Module-04

React State

Introduction to React State

1. **Static vs Dynamic Components**
   - Previously, we worked with static components that did not change.
   - To make components responsive, React uses **state**.
2. **What is State?**
   - State is a data structure in a React component that **changes over time**.
   - Unlike **props**, which are immutable, **state** is mutable and affects the UI.
3. **State and Rendering**
   - The UI updates automatically when the **state changes**.
   - React **re-renders** the component when the state is modified.
4. **Goal of the Chapter**
   - Add a button that appends a new row to the issue list on click.
   - Learn how to manipulate state, handle events, and pass data between components.
5. **Initial Implementation**
   - Start by adding a row using a **timer** instead of user interaction.
   - Later, replace the timer with a **button and a form** for user input.
6. **Defining State in a Component**
   - The **state** is stored in this.state as an object with key-value pairs.
   - Example:

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

   - **State should include only dynamic data** affecting the UI.
7. **Choosing What to Store in State**
   - Store data that **affects rendering** and **changes dynamically**.
   - Example: The list of issues should be stored in the state.
   - **Do not store** static values like table border styles.
8. **Modifying State in IssueTable Component**

- Store the initial issues in a variable called initialIssues.
- Update render() to use this.state.issues:

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

9. **Initializing State in Constructor**
   - The constructor sets the initial state:

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

10. **Next Steps**

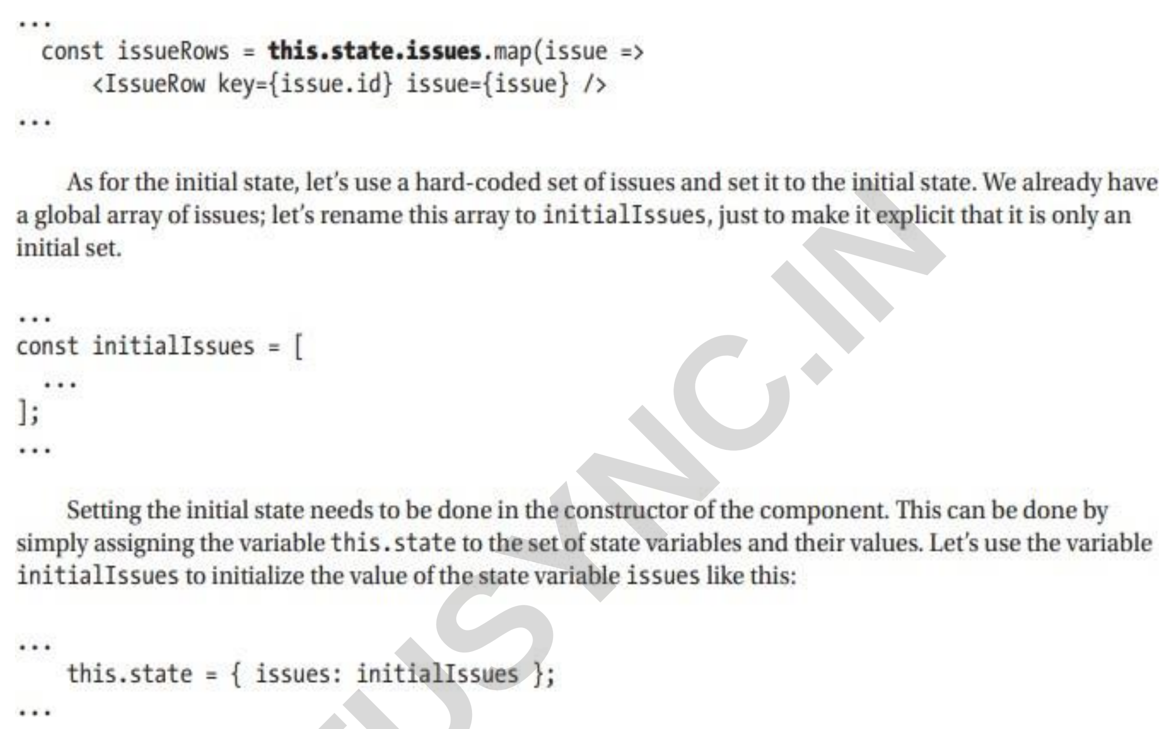
- Implement state updates using setState().
- Introduce user interaction with a button and form input

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 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 = [
  ...
];
...
```

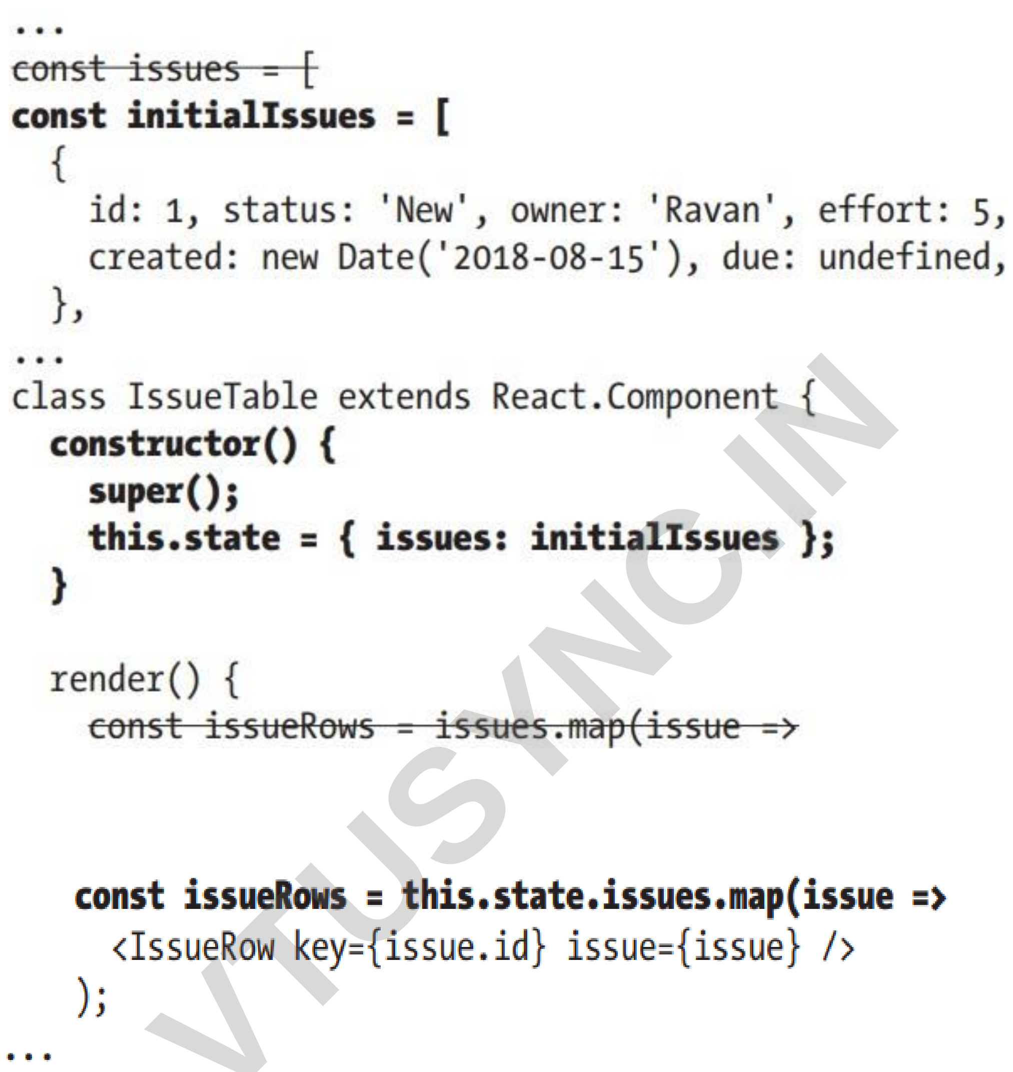
Setting the initial state needs to be done in the constructor of the component. This can be done by simply assigning the variable `this.state` to the set of state variables and their values. Let's use the variable `initialIssues` to initialize the value of the state variable `issues` like this:

```
...
    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} />
    );
...
```

Async State Initialization

1. State Initialization in the Constructor

- React **state must be assigned** in the constructor.
- Since data is fetched asynchronously, initialize issues as an **empty array**:

  constructor() {

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

2. Modifying State with setState()

- State updates must be done using this.setState().
- Example of updating the issues state dynamically:

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

- setState() **merges** the new state with the existing state.

3. Why Is an Async Call Needed for Fetching Data?

- Data is **usually fetched from a server**, not statically available.
- Fetching requires an **asynchronous API call**.
- We simulate this with setTimeout().

4. Simulating an API Call with setTimeout()

- Mimics real-world API call behavior:

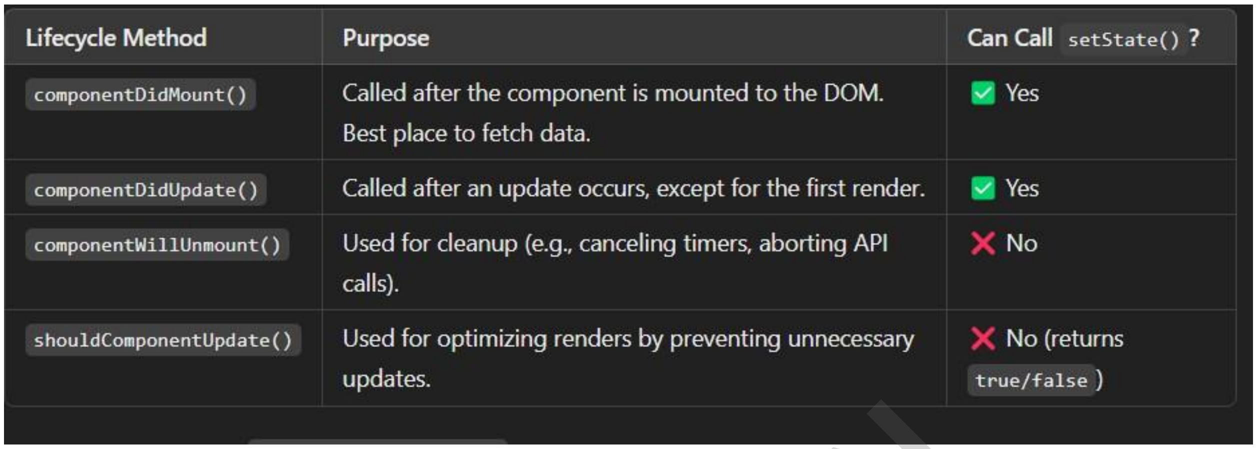
```
loadData() {
  setTimeout(() => {
    this.setState({ issues: initialIssues });
  }, 500);
}
```

- The **500ms delay** represents an API response time.

5. Why Not Call loadData() in the Constructor?

- The **constructor only initializes** the component, but the UI rendering happens later.
- Calling setState() inside the constructor **may cause errors** if rendering is not finished.

6. React Lifecycle Methods for Managing State Updates



Lifecycle Method	Purpose	Can Call `setState()` ?
`componentDidMount()`	Called after the component is mounted to the DOM. Best place to fetch data.	✅ Yes
`componentDidUpdate()`	Called after an update occurs, except for the first render.	✅ Yes
`componentWillUnmount()`	Used for cleanup (e.g., canceling timers, aborting API calls).	❌ No
`shouldComponentUpdate()`	Used for optimizing renders by preventing unnecessary updates.	❌ No (returns `true/false`)

7. Fetching Data in componentDidMount()

- Ensures the **component is fully rendered** before making an API call.
- Implementation:

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

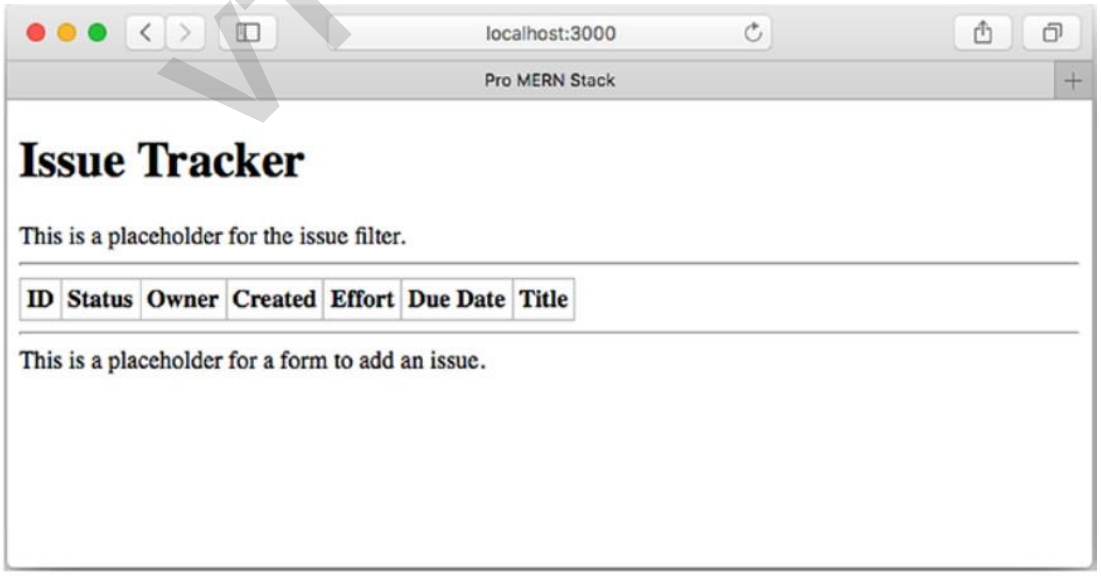
8. Key Takeaways

- **State must be initialized in the constructor but updated with setState().**
- **Data fetching is asynchronous** and should be handled in lifecycle methods.
- **Use componentDidMount()** to safely fetch and update state.
- **Lifecycle methods** help manage state and optimize rendering.

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



Issue Tracker

This is a placeholder for the issue filter.

ID	Status	Owner	Created	Effort	Due Date	Title

This is a placeholder for a form to add an issue.

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

Updating State

1. Changing a Portion of the State Instead of Overwriting It

- Instead of replacing the whole state, we update just part of it.
- Example: **Adding a new issue** to the existing issues array.

2. Creating a Method to Add a New Issue

- The method assigns an **ID** and **creation date** before adding the issue:

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

3. Directly Modifying State is NOT Allowed

- The state must be treated as **immutable**.
- These **incorrect** examples modify state directly:

```
this.state.issues.push(issue); // + Incorrect
issues = this.state.issues;
issues.push(issue);
this.setState({ issues: issues }); // + Incorrect
```

4. Why Can't We Modify State Directly?

- React **does not detect direct mutations** of the state.
- Lifecycle methods that compare **previous and new states** may fail.
- Using setState() ensures React detects changes and **rerenders** correctly.

5. Correct Way: Using a Copy of the State

- Create a **shallow copy** of the array using slice():

```
const issues = this.state.issues.slice();
issues.push(issue);
this.setState({ issues: issues });
```

- This ensures React recognizes a **new reference** and updates the component properly.

6. Alternative: Using Immutability Libraries

- **Libraries like Immutable.js** help manage deep state updates efficiently.
- **Not needed for simple cases** like appending an issue.

7. Simulating an Automatic Issue Addition

- Instead of a UI, we **automatically add** an issue after 2 seconds using setTimeout().

8. Define a Sample Issue

- Declare a **predefined issue**:

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

9. Adding the Sample Issue with a Timer

- In constructor(), schedule an automatic addition of the issue:

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

10. Key Takeaways

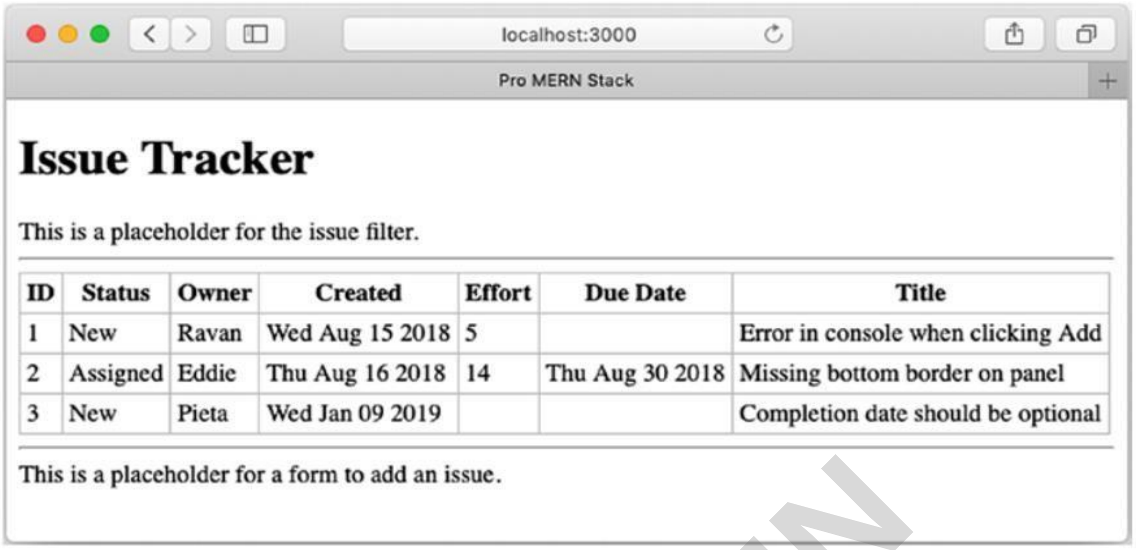
Never modify state directly; always use setState().
Make a copy of the state before updating it.
Use lifecycle methods to manage state updates.
For complex state updates, consider using immutability libraries.
State updates trigger React rerenders, ensuring UI consistency.

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



Issue Tracker

This is a placeholder for the issue filter.

ID	Status	Owner	Created	Effort	Due Date	Title
1	New	Ravan	Wed Aug 15 2018	5		Error in console when clicking Add
2	Assigned	Eddie	Thu Aug 16 2018	14	Thu Aug 30 2018	Missing bottom border on panel
3	New	Pieta	Wed Jan 09 2019			Completion date should be optional

This is a placeholder for a form to add an issue.

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.

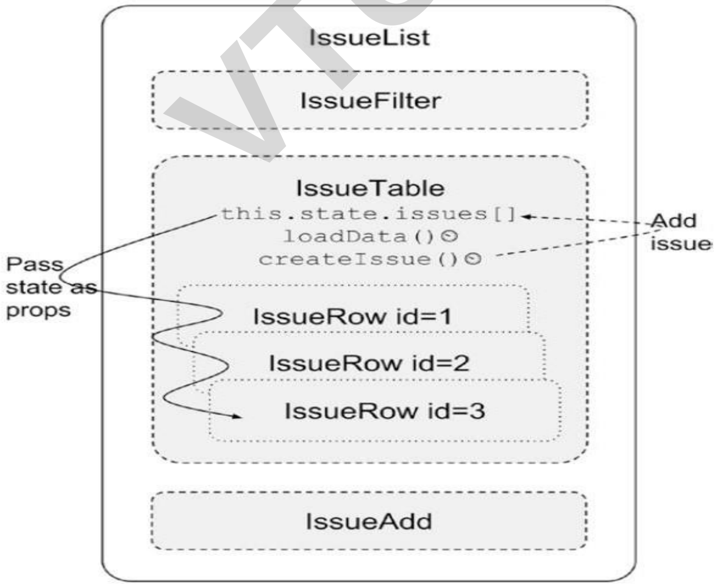


Figure 4-3. Setting state and passing data as props

Lifting State Up

In React, **Lifting State Up** is a pattern used to manage shared state between multiple components. Instead of keeping state separately in sibling components, we move it to their **closest common parent**. This allows child components to receive the shared state as **props** and update it via **callback functions** provided by the parent.

Why Lift State Up?

- **Ensures Sibling Communication:** React does not allow direct communication between sibling components. Instead, the parent can hold the state and pass it down.
- **Centralized State Management:** Keeping state in a common parent prevents data duplication and inconsistencies.
- **Easier Updates:** Since all state changes happen in the parent, updating and debugging become simpler.

Example (Issue Tracker Scenario)

- Initially, IssueTable managed the issues list and createIssue() method.
- To allow IssueAdd to trigger issue creation, **state and methods were moved to IssueList (the common parent).**
- IssueTable now receives issues as **props** instead of managing its own state.
- IssueAdd calls createIssue() through a **prop function** from IssueList.

Key Changes in Code

1. **Move State and Methods to Parent (IssueList)**

```
class IssueList extends React.Component {
  constructor() {
    super();
    this.state = { issues: [] };
    this.createIssue = this.createIssue.bind(this);
  }
  createIssue(issue) {
    issue.id = this.state.issues.length + 1;
    issue.created = new Date();
    const newIssueList = [...this.state.issues, issue];
    this.setState({ issues: newIssueList });
  }
  render() {
    return (
      <>
        <IssueTable issues={this.state.issues} />
        <IssueAdd createIssue={this.createIssue} />
      </>
    );
  }
}
```

2. **Pass State Down as Props (IssueTable)**

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

3. **Pass Method as Props (IssueAdd)**

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

4. **Use the Passed Method in IssueAdd**

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

This setup allows IssueAdd to trigger a new issue addition without directly modifying the state. Instead, it calls createIssue() in IssueList, ensuring state updates in a controlled way.

Event Handling

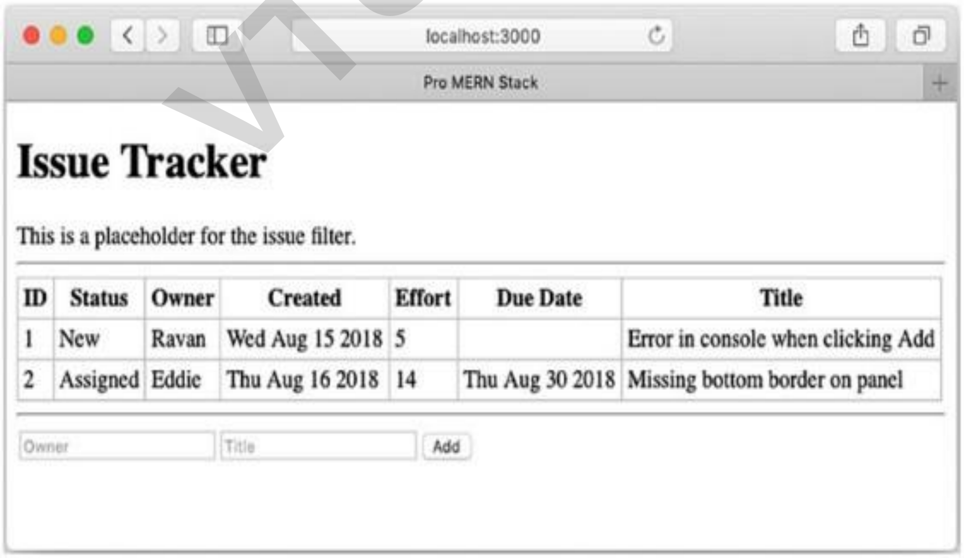
Let's now add an issue interactively, on the click of a button rather than use a timer to do this. We'll create a form with two text inputs and use the values that the user enters in them to add a new issue. An Add button will trigger the addition. Let's start by creating the form with two text inputs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



Issue Tracker

This is a placeholder for the issue filter.

ID	Status	Owner	Created	Effort	Due Date	Title
1	New	Ravan	Wed Aug 15 2018	5		Error in console when clicking Add
2	Assigned	Eddie	Thu Aug 16 2018	14	Thu Aug 30 2018	Missing bottom border on panel

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.

So, let's rewrite the form declaration with a name and an on Submit 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.
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 = "";
  }
...
```

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. 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.

```
...
  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

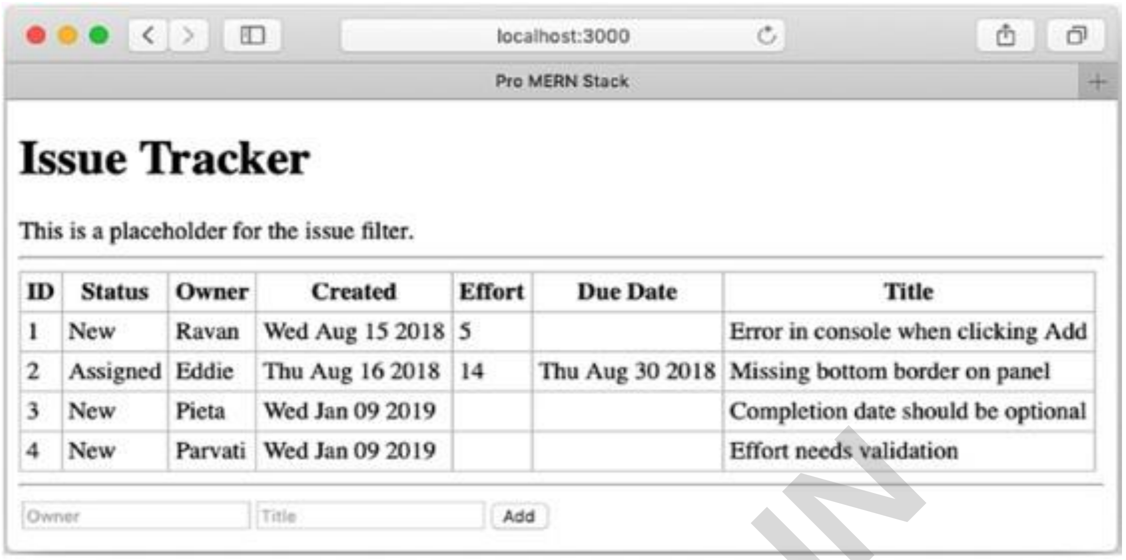
```
class IssueAdd extends React.Component {
  constructor() {
    super();
    this.handleSubmit = this.handleSubmit.bind(this);
  }

  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



Issue Tracker

This is a placeholder for the issue filter.

ID	Status	Owner	Created	Effort	Due Date	Title
1	New	Ravan	Wed Aug 15 2018	5		Error in console when clicking Add
2	Assigned	Eddie	Thu Aug 16 2018	14	Thu Aug 30 2018	Missing bottom border on panel
3	New	Pieta	Wed Jan 09 2019			Completion date should be optional
4	New	Parvati	Wed Jan 09 2019			Effort needs validation

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

many rows as you like. If you add two issues, you'll get a screen like the one in Figure 4-5.

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. This new UI hierarchy data and function flow is depicted in Figure 4-6.

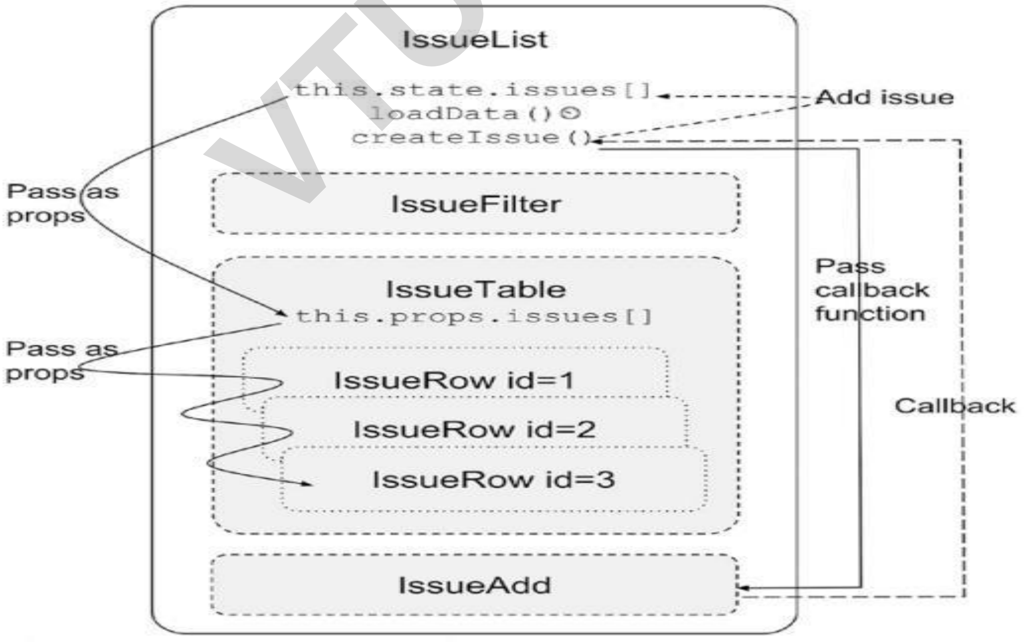


Figure 4-6. Component hierarchy and data flow after lifting state up

Stateless Components

Stateless components, also called **functional components**, are components that only receive props and render UI without maintaining any internal state.

Why Use Stateless Components?

- **Improved Performance:** They are faster because they don't have lifecycle methods or state updates.
- **Cleaner Code:** They are simpler and easier to read.
- **Better Maintainability:** They focus only on rendering, making debugging easier.

Example: Converting Class Components to Stateless Components

Before (Class Component)

```
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 (Stateless Functional Component)

```
const IssueRow = ({ issue }) => (
  <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>
);
```

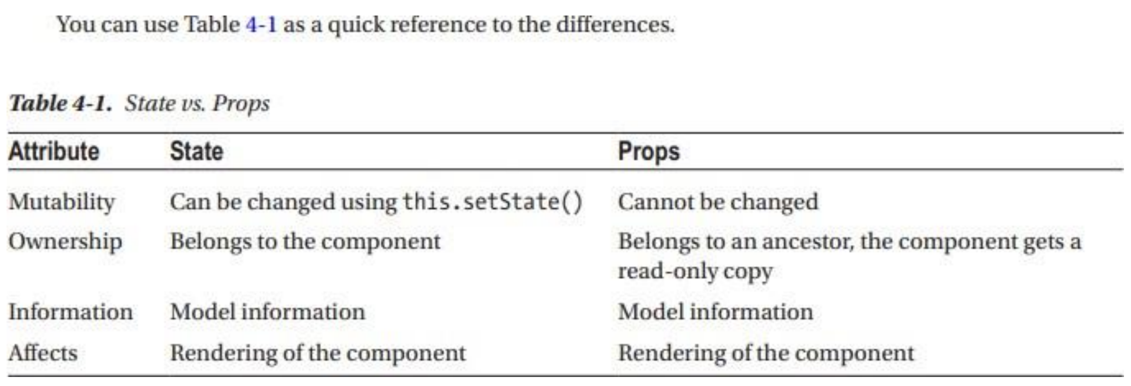
Designing Components

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. Prop

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.



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

Express

1. What is Express?
Express.js is a **minimal** and **flexible** web framework for Node.js. It provides essential web functionalities through **middleware** and enables efficient routing.

2. Routing in Express

Routing directs incoming HTTP requests to the appropriate handler.

Basic Route Example:

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

- The **HTTP method** (GET) and **path** (/hello) define the route.
- The **handler function** processes the request and sends a response.

Route Parameters:

Used to capture dynamic values from URLs.

```
app.get('/customers/:customerId', (req, res) => {
  res.send(`Customer ID: ${req.params.customerId}`);
});
```

- /customers/1234 → req.params.customerId = 1234

3. Express Request Matching and Order

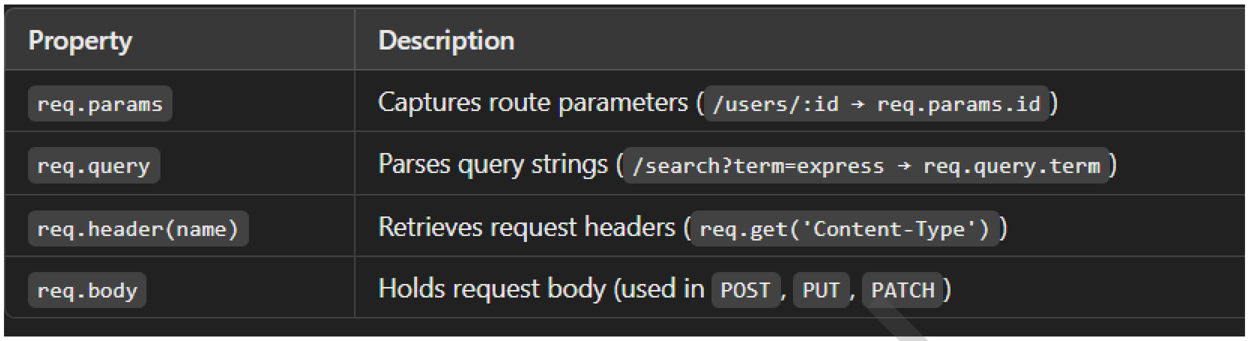
- **Routes are matched in order** of definition.
- More specific routes should be defined **before** generic ones.

Example:

```
app.get('/api/issues', handler);  // Specific
app.use('/api/*', middleware);   // Generic (placed after)
```

4. Express Request Object (req)

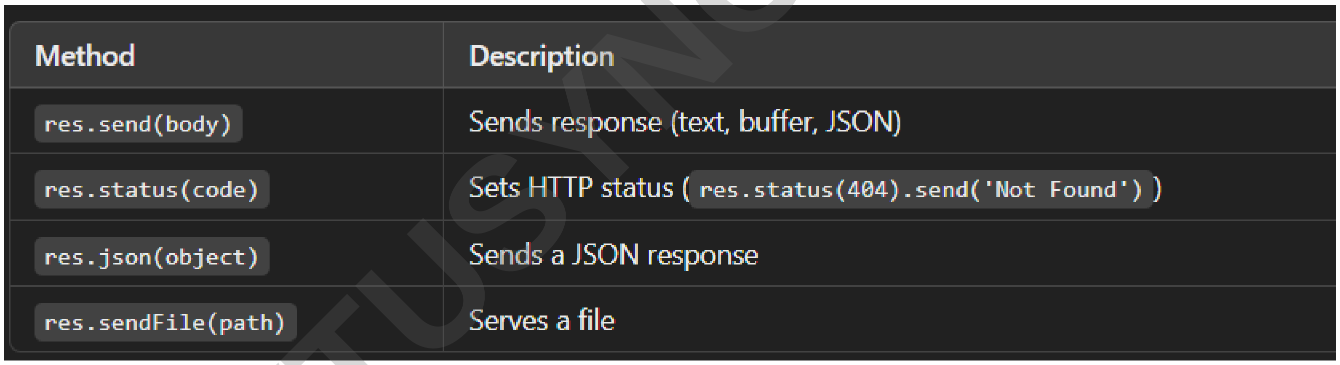
Holds information about the HTTP request.



Property	Description
`req.params`	Captures route parameters (`/users/:id → req.params.id`)
`req.query`	Parses query strings (`/search?term=express → req.query.term`)
`req.header(name)`	Retrieves request headers (`req.get('Content-Type')`)
`req.body`	Holds request body (used in `POST`, `PUT`, `PATCH`)

5. Express Response Object (res)

Used to send responses back to the client.



Method	Description
`res.send(body)`	Sends response (text, buffer, JSON)
`res.status(code)`	Sets HTTP status (`res.status(404).send('Not Found')`)
`res.json(object)`	Sends a JSON response
`res.sendFile(path)`	Serves a file

6. Middleware in Express

Middleware functions process requests before sending a response.

Application-Level Middleware:

```
app.use((req, res, next) => {
  console.log('Request received');
  next(); // Pass control to next middleware
});
```

Built-in Middleware:

- **express.static** → Serves static files (CSS, images, HTML).
- **Example:**

app.use(express.static('public'));

- Access files via /index.html, /style.css.

Path-Specific Middleware:

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

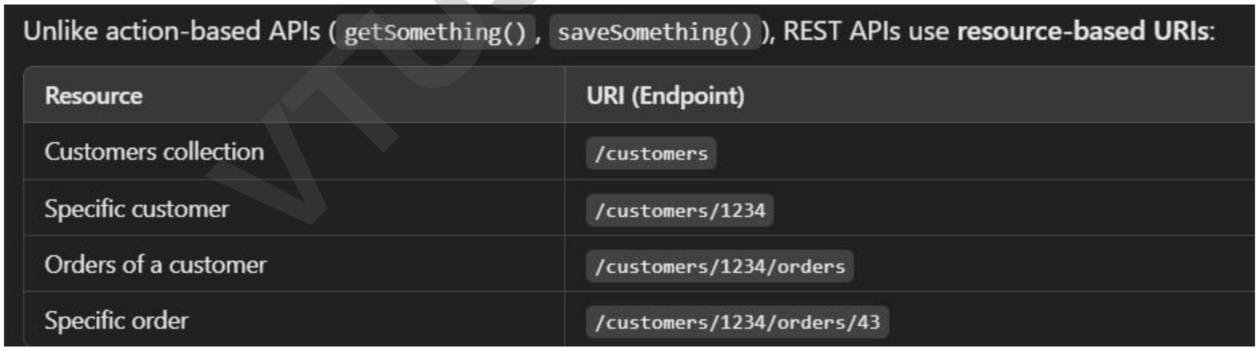
- Static files must be accessed via /public/index.html.

REST API

1. What is REST?

REST (**Representational State Transfer**) is an **architectural pattern** for designing web APIs. It focuses on:
Resources (nouns, such as customers or orders)
HTTP methods (verbs like GET, POST, PUT, DELETE)
Stateless communication (each request contains all required information)

2. REST is Resource-Based



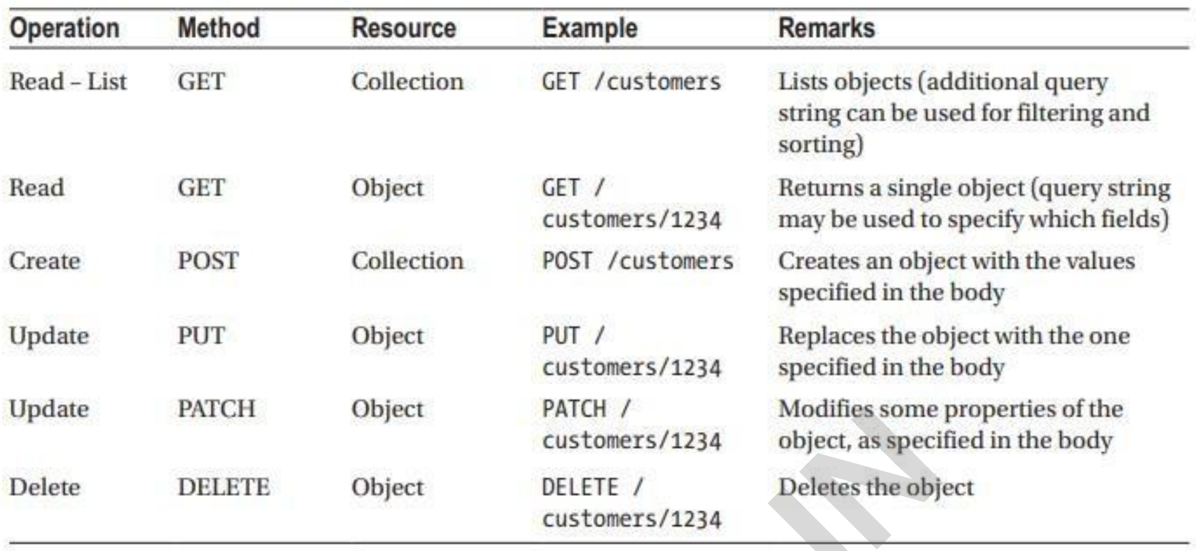
Unlike action-based APIs (`getSomething()`, `saveSomething()`), REST APIs use **resource-based URIs**:

Resource	URI (Endpoint)
Customers collection	`/customers`
Specific customer	`/customers/1234`
Orders of a customer	`/customers/1234/orders`
Specific order	`/customers/1234/orders/43`

3. **HTTP Methods as Actions**

HTTP methods represent **CRUD** (Create, Read, Update, Delete) operations:

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

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.

GraphQL

While REST follows a structured API design with multiple endpoints, GraphQL introduces a **single endpoint** that enables clients to request **only the data they need**.

Shortcomings of REST APIs:

1. **Over-fetching**: REST APIs return **more data than necessary** because predefined endpoints deliver full objects.
2. **Under-fetching**: Clients often need to make **multiple API requests** to get related data.
3. **Versioning Issues**: Adding new fields often requires **creating a new API version**.
4. **Multiple Endpoints**: REST requires **separate endpoints for different resources**, making API management more complex.

Key Features of GraphQL

Field Specification (Fine-Grained Data Control)

- In REST, the API response **includes all fields** of an object.
- In GraphQL, the client **must specify** the required fields.
- This reduces **network load**, making it efficient for mobile and web applications.

Example GraphQL Query:

```
{
  user(id: "1234") {
    name
    email
  }
}
```

Response (only requested fields are returned):

```
{
  "data": {
    "user": {
      "name": "Alice",
      "email": "alice@example.com"
    }
  }
}
```

Graph-Based Data Model (Natural Relationship Handling)

- REST is **resource-based**, treating each entity as a separate resource.
- GraphQL is **graph-based**, **naturally supporting relationships** between objects.

Example: Fetch a user and their assigned issues in **one** request:

```
{
  user(id: "1") {
    name
    issues {
      title
      status
    }
  }
}
```

Single Endpoint (Efficient API Structure)

- **REST APIs** require multiple endpoints (/users, /orders, /products).
- **GraphQL APIs** use a **single endpoint** (e.g., /graphql).
- The request structure (query) defines which data should be returned.

Strongly Typed Schema (Error Prevention & Data Validation)

GraphQL enforces **strong data types** using a schema language.

- Every field and argument has a **defined type**.
- This ensures **valid data queries** and provides **descriptive error messages**.

Example GraphQL Schema:

```
type User {
  id: ID!
  name: String!
  email: String!
}
```

Introspection (Self-Documenting APIs)

GraphQL servers **can describe their own schema** dynamically.

- Developers can explore available queries and mutations **without external documentation**.
- Tools like **Apollo Playground** allow real-time API exploration and testing.

Libraries

- Parsing and dealing with the type system language (also called the GraphQL Schema Language) as well as the query language is hard to do on your own.
- Fortunately, there are tools and libraries available in most languages for this purpose.
- For JavaScript on the back-end, there is a reference implementation of GraphQL called GraphQL.js. To tie this to Express and enable HTTP requests to be the transport mechanism for the API calls, there is a package called express-graphql.
- But these are very basic tools that lack some advanced support such as modularized schemas and seamless handling of custom scalar types.
- The package graphql-tools and the related apollo-server are built on top of GraphQL.js to add these advanced features. We will be using the advanced packages for the Issue Tracker application in this chapter.
- I will cover only those features of GraphQL that are needed for the purpose of the application.
- For advanced features that you may need in your own specific application, do refer to the complete documentation of GraphQL at https://graphql.org and the tools at https://www.apollographql.com/ docs/graphql-tools/.

The About API

1. **Installing Required Packages**

To get started, you need the following **npm** packages:

- **graphql**: The base GraphQL package.
- **apollo-server-express**: Apollo Server middleware for Express.js.

Run the following command to install these:

```
npm install graphql@0 apollo-server-express@2
```

2. **Defining the GraphQL Schema**

GraphQL uses a schema to define the structure of data and operations.

- **Query** type: Defines read operations.
- **Mutation** type: Defines write (update/create/delete) operations.

Example Schema:

```
const typeDefs = `
 type Query {
   about: String!  // Read operation
 }
 type Mutation {
   setAboutMessage(message: String!): String  // Write operation
 }
`;
```

- about: A **mandatory** (!) string field that provides the about message.
- setAboutMessage(message: String!): A mutation that updates the about message.

3. **Creating Resolvers**

Resolvers are functions that define how each GraphQL field should behave.

Example Resolvers:

```
let aboutMessage = "Issue Tracker API v1.0";  // Initial message

const resolvers = {
 Query: {
   about: () => aboutMessage,  // Returns the message
 },
 Mutation: {
   setAboutMessage(_, { message }) {
    return (aboutMessage = message);  // Updates the message
   },
 },
};
```

- **Query Resolver**:
  - about: Returns the aboutMessage string.
- **Mutation Resolver**:
  - setAboutMessage: Updates aboutMessage with the new value.

4. **Setting Up Apollo Server**

Apollo Server provides an easy way to integrate GraphQL with Express.

```
const { ApolloServer } = require('apollo-server-express');

const server = new ApolloServer({
 typeDefs,
 resolvers,
});
```

This creates a GraphQL server instance using the schema (typeDefs) and resolvers.

5. **Integrating with Express.js**

To run the GraphQL server, we integrate it with an Express application:

```
const express = require('express');
const app = express();

app.use(express.static('public'));  // Serve static files

server.applyMiddleware({ app, path: '/graphql' });  // Mount GraphQL endpoint

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

- **Express** is used to create the HTTP server.
- **GraphQL API is exposed at** /graphql.
- **Server runs on** http://localhost:3000.

6. **Using GraphQL Playground**

Apollo Server provides a built-in **GraphQL Playground**, accessible at:

http://localhost:3000/graphql

This tool allows you to:

- Explore the schema.
- Run queries and mutations.
- View introspection details.

7. **Testing the API**

You can use GraphQL Playground to run the following queries:

1. Read the about message:

```
query {
  about
}
```

Response:

```
{

  "data": {
    "about": "Issue Tracker API v1.0"
  }
}
```

2. Update the about message:

```
mutation {
  setAboutMessage(message: "New API Version")
}
```

Response:

```
{
  "data": {
    "setAboutMessage": "New API Version"
  }
}
```

3. Verify the update:

```
query {
  about
}
```

Response:

```
{
  "data": {
    "about": "New API Version"
  }
}
```

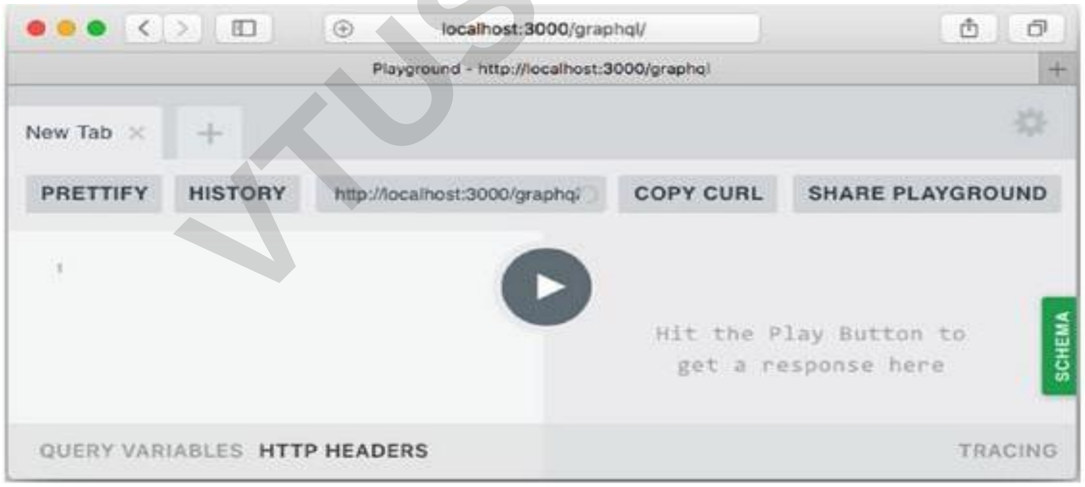

localhost:3000/graphql/
Playground - http://localhost:3000/graphql
New Tab
PRETTIFY
HISTORY
http://localhost:3000/graphql
COPY CURL
SHARE PLAYGROUND
Hit the Play Button to get a response here
SCHEMA
QUERY VARIABLES
HTTP HEADERS
TRACING

Figure 5-1. The GraphQL Playground

GraphQL Schema File

This is a great approach for keeping your GraphQL schema manageable as it grows! Extracting the schema into a separate .graphql file not only makes your code more readable but also improves maintainability.

Key Takeaways:

1. **Separation of Concerns:** Moving the schema to a .graphql file keeps the server.js file clean and modular.
2. **File Reading:** The fs.readFileSync function reads the schema file into a string for ApolloServer.
3. Nodemon Watch: **Since nodemon only watches .js files by default, adding -**e js,graphql ensures it also watches .graphql files.

Potential Enhancements:

- **Use fs.promises.readFile for async operations:**
  Instead of readFileSync, you can use fs.promises.readFile with await for a non-blocking approach:

  ```
  const fs = require('fs').promises;
  const typeDefs = await fs.readFile('./server/schema.graphql', 'utf-8');
  ```

- **Organizing the Project:**
  - Place your schema in a graphql/ folder (server/graphql/schema.graphql).
  - Structure resolvers in a separate file (server/graphql/resolvers.js).

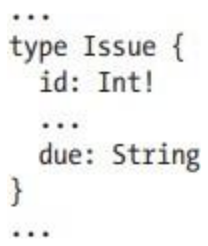
The List API

1. Define the GraphQL Schema (schema.graphql)

- You introduce a **custom type** Issue that represents an issue object.
- Since GraphQL doesn't have a built-in **Date** type, you use **String** for created and due.
- Add a new **Query field** issueList to return an **array of issues** ([Issue!]! ensures a non-nullable array with non-nullable elements).

Now that you have learned the basics of GraphQL, let’s make some progress toward building the Issue Tracker application using this knowledge. The next thing we’ll do is implement an API to fetch a list of issues. We’ll test it using the Playground and, in the next section, we’ll change the front-end to integrate with this new API.

Let’s start by modifying the schema to define a custom type called Issue. It should contain all the fields of the issue object that we have been using up to now. But since there is no scalar type to denote a date in GraphQL, let’s use a string type for the time being. We’ll implement custom scalar types later in this chapter. So, the type will have integers and strings, some of which are optional. Here’s the partial schema code for the new type.



```
...
type Issue {
  id: Int!
  ...
  due: String
}
...
```

Now, let’s add a new field under Query to return a list of issues. The GraphQL way to specify a list of another type is to enclose it within square brackets. We could use [Issue] as the type for the field, which we will call issueList.

But we need to say not only that the return value is mandatory, but also that each element in the list cannot be null. So, we have to add the exclamation mark after Issue as well as after the array type, as in [Issue!]!.

Let’s also separate the top-level Query and Mutation definitions from the custom types using a comment. The way to add comments in the schema is using the # character at the beginning of a line. All these changes are listed in Listing 5-5

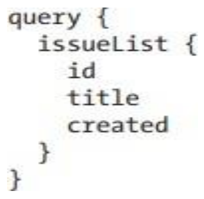
Listing 5-5. schema.graphql: Changes to Include Field issueList and New Issue Type

```
type Issue {
  id: Int!
  title: String!
  status: String!
  owner: String
  effort: Int
  created: String!
  due: String
}

##### Top level declarations

type Query {
  about: String!
  issueList: [Issue!]!
}

type Mutation {
  setAboutMessage(message: String!): String
}
```

Listing 5-6. server.js: Changes for issueList Query Field

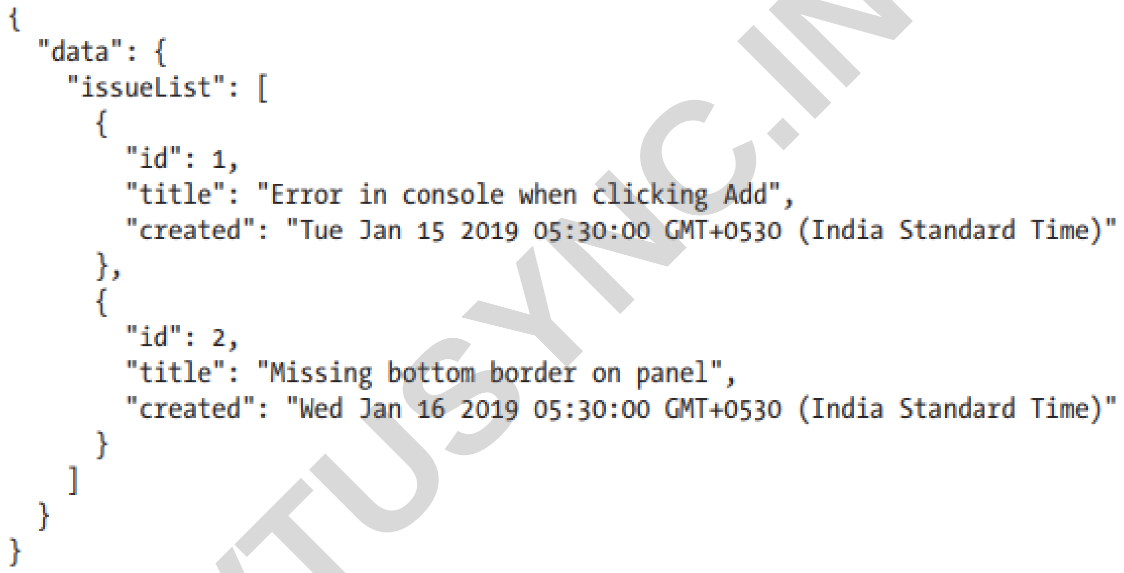
```
...
let aboutMessage = "Issue Tracker API v1.0";

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',
  },
];

const resolvers = {
  Query: {
    about: () => aboutMessage,
    issueList,
  },
  Mutation: {
    setAboutMessage,
  },
};

function setAboutMessage(_, { message }) {
  return aboutMessage = message;
}

function issueList() {
  return issuesDB;
}
...
```

To test this in the Playground, you will need to run a query that specifies the issueList field, with subfields. But first, a refresh of the browser is needed so that the Playground has the latest schema and doesn't show errors when you type the query.



```
query {
  issueList {
    id
    title
    created
  }
}
```

This query will result in an output like this.



```
{
  "data": {
    "issueList": [
      {
        "id": 1,
        "title": "Error in console when clicking Add",
        "created": "Tue Jan 15 2019 05:30:00 GMT+0530 (India Standard Time)"
      },
      {
        "id": 2,
        "title": "Missing bottom border on panel",
        "created": "Wed Jan 16 2019 05:30:00 GMT+0530 (India Standard Time)"
      }
    ]
  }
}
```

.

List API Integration

Now that the **GraphQL List API** is functional, we need to **integrate it into the UI** by modifying the IssueList component in React.

1. Include Fetch Polyfill (For Older Browsers)

Since we are using fetch(), we include a **polyfill** for Internet Explorer and older browsers in index.html:

<script src="https://unpkg.com/@babel/polyfill@7/dist/polyfill.min.js"></script>

<script src="https://unpkg.com/whatwg-fetch@3.0.0/dist/fetch.umd.js"></script>

2. Modify the loadData() Method in IssueList

Construct GraphQL Query

- Create a GraphQL query string to fetch all issue fields:

```
const query = `query {
  issueList {
    id title status owner
    created effort due
  }
}`;
```

Make an API Request using fetch()

- Use the fetch() method to send a **POST** request to /graphql:

```
const response = await fetch('/graphql', {
  method: 'POST',
  headers: { 'Content-Type': 'application/json' },
  body: JSON.stringify({ query })
});
```

Handle the API Response

- Convert the response **JSON** into a JavaScript object and update state:

```
const result = await response.json();
this.setState({ issues: result.data.issueList });
```

3. Fixing Date Handling in IssueRow Component

- Initially, created and due were Date objects, but now they are **strings**.
- **Fix:** Remove .toDateString() and use the string values directly:

```
<td>{issue.created}</td>
<td>{issue.due}</td>
```

4. Remove the Hardcoded Initial Issues

- Previously, the initialIssues array was used as default data.
- Since data is now fetched dynamically, **remove initialIssues** from App.jsx.

5. Testing the Integration

- **Refresh the browser**: The issue list will now be loaded from the API.
- The UI **remains the same**, except:
  - **Long date strings** appear instead of formatted dates.
  - **Add operation does not work** (we will fix this in the next section).

Custom Scalar Types

Using Custom Scalar Types for Date in GraphQL

Storing dates as **strings** in the database or API responses **is not ideal** because:

1. **Sorting and filtering become complex** (since string-based sorting does not work correctly for dates).
2. **Time zone and localization issues** (dates should be displayed in the user's local time).
3. **Standardization issues** (different formats can cause inconsistencies).

1. Define a Custom GraphQL Scalar Type for Date

GraphQL does not support Date natively, so we create a **custom scalar type**.

Modify schema.graphql

- Use scalar instead of type to define a **new GraphQL type** for dates.
- Update the Issue type to use GraphQLDate instead of String.

```
scalar GraphQLDate

type Issue {
  id: Int!
  title: String!
  status: String!
  owner: String
  effort: Int
  created: GraphQLDate!
  due: GraphQLDate
}
```

2. Implement the Resolver for GraphQLDate

A **GraphQL scalar type resolver** handles:

- **Serialization** (converting a Date to an ISO string for API responses).
- **Parsing** (converting an ISO string back into a Date object when receiving input).

Modify server.js

1. **Import the required package**:

```
const { GraphQLScalarType } = require('graphql');
```

2. **Create the custom GraphQLDate type resolver**:

```
const GraphQLDate = new GraphQLScalarType({
  name: 'GraphQLDate',
  description: 'A Date() type in GraphQL as a scalar',
  serialize(value) {
    return value.toISOString(); // Convert Date to ISO 8601 string
  },
  parseValue(value) {
    return new Date(value); // Convert ISO string to Date object
  },
  parseLiteral(ast) {
    return new Date(ast.value); // Convert AST literal to Date object
  }
});
```

3. **Include GraphQLDate in the resolvers object**:

```
const resolvers = {
  Query: {
    about: () => aboutMessage,
    issueList,
  },
  Mutation: {
    setAboutMessage,
  },
  GraphQLDate, // Add custom scalar type
};
```

3. **How This Works**

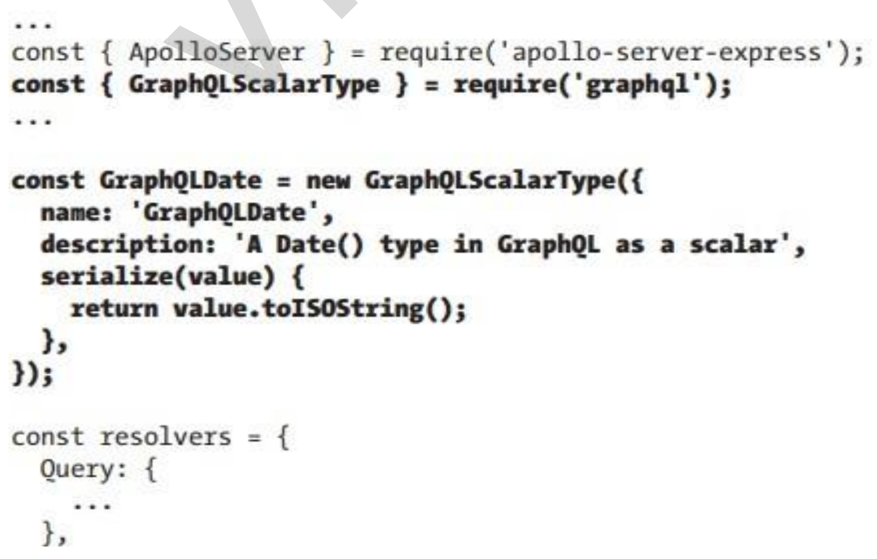
API Response**: When fetching issues, GraphQLDate will convert Date objects to** ISO 8601 strings**.**
API Input Handling**: If a new issue is added with a date in ISO format, it will be** converted back to a Date object **automatically.**

4. Next Steps

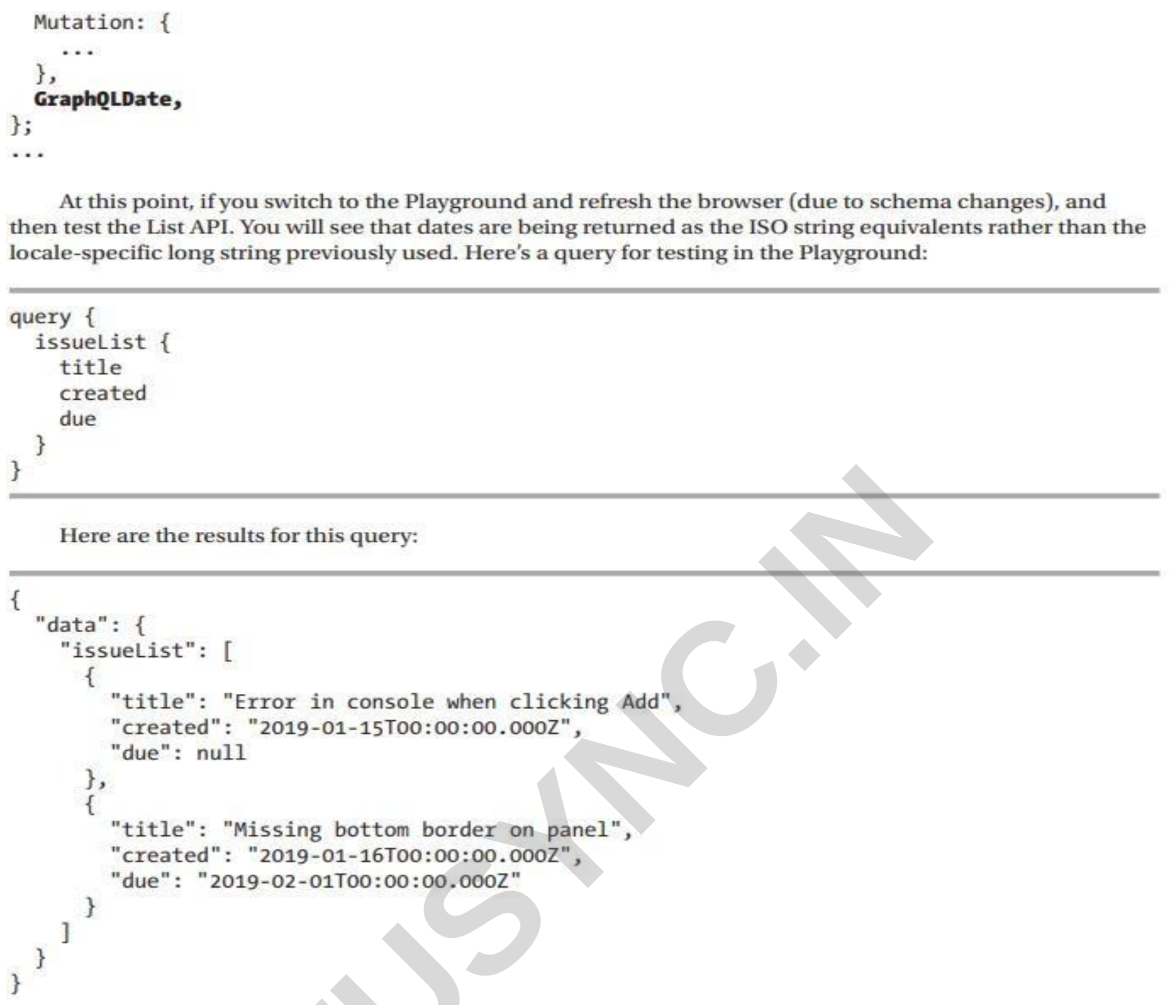
1. **Update the front-end** to properly handle GraphQLDate values.
2. **Ensure UI displays dates in a localized format** using toLocaleDateString() or similar methods.
3. **Modify the Add Issue mutation** to accept ISO date strings.

Finally, we need to set this resolver at the same level as Query and Mutation (at the top level) as the value for the scalar type GraphQLDate. The complete set of changes in server.js is shown in Listing 5-10.

Listing 5-10. server.js: Changes for Adding a Resolver for GraphQLDate



```
...
const { ApolloServer } = require('apollo-server-express');
const { GraphQLScalarType } = require('graphql');
...

const GraphQLDate = new GraphQLScalarType({
  name: 'GraphQLDate',
  description: 'A Date() type in GraphQL as a scalar',
  serialize(value) {
    return value.toISOString();
  },
});

const resolvers = {
  Query: {
    ...
  },
```



```
  Mutation: {
    ...
  },
  GraphQLDate,
};
...
```

At this point, if you switch to the Playground and refresh the browser (due to schema changes), and then test the List API. You will see that dates are being returned as the ISO string equivalents rather than the locale-specific long string previously used. Here's a query for testing in the Playground:

```
query {
  issueList {
    title
    created
    due
  }
}
```

Here are the results for this query:

```
{
  "data": {
    "issueList": [
      {
        "title": "Error in console when clicking Add",
        "created": "2019-01-15T00:00:00.000Z",
        "due": null
      },
      {
        "title": "Missing bottom border on panel",
        "created": "2019-01-16T00:00:00.000Z",
        "due": "2019-02-01T00:00:00.000Z"
      }
    ]
  }
}
```

The Create API

Implementing an API for Creating a New Issue in GraphQL

Now, we will add an API endpoint to **create new issues** in our in-memory database. This involves:

1. **Defining an IssueInputs input type** in schema.graphql.
2. **Adding the issueAdd mutation** to the GraphQL schema.
3. **Implementing the resolver** for issueAdd in server.js.
4. **Updating the GraphQLDate scalar** to handle parsing input dates.

1. Modify schema.graphql

First, define an **input type** IssueInputs (separate from the Issue type because it does not include id or created).

```
"Toned down Issue, used as inputs, without server-generated values."
input IssueInputs {
  title: String!
  "Optional, if not supplied, will be set to 'New'"
  status: String
  owner: String
  effort: Int
  due: GraphQLDate
}

type Mutation {
  setAboutMessage(message: String!): String
  issueAdd(issue: IssueInputs!): Issue!
}
```

- **Mandatory fields**: title (required with !).
- **Optional fields**: status, owner, effort, due (GraphQLDate).
- **Default value**: status is set to "New" if not provided.

2. Implement issueAdd Resolver in server.js

Now, implement the resolver function to:

- **Generate id and created fields automatically**.
- **Default status to 'New' if missing**.
- **Push the issue to issuesDB**.

Modify server.js

1. **Define issueAdd function**:

```
const issuesDB = []; // In-memory storage

function issueAdd(_, { issue }) {
  issue.created = new Date();
  issue.id = issuesDB.length + 1;

  if (issue.status === undefined) {
    issue.status = 'New';
  }

  issuesDB.push(issue);
  return issue;
}
```

2. **Add issueAdd to the Mutation resolver**:

```
const resolvers = {
  Query: {
    about: () => aboutMessage,
    issueList,
  },
  Mutation: {
    setAboutMessage,
    issueAdd, // New Mutation
  },
  GraphQLDate, // Custom Date scalar
};
```

3. Update GraphQLDate Resolver to Handle Parsing

Since IssueInputs includes GraphQLDate, we need to update the **date scalar resolver**.

1. **Import Kind from graphql/language**:

```
const { GraphQLScalarType, Kind } = require('graphql');
```

2. **Modify GraphQLDate resolver**:

```
const GraphQLDate = new GraphQLScalarType({
  name: 'GraphQLDate',
  description: 'A Date() type in GraphQL as a scalar',
  serialize(value) {
    return value.toISOString(); // Convert Date to ISO 8601 string
  },
  parseValue(value) {
    return new Date(value); // Convert ISO string to Date object
  },
  parseLiteral(ast) {
    return ast.kind === Kind.STRING ? new Date(ast.value) : undefined;
  }
});
```

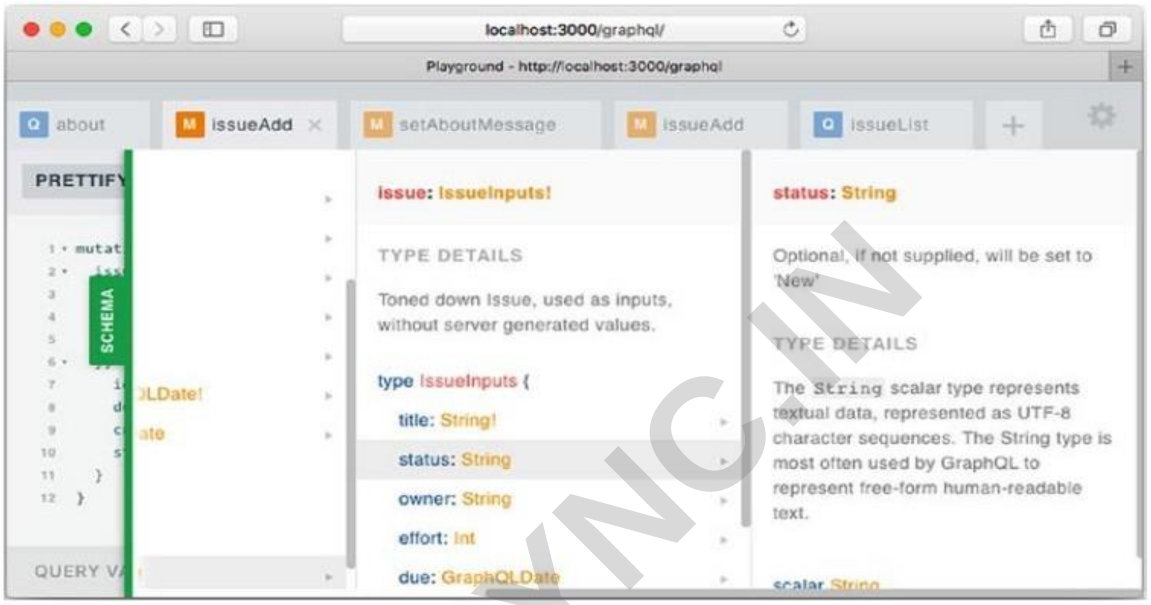
4. Testing the issueAdd Mutation

Now, test the API by sending the following mutation request in GraphiQL or Postman:

```
mutation {
  issueAdd(issue: {
    title: "GraphQL Issue",
    owner: "Alice",
    effort: 5,
    due: "2025-02-15T12:00:00.000Z"
  }) {
    id
    title
    status
    owner
    effort
    created
    due
  }
}
```

Expected Response

```
{
  "data": {
    "issueAdd": {
      "id": 1,
      "title": "GraphQL Issue",
      "status": "New",
      "owner": "Alice",
      "effort": 5,
      "created": "2025-02-12T10:30:00.000Z",
      "due": "2025-02-15T12:00:00.000Z"
    }
  }
}
```



localhost:3000/graphql/

Playground - http://localhost:3000/graphql

issue: IssueInputs!

TYPE DETAILS

Toned down Issue, used as inputs, without server generated values.

```
type IssueInputs {
  title: String!
  status: String
  owner: String
  effort: Int
  due: GraphQLDate
```

status: String

Optional, if not supplied, will be set to 'New'

TYPE DETAILS

The `String` scalar type represents textual data, represented as UTF-8 character sequences. The String type is most often used by GraphQL to represent free-form human-readable text.

scalar String

Figure 5-3. *Schema showing descriptions of IssueInputs and status*

Create API Integration

In this integration, we modify the **IssueAdd** component and the **createIssue** function to send a new issue to the server via GraphQL mutation.

1. Modify IssueAdd to Handle User Input

- We remove the default "status": "New" from the frontend since the backend handles it.
- We set the due date to **10 days from today** using JavaScript.

```
const issue = {
  owner: form.owner.value,
  title: form.title.value,
  due: new Date(new Date().getTime() + 1000 * 60 * 60 * 24 * 10), // 10 days later
};
```

2. Construct the GraphQL Mutation Query

- We create a mutation query using a **template string**.
- Since GraphQL requires date fields as strings, we convert due to **ISO format**.
- We only request the id field in the response.

```
const query = `mutation {
  issueAdd(issue:{
    title: "${issue.title}",
    owner: "${issue.owner}",
    due: "${issue.due.toISOString()}"
  }) {
    id
  }
}`;
```

3. Send the GraphQL Request Using fetch

- We send the query using a **POST request** with JSON body.
- Headers specify that the request contains JSON.

```
const response = await fetch('/graphql', {
  method: 'POST',
  headers: { 'Content-Type': 'application/json' },
  body: JSON.stringify({ query })
});
```

4. Refresh the Issue List Instead of Updating State Manually

- Instead of adding the new issue to state.issues, we **reload the entire issue list**.
- This ensures data accuracy in case of errors or concurrent updates.

```
this.loadData(); // Reload issue list
```

5. Result

- The new issue is added with a **due date of 10 days from now**.
- Refreshing the page shows the new issue because it is stored on the server.

Query Variables

Instead of embedding dynamic values directly inside the query string, GraphQL allows us to use **query variables**, which are passed separately in a JSON object. This approach improves **security, readability, and reliability** by avoiding issues with escaping special characters.

Why Use Query Variables?

1. **Avoids string concatenation issues** – Prevents errors with special characters like quotes (") and curly braces ({}).
2. **More secure** – Similar to prepared statements in SQL, reducing risks like **GraphQL Injection**.
3. **Easier debugging** – Queries remain **clean** while variables are passed separately.

How to Use Query Variables in a Mutation?

1. Name the Mutation

We name the mutation (setNewMessage) and replace **static values** with variables (starting with $).

Before (hardcoded value inside query):

```
mutation {
  setAboutMessage(message: "New About Message")
}
```

After (using a variable $message):

```
mutation setNewMessage($message: String!) {
  setAboutMessage(message: $message)
}
```

2. Declare Variables in JSON Format

When executing this query (e.g., in GraphQL Playground or an API call), the **variables must be passed separately** as JSON.

```
{
  "message": "Hello World!"
}
```

Applying This to issueAdd Mutation

Instead of inserting values directly in a template string, we pass them as variables.

Before (string template approach, error-prone)

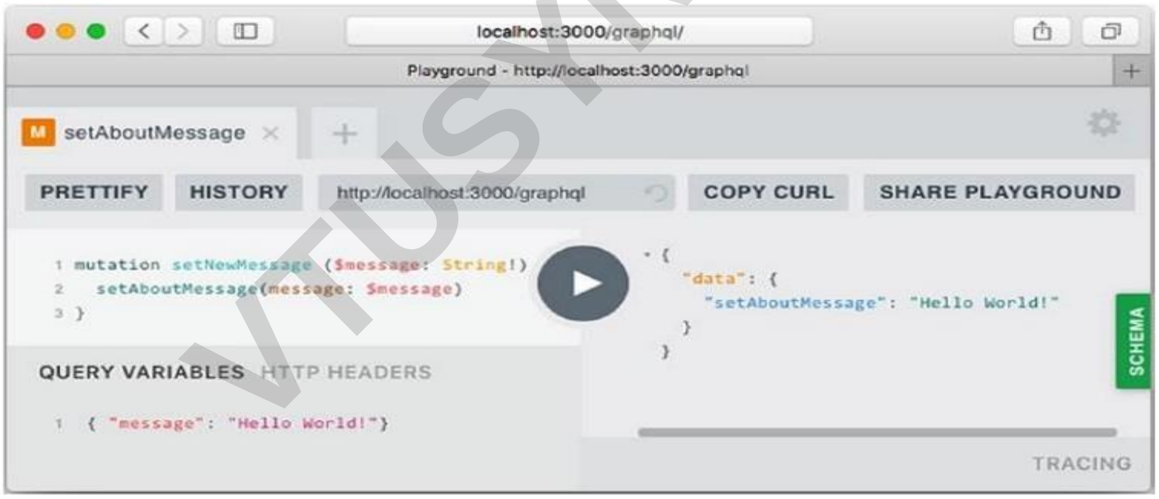
```
const query = `mutation {
  issueAdd(issue:{
    title: "${issue.title}",
    owner: "${issue.owner}",
    due: "${issue.due.toISOString()}"
  }) {
    id
  }
}`;
```

After (query variables approach, safer and cleaner)

```
const query = `mutation addIssue($issue: IssueInputs!) {
  issueAdd(issue: $issue) {
    id
  }
}`;

const variables = {
  issue: {
    title: issue.title,
    owner: issue.owner,
    due: issue.due.toISOString()
  }
};

const response = await fetch('/graphql', {
  method: 'POST',
  headers: { 'Content-Type': 'application/json' },
  body: JSON.stringify({ query, variables })
});
```

Benefits of Using Query Variables

✔**More Secure** – Avoids issues with special characters in user input.
✔**Easier Debugging** – Query stays static while variables are dynamic.
✔**Improves Performance** – GraphQL servers can **cache and optimize** queries better.



```
mutation setNewMessage ($message: String!)
  setAboutMessage(message: $message)
}
```

```
{ "message": "Hello World!"}
```

```
{
  "data": {
    "setAboutMessage": "Hello World!"
  }
}
```

Figure 5-4. *Playground with query variables*

Input Validations

1. Input Validations :Input validation ensures that user input meets specific criteria before being processed by the server. In GraphQL, we can implement validation in two primary ways:

a. Schema-Level Validations

- **Using Enums for Restricted Values:**
  - Instead of using a plain string for certain fields (e.g., status), GraphQL allows us to use enums to limit the possible values.
  - Example:

    ```
    enum StatusType {
      New
      Assigned
      Fixed
      Closed
    }

    type Issue {
      status: StatusType!
    }
    ```

  - If a user provides an invalid value (e.g., "Unknown" instead of New), GraphQL will return a validation error automatically.
- **Providing Default Values:**
  - Default values can be assigned to fields to prevent missing values.
  - Example:

    ```
    input IssueInputs {
      status: StatusType = New
    }
    ```

  - If a user does not provide a status, it will default to "New".

b. Programmatic Validations

- Implemented in the resolver function before saving data.
- Example validation rules:
  - **Title must be at least 3 characters long**
  - **Owner is required when status is "Assigned"**
  - **Invalid date values should be detected**

Example:

```
function validateIssue(_, { issue }) {
```

```
  const errors = [];

  if (issue.title.length < 3) {
    errors.push('Field "title" must be at least 3 characters long.');
  }

  if (issue.status === 'Assigned' && !issue.owner) {
    errors.push('Field "owner" is required when status is "Assigned"');
  }

  if (errors.length > 0) {
    throw new UserInputError('Invalid input(s)', { errors });
  }
}
```

- **Validating Dates:**
  - Ensuring the provided date is in the correct format.
  - Example:

    ```
    parseValue(value) {
      const dateValue = new Date(value);
      return isNaN(dateValue) ? undefined : dateValue;
    }
    ```

Displaying Errors

Errors must be displayed properly in the UI to improve the user experience. We handle two types of errors:

a. Transport Errors (e.g., Network Issues)

- These errors occur when there is a problem with the API request.
- Handled using a try-catch block around the fetch function.
- Example:

  ```
  async function graphQLFetch(query, variables = {}) {
    try {
      const response = await fetch('/graphql', {
        method: 'POST',
  ```

```
      headers: { 'Content-Type': 'application/json' },
      body: JSON.stringify({ query, variables })
    });

    const result = await response.json();
    return result.data;
  } catch (e) {
    alert(`Error in sending data to server: ${e.message}`);
  }
}
```

b. User Input Errors (Validation Failures)

- These errors occur when the user provides invalid data (e.g., empty title, missing owner).
- Displaying errors for user input:

```
if (result.errors) {
  const error = result.errors[0];

  if (error.extensions.code === 'BAD_USER_INPUT') {
    const details = error.extensions.exception.errors.join('\n ');
    alert(`${error.message}:\n ${details}`);
  } else {
    alert(`${error.extensions.code}:  ${error.message}`);
  }
}
```

c. Example Error Messages

1. **Missing Title**

```
{
  "message": "Invalid input(s)",
  "errors": ["Field \"title\" must be at least 3 characters long."]
}
```

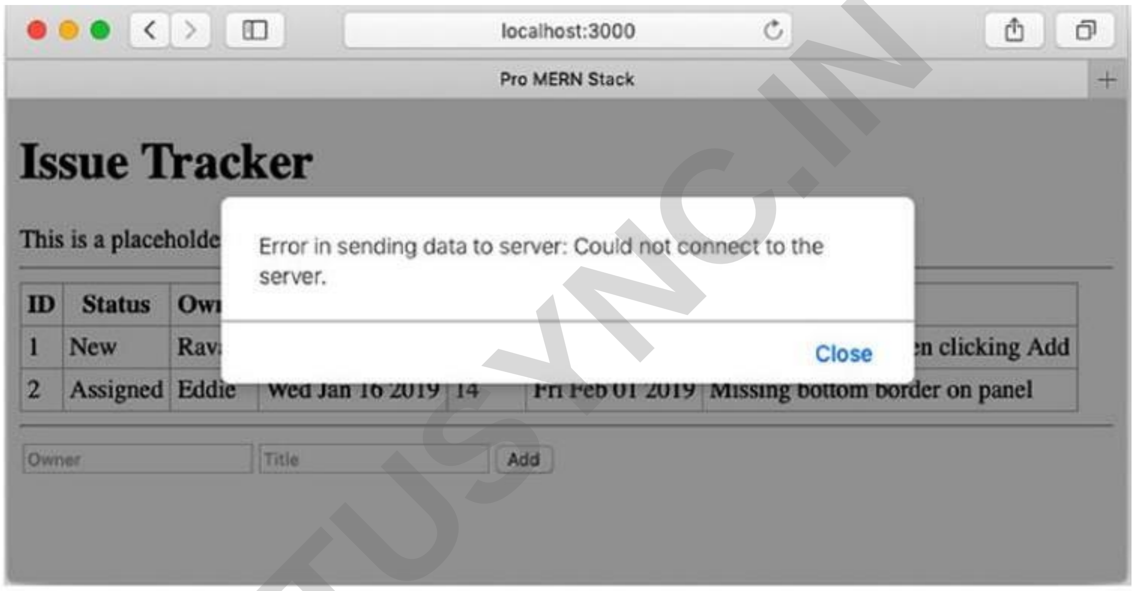
2. **Invalid Date**

```
{
  "message": "Expected type GraphQLDate, found \"not-a-date\"."
}
```

3. **Invalid Status**

```
{
  "message": "Expected type StatusType, found \"Unknown\"."
}
```

To test transport errors, you can stop the server after refreshing the browser and then try to add a new issue. If you do that, you will find the error message like the screenshot in Figure 5-5.



Error in sending data to server: Could not connect to the server.

Figure 5-5. Transport error message