Module-03

Introduction to MERN

MERN is a **full-stack** JavaScript solution for developing web applications, meaning both the client-side (React) and server-side (Node.js, Express) are written in JavaScript, simplifying the development process. The rise of MERN is linked to the trend of building **Single Page Applications (SPAs)**, where data is dynamically loaded without reloading the entire page. React is a key part of this as it allows for dynamic updates to the user interface. MongoDB and Express work together to handle data and server-side logic.

The MERN stack is a popular collection of technologies used to build modern web applications. MERN stands for:

1. **MongoDB**: A NoSQL database that stores data in a flexible, JSON-like format, making it ideal for applications that require scalable and flexible data storage.
2. **Express.js**: A minimal and flexible Node.js web application framework that simplifies server-side development by providing essential features like routing and middleware to handle HTTP requests.
3. **React**: A JavaScript library for building user interfaces. Unlike full-fledged frameworks like Angular, React focuses on the "view" part of the MVC (Model-View-Controller) architecture. It allows developers to create interactive and dynamic user interfaces by breaking them down into reusable components.

4. **Node.js**: A JavaScript runtime built on Chrome's V8 JavaScript engine. It allows developers to run JavaScript on the server-side, enabling full-stack JavaScript development (both client and server-side).

MERN Components

Table 1-1. *Versions of Various Tools and Libraries*

Component	Major Version	Remarks
Node.js	10	This is an LTS (long term support) release
Express	4	--
MongoDB	3.6	Community Edition
React	16	--
React Router	4	--
React Bootstrap	0.32	There is no 1.0 release yet, things may break with 1.0
Bootstrap	3	This is compatible with React Bootstrap 0 (and 1 when released)
Webpack	4	--
ESLint	5	--
Babel	7	--

The **MERN stack** consists of four core components that work together to create modern web applications. Here are the key components in detail:

1. MongoDB (Database)

- **Type:** NoSQL database.
- **Role:** Stores data in flexible, JSON-like documents (BSON format).
- **Why MongoDB?**
  - It's scalable, fast, and schema-less, allowing dynamic data models.
  - Works well with JavaScript and JSON data structures.
  - MongoDB is often used in situations where the data doesn't fit neatly into tables, making it perfect for modern web apps with varied or complex data.

2. Express.js (Backend Framework)

- **Type:** Web application framework for Node.js.
- **Role:** Handles HTTP requests, routing, and middleware.
- **Why Express.js?**
  - Simplifies the setup and creation of server-side applications.
  - Manages API endpoints, sessions, cookies, authentication, and other web server-related functionality.
  - It acts as a middleware to interact with databases and the front-end efficiently.

3. React.js (Frontend Library)

- **Type:** Frontend JavaScript library for building user interfaces.
- **Role:** Renders dynamic and interactive UIs by breaking down the UI into reusable components.
- **Why React?**
  - Component-based structure allows for modular and maintainable code.
  - Virtual DOM for optimized performance and minimal UI updates.
  - React is declarative, meaning you describe what the UI should look like, and React takes care of updates when data changes.
  - It's commonly used to build **Single Page Applications (SPAs)**.

4. Node.js (JavaScript Runtime)

- **Type:** JavaScript runtime environment.
- **Role:** Runs JavaScript code server-side, enabling you to use JavaScript for both front-end and back-end code.
- **Why Node.js?**
  - Asynchronous, non-blocking, and event-driven, making it well-suited for real-time applications and high-performance tasks.
  - Runs JavaScript code outside the browser, which helps unify development for both front-end and back-end teams.

- The **npm** package manager makes it easy to integrate third-party libraries.

Server-Less Hello World

The **Server-less Hello World** is a simple example of how to use **React** and **ReactDOM** in a single HTML file to render a "Hello World!" message, all without the need for a server or complex setup.

Here's the process:

1. **HTML Structure**:
   You begin with a basic HTML file that includes the essential tags like <html>, <head>, and <body>.
2. **Including React Libraries**:
   In the <head> section, you include React and ReactDOM using <script> tags pointing to CDN URLs:
   - **React**: Manages the creation and handling of components and their states.
   - **ReactDOM**: Converts React components into actual DOM elements that can be rendered in the browser.
3. **Creating the React Element**:
   - Inside the <body>, you create a div element with an id="content", which will later hold your React components.
   - Using React.createElement(), you define a React element. In this case, a div with a title attribute containing an h1 heading with the text "Hello World!".

4. **Rendering the Element**:
    - Finally, you use ReactDOM.render() to render the React element inside the div with the ID content in the HTML page.
5. **Result**:

   When opened in a modern browser, this file will display the message "Hello World!" inside a div element, and you can hover over the div to see the tooltip showing its title.

Listing 2-1. index.html: Server-less Hello World

```
<!DOCTYPE HTML>
<html>

<head>
  <meta charset="utf-8">
  <title>Pro MERN Stack</title>

  <script src="https://unpkg.com/react@16/umd/react.development.js"></script>
  <script src="https://unpkg.com/react-dom@16/umd/react-dom.development.js"></script>
</head>

<body>
  <div id="contents"></div>

  <script>
    const element = React.createElement('div', {title: 'Outer div'},
      React.createElement('h1', null, 'Hello World!')
    );

    ReactDOM.render(element, document.getElementById('content'));
  </script>
</body>

</html>
```

You can test this file by opening it in a browser. It may take a few seconds to load the React libraries, but soon enough, you should see the browser displaying the caption, as seen in Figure 2-1. You should also be able to hover over the text or anywhere to its right side within the boundaries of the outer div, and you should be able to see the tooltip “Outer div” pop up

Figure 2-1. *A Hello World written in React*

React Components :

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)
- 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)

React Classes

In React, **class components** are a fundamental way to create reusable UI elements. These classes are built by extending React. Component, which provides essential functionality such as state management and lifecycle methods.

Key Features of React Classes

1. **Extends React.Component** – Every class component must extend React.Component to inherit its capabilities.
2. **Render Method** – The render() function is mandatory and must return a JSX element that React will display in the UI.
3. **State Management** – Class components can hold and update state using this.state.
4. **Lifecycle Methods** – Special methods like componentDidMount(), componentDidUpdate(), and componentWillUnmount() allow control over different stages of the component's existence.

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 {
  ...
}
...
```

Now, within this class, a render() method is needed, which should return an element. We'll use the same JSX 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
, an instance of the HelloWorld class can be created like this:

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

Now, this element can be used in place of the
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 and
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'));
```

Composing Components

React allows building UI using both **built-in** and **user-defined** components. **Component composition** helps break the UI into **smaller, independent pieces**, making it easier to develop, understand, and maintain.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)

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

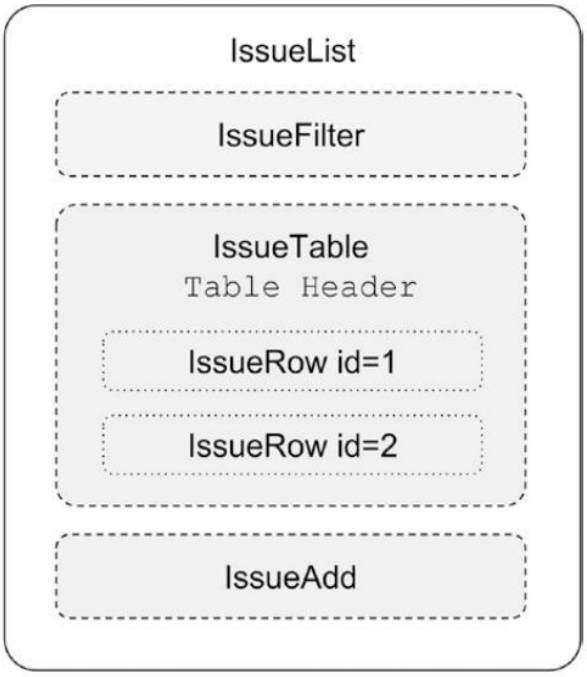

Figure 3-3. *Issue List UI hierarchy with issue rows*

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>
    );
  }
}
...
```

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. 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>
    );
  }
}
```

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.

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" />
...
```

Now, let's construct the IssueTable component, which is essentially a

, 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.

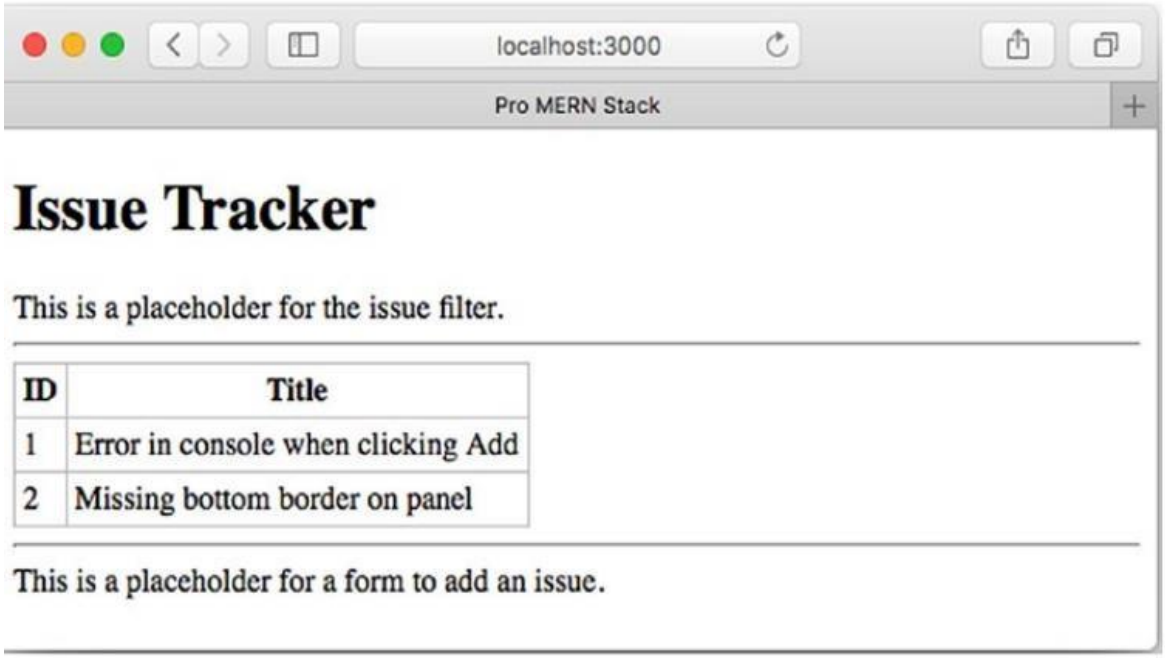

Figure 3-4. Passing data to child components

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.

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.

```
...
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 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
and includes an emphasized piece of text. This change is shown in Listing 3-5

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

```
...
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.

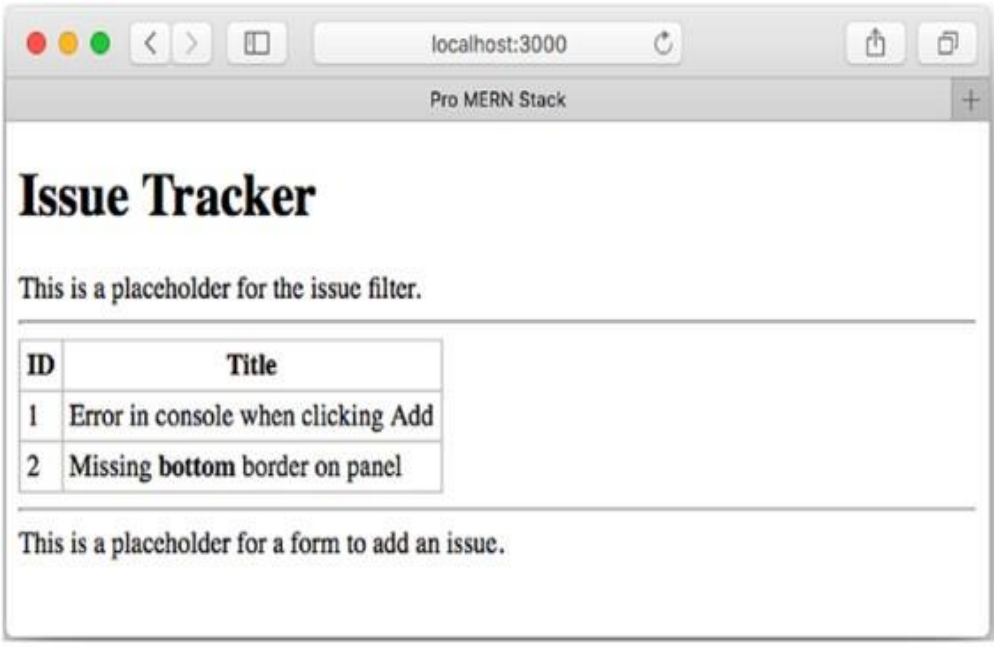

Figure 3-5. *Passing data to child components*

Dynamic Composition

- In this section, we'll replace our hard-coded set of Issue Row 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.

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.

```
...
  <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:

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.

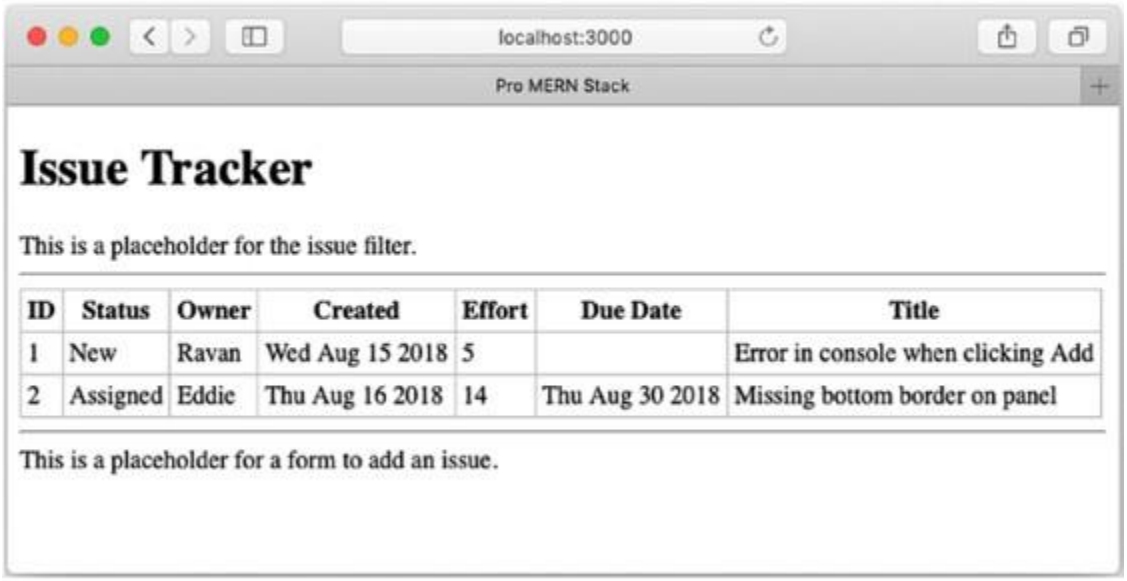

Figure 3-6. *Issue Rows constructed programmatically from an array*