...

async function issueList() {
  return issuesDB;
  const issues = await db.collection('issues').find({}).toArray();
  return issues;
}
...

async function connectToDb() {
  const client = new MongoClient(url, { useNewUrlParser: true });
  await client.connect();
  console.log('Connected to MongoDB at', url);
  db = client.db();
}

const server = new ApolloServer({
...

(async function () {
  try {
    await connectToDb();
```



```
    app.listen(3000, function () {
      console.log('App started on port 3000');
    });
  } catch (err) {
    console.log('ERROR:', err);
  }
})();
```

■ **Note** We did not have to do anything special due to the fact that the resolver `issueList()` is now an async function, which does not immediately return a value. The `graphql-tools` library handles this automatically. A resolver can return a value immediately or return a Promise (which is what an async function returns immediately). Both are acceptable return values for a resolver.

Since the issues from the database now contain an `_id` in addition to the `id` field, let's include that in the GraphQL schema of the type `Issue`. Otherwise, clients who call the API will not be able to access this field. Let's use `ID` as its GraphQL data type and make it mandatory. This change is shown in Listing 6-5.

Listing 6-5. schema.graphql: Changes to add _id as a Field in Issue

```
...
type Issue {
  _id: ID!
  id: Int!
  ...
}
...
```

Now, assuming that the server is still running (or that you have restarted the server and the compilation), if you refresh the browser, you will find that the two initial sets of issues are listed in a table, as before. The UI itself will show no change, but to convince yourself that the data is indeed coming from the database, you could modify the documents in the collection using the mongo shell and the `updateMany()` method on the collection. If, for example, you update effort to 100 for all the documents and refresh the browser, you should see that the effort is indeed showing 100 for all the rows in the table.

EXERCISE: READING FROM MONGODB

1. We are saving the connection in a global variable. What happens when the connection is lost? Stop the MongoDB server and start it again to see what happens. Does the connection still work?
2. Shut down the MongoDB server, wait for a minute or more, and then start the server again. Now, refresh the browser. What happens? Can you explain this? What if you wanted a longer period for the connection to work even if the database server is down? Hint: Look up the connection settings parameters at http://mongodb.github.io/node-mongodb-native/3.1/reference/connecting/connection-settings/.

3. We used `toArray()` to convert the list of issues into an array. What if the list is too big, say, a million documents? How would you deal with this? Hint: Look up the documentation for the MongoDB Node.js driver's `Cursor` at http://mongodb.github.io/node-mongodb-native/3.1/api/Cursor.html. Note that the `find()` method returns a `Cursor`.

Writing to MongoDB

In order to completely replace the in-memory database on the server, we'll also need to change the Create API to use the MongoDB database. As you saw in the MongoDB CRUD Operations section, the way to create a new document is to use the `insertOne()` method on the collection.

We used the size of the in-memory array to generate the new document's `id` field. We could do the same, using the `count()` method of the collection to get the next ID. But there is a small chance when there are multiple users using the application that a new document is created between the time we call the `count()` method and the time we call the `insertOne()` method. What we really need is a reliable way of generating a sequence of numbers that cannot give us duplicates, much like sequences in popular relational databases.

MongoDB does not provide such a method directly. But it does support an atomic update operation, which can return the result of the update. This method is called `findOneAndUpdate()`. Using this method, we can update a counter and return the updated value, but instead of using the `$set` operator, we can use the `$inc` operator, which increments the current value.

Let's first create a collection with the counter that holds a value for the latest Issue ID generated. To make it a bit generic, let's assume we may have other such counters and use a collection with an ID set to the name of the counter and a value field called `current` holding the current value of the counter. In the future, we could add more counters in the same collections, and these would translate to one document for each counter.

To start, let's modify the schema initialization script to include a collection called `counters` and populate that with one document for the counter for issues. Since there are insertions that create a few sample issues, we'll need to initialize the counter's value to the count of inserted documents. The changes are in `init.mongo.js`, and Listing 6-6 shows this file.

Listing 6-6. init.mongo.js: Initialize Counters for Issues

```
...
print('Inserted', count, 'issues');

db.counters.remove({ _id: 'issues' });
db.counters.insert({ _id: 'issues', current: count });
...
```

Let's run the schema initialization script again to make this change take effect:

```
$ mongo issuetracker scripts/init.mongo.js
```

Now, a call to `findOneAndUpdate()` that increments the `current` field is guaranteed to return a unique value that is next in the sequence. Let's create a function in `server.js` that does this, but in a generic manner. We'll let it take the ID of the counter and return the next sequence. In this function, all we have to do is call `findOneAndUpdate()`. It identifies the counter to use using the ID supplied, increments the field called `current`, and returns the new value. By default, the result of the `findOneAndUpdate()` method returns

the original document. To make it return the new, modified document instead, the option `returnOriginal` has to be set to `false`.

The arguments to the method `findOneAndUpdate()` are (a) the filter or match, for which we used `_id`, then (b) the update operation, for which we used a `$inc` operator with value `1`, and finally, (c) the options for the operation. Here's the code that will do the needful:

```
...
async function getNextSequence(name) {
  const result = await db.collection('counters').findOneAndUpdate(
    { _id: name },
    { $inc: { current: 1 } },
    { returnOriginal: false },
  );
  return result.value.current;
}
...
```

■ **Note** The option for returning the current or new value is called differently in the Node.js driver and in the mongo shell. In the mongo shell, the option is called `returnNewDocument` and the default is `false`. In the Node.js driver, the option is called `returnOriginal` and the default is `true`. In both cases, the default behavior is to return the original, so the option must be specified to return the new document.

Now, we can use this function to generate a new ID field and set it in the supplied `issue` object in the resolver `issueAdd()`. We can then write to the collection called `issues` using `insertOne()`, and then read back the newly created issue using `findOne()`.

```
...
  issue.id = await getNextSequence('issues');

  const result = await db.collection('issues').insertOne(issue);
  const savedIssue = await db.collection('issues')
    .findOne({ _id: result.insertedId });
  return savedIssue;
...
```

Finally, we can get rid of the in-memory array of issues in the server. Including this change, the complete set of changes in `server.js` is represented in Listing 6-7.

Listing 6-7. server.js: Changes for Create API to Use the Database

```
...
const issuesDB = [
  {
    id: 1, status: 'New', owner: 'Ravan', effort: 5,
    ...
  },
  ...
];
```

```
...
async function getNextSequence(name) {
  const result = await db.collection('counters').findOneAndUpdate(
    { _id: name },
    { $inc: { current: 1 } },
    { returnOriginal: false },
  );
  return result.value.current;
}

async function issueAdd(_, { issue }) {
  const errors = [];
  ...
  issue.created = new Date();

  issue.id = issuesDB.length + 1;
  issue.id = await getNextSequence('issues');

  issuesDB.push(issue);
  const result = await db.collection('issues').insertOne(issue);

  return issue;
  const savedIssue = await db.collection('issues')
    .findOne({ _id: result.insertedId });
  return savedIssue;
...
```

Testing this set of changes will show that new issues can be added, and even on a restart of the Node.js server, or the database server, the newly added issues are still there. As a cross-check, you could use the mongo shell to look at the contents of the collection after every change from the UI.

EXERCISE: WRITING TO MONGODB

1. Could we have just added the `_id` to the passed-in object and returned that instead of doing a `find()` for the inserted object?

Answers are available at the end of the chapter.

Summary

In this chapter, you learned about the installation and other ways of getting access to an instance of a database in MongoDB. You saw how to use the mongo shell and the Node.js driver to access the basic operations in MongoDB: the CRUD operations. We then modified the Issue Tracker application to use some of these methods to read and write to the MongoDB database, thus making the issue list persistent.

I covered only the very basics of MongoDB, only the capabilities and features that will be useful to build the Issue Tracker application, which is a rather simple CRUD application. In reality, the capabilities of the database as well as the Node.js driver and the mongo shell are vast, and many more features of MongoDB may be required for a complex application. I encourage you to take a look at the MongoDB documentation

(https://docs.mongodb.com/manual/) and the Node.js driver documentation (http://mongodb.github.io/node-mongodb-native/) to familiarize yourself with what else the database and the Node.js drivers are capable of.

Now that we have used the essentials of the MERN stack and have a working application, let's take a break from implementing features and get a bit organized instead. Before the application gets any bigger and becomes unwieldy, let's modularize the code and use tools to improve our productivity.

We'll do this in the next chapter, by using Webpack, one of the best tools that can be used to modularize both the front-end and the back-end code.

Answers to Exercises

Exercise: MongoDB Basics

1. As per the mongo shell documentation under "Access the mongo shell Help", you can find that there is a method called `help()` on many objects, including the `cursor` object. The way to get help on this is using `db.collection.find().help()`.

   But since this is also a JavaScript shell like Node.js, pressing Tab will auto-complete and a double-Tab will show a list of possible completions. Thus, if you assign a cursor to a variable and press Tab twice after typing the variable name and a dot after that, the shell will list the possible completions, and that is a list of methods available on the cursor.

Exercise: MongoDB CRUD Operations

1. This can be done using the `$exists` operator like this:

   ```
   > db.employees.find({ "name.middle": { $exists: true } })
   ```

2. The filter specification is not a JSON document, because it is not a string. It is a regular JavaScript object, which is why you are able to skip the quotes around the property names. You will also be able to have real `Date` objects as field values, unlike a JSON string.

3. The `$unset` operator in an update can be used to unset a field (which is actually different from setting it to null). Here is an example:

   ```
   > db.employees.update({{_id: ObjectId("57b1caea3475bb1784747ccb")},
   {"name.middle": {$unset: null}})
   ```

   Although we supplied `null` as the value for `$unset`, this value is ignored. It can be anything.

4. The `1` indicates an ascending sort order for traversing the index. `-1` is used to indicate a descending sort order. This is useful only for compound (aka composite) indexes, because a simple index on one field can be used to traverse the collection in both directions.

Exercise: Schema Initialization

1. The advantage of using the Node.js driver is that there is one way of doing things across the application and the scripts, and the familiarity will help prevent errors. But running the program requires a proper Node.js environment, including npm modules installed, whereas the mongo shell script can be run from anywhere, provided the machine has the mongo shell installed.

2. A search bar is quite helpful when searching for issues. A text index (an index based on the words) on the title field would be useful in this case. We'll implement a text index toward the end of the book.

Exercise: Reading from MongoDB

1. The connection object is in fact a connection pool. It automatically determines the best thing to do: reuse an existing TCP connection, reestablish a new connection when the connection is broken, etc. Using a global variable (at least, reusing the connection object) is the recommended usage.

2. If the database is unavailable for a short period (less than 30 seconds), the driver retries and reconnects when the database is available again. If the database is unavailable for a longer period, the read throws an error. The driver is also unable to reestablish a connection when the database is restored. The application server needs to be restarted in this case.

   The default interval of 30 seconds can be changed using the connection settings `reconnectTries` or `reconnectInterval`.

3. One option is to use `limit()` on the result to limit the return value to a maximum number of records. For example, `find().limit(100)` returns the first 100 documents. If you were to paginate the output in the UI, you could also use the `skip()` method to specify where to start the list.

   If, on the other hand, you think the client can handle large lists but you don't want to expend that much memory in the server, you could deal with one document at a time using `hasNext()` and `next()` and stream the results back to the client.

Exercise: Writing to MongoDB

1. Adding the `_id` and returning the object passed in would have worked, so long as you know for a fact that the write was a success and the object was written to the database as is. In most cases, this would be true, but it's good practice to get the results from the database, as that is the ultimate truth.