Full Stack

Thilan Tran

Summer 2019

Contents

Full Stack Course

Overview

- server and web browser communicate through **HTTP** protocol
  - browser makes *requests*, server *responds* to requests
    - every webpage makes requests to GET requests to static files on load:
      - eg. HTML page, CSS style sheet, JS script file
  - request types include GET, POST, etc.
  - response headers define status code, response size, time, content-type
- traditionally, application logic is on the *server*
- however, *browser* can:
  - application logic using requests for data (with AJAX) to fetch dynamic content
  - modify the HTML being rendered through the **Document Object Model** (DOM)
- a **Single Page Application** (SPA) comprises of only one HTML page:
  - contents are manipulated purely with JS in the browser
  - rather than having separate pages fetched from the server

AJAX and dynamic content with pure Javascript:

```
var xhttp = new XMLHttpRequest();

// attaching a callback to an event handler
xhttp.onreadystatechange = function() {
  if (this.readyState == 4 && this.status == 200) {
    const data = JSON.parse(this.responseText);
    console.log(data);
  }

  // DOM manipulation
  var ul = document.createElement('ul');
  ul.setAttribute('class', 'notes');

  data.forEach(function(note) {
    var li = document.createElement('li');
    ul.appendChild(li);
    li.appendChild(document.createTextNode(note.content));
  })

  document.getElementById('notes').appendChild(ul);
}

xhttp.open('GET', '/data.json', true);
xhttp.send();

// AJAX POST
xhttpPost = new XMLHttpRequest();
xhttpPost.open('POST', '/new_note', true);
xhttpPost.setRequestHeader('Content-type', 'application/json');
xhttpPost.send(JSON.stringify(note));
```

React Introduction

- useful tool for quick starting React app: `npx create-react-app projectName`
- React is made up of **components**
  - the root App component is then rendered to the DOM of an empty HTML page
  - reusable components can be nested and combined together
- React uses JSX code to embed HTML code within JS
  - allows for dynamic content within components
  - JSX is compiled into regular JS code using Babel
- data is passed to components through **props**
  - functional components receive all props as an argument
  - can easily *destructure* props directly
- JS allows helper functions to be defined within functions
  - provides functional programming techniques such as map, reduce, filter
- modularize components into **modules**

Basics of React:

```
import React from 'react';
import ReactDOM from 'react-dom';

const Hello = ({ name, age }) => {
  return (
    <div>
      <p>Hello {name}, you are {age} years old.</p>
    </div>
  )
}

const App = () => {
  const name = 'Bob';
  const age = 30;

  return (
    <div>
      <Hello name="Builder" age={26+10} />
      <Hello name={name} age={age} />
    </div>
  )
}

ReactDOM.render(<App />, document.getElementById('root'));
```

- add **state** to a component using the `useState` hook
  - `const [state, setState] = useState(initState)`
  - hook returns a variable representing the state and a function to set the state
  - React *automatically* re-renders a component when its state changes
  - for more complex state:
    - use the hook multiple times to create separate state pieces
  - should *never* mutate component state directly:
    - always use immutable functions such as concat, or construct new state using spread syntax
    - allows React to easily detect change in state and optimize Virtual DOM
- use **event handlers** to register callbacks to certain events:
  - eg. on button click, or form submit
  - with parametrized event handlers, *curry* the function
    - otherwise, the function will be called immediately
- an option for sharing data with child components:
  - pass state and event handlers to child components
- general rules for hooks:
  - never called from inside of a loop or conditional expression
  - only called from function body defining a component
- utilize **conditional rendering** for more dynamic component
- arrays/collections are often *mapped* into React components
  - each element needs a unique **key** prop to distinguish itself
    - allows React to easily detect changes in state

- styling React components:
  - use a stylesheet:
    - use `className` property on components
  - use inline styles as a JS object
    - use `style` property

Counting component using state and event handlers:

```
import { useState } from 'react';

const App = () => {
  const [counter, setCounter] = useState(0);

  // currying the function
  setVal = (val) => () => setCounter(val);

  return (
    <div>
      <div>{counter}</div>
      // <button onClick={setCounter(counter+1)}>plus</button>
      // currying the function so it is not called immediately
      <button onClick={() => setCounter(counter+1)}>plus</button>

      <button onClick={setVal(0)}>zero</button>
    </div>
  );
}
```

- *controlled* components are a common pattern with React forms
  - each input is saved into a `useState` hook
  - the state is set automatically on input change

Example controlled React form:

```
const App = (props) => {
  const [notes, setNotes] = useState(props.notes);
  const [content, setContent] = useState('');

  const addNote = (e) => {
    e.preventDefault();
    setNotes(notes.concat({ content, date: new Date().toISOString() }))
  };

  return (
    <div>
```

```
   ...
   <form onSubmit={addNote}>
    <input value={newNote} onChange={({target}) => setContent(target.value)}/>
    <button type="submit">save</button>
   </form>
  </div>
 );
}
```

Server Communication

- communication with the server from the browser happens *asynchronously*:
  - using callbacks, promises, or async/await
  - **promises** have distinct states:
    - pending, fulfilled/resolved, or rejected
  - can chain promises using `.then` , and catch errors using `.catch`
- instead of native javascript, Axios library handles requests with promises

Using Axios:

```
axios
 .get('https://localhost:3001/api')
 .then(res => {
  const data = res.data;
  console.log(data);
 });
```

- the `useEffect` React hook deals with side effects in components:
  - eg. fetching data, handling subscriptions, and manually changing the DOM
  - takes the callback effect and an array of dependencies to determine when to rerun the hook
    - empty array indicates effect is only to be run once on first render

Using the effect hook:

```
useEffect(() => {
 console.log('start of effect');
 axios
  .get('https://localhost:3001/api')
  .then(res => {
   console.log('promise fulfilled');
   const data = res.data;
```

```
      setData(data);
    });
}, [])
```

Creating a service module for backend communication:

```
const baseUrl = '...';

// returns a promise, extracts data field from response
const getAll = () =>
  axios.get(baseUrl).then(res => res.data);

const create = (obj) =>
  axios.post(baseUrl, obj).then(res => res.data);

const update = (id, obj) =>
  axios.put(`${baseUrl}/${id}`, obj).then(res => res.data);

export default { getAll, create, update };
```

Using the service module in React:

```
import noteService from './services/notes';

const App = () => {
  ...
  useEffect(() => {
    noteService.getAll().then(init => setNotes(init));
  }, []);

  const toggleImportance = (id) => {
    const note = notes.find(n => n.id === id);
    const updated = { ...note, important: !note.important };

    noteService
      .update(id, updated)
      .then(returnedNote => {
        setNotes(notes.map(note => note.id !== id ? note: returnedNote));
      })
      .catch(err => {
        alert('Note was already deleted from server');
        setNotes(notes.filter(n => n.id !== id));
      });
  };
  const addNote = (e) => {
    e.preventDefault();
    const newNote = { ... };

    noteService.create(newNote).then(returnedNote => {
      setNotes(notes.concat(returnedNote));
      setContent('');
    })
  };
}
```

Node.js and Express

- NodeJS is a JS runtime based on Chrome's V8 JS engine
  - allows server applications to be written in Javascript
- `npm init` to start a Node application
  - project details in `package.json` file
  - use nodemon module to automatically watch file changes
- **REST** API is a convention for organizing resources by url on the server
  - Representational State Transfer
    - defines a uniform interface
  - every resource should have a unique identifier
    - fetched with GET requests
    - added with POST requests
    - edited with PUT requests
    - deleted with DELETE requests
  - Postman program to test api requests
- handling request conventions:
  - GET requests should be *safe*:
    - not cause any side effects (changes in database)
  - all requests except POST (eg. PUT, DELETE) should be *idempotent*:
    - if a request has side effects:
      - result should be the same regardless of how many times request is sent

Simple web server with pure Node:

```
// Node doesn't use latest ES6 import/export syntax
const http = require('http');

const data = { ... };

const app = http.createServer((req, res) => {
  res.writeHead(200, { 'Content-Type': 'application/json' });
  res.end(JSON.stringify(data));
});

const PORT = 3001;
app.listen(port);
console.log(`Server running on port ${PORT}`);
```

- Express library is an alternative to http module
  - simpler routing API

    - gives access to request and response objects
        - access headeres with `request.headers`
- every route in Express is **middleware** that processes requests/responses
    - app can use several middleware at the same time:
        - executed one by one, in order
    - other types of middlewares include loggers, error handlers

Simple web server with Express:

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

const data = [ ... ];

app.get('/', (req, res) => {
  // automatically sets Content-Type to text/html, status 200
  res.send('<h1>Hello World!</h1>');
});

app.get('/api', (req, res) => {
  // automatically sets Content-Type to application/json
  res.json(data);
});

const PORT = 3001;
app.listen(PORT, () => {
  console.log(`Server running on port ${PORT}`);
});
```

Route parameters with Express:

```javascript
app.get('/notes/:id', (req, res) => {
  // Express casts parameters to strings
  const id = Number(req.params.id);
  const note = notes.find(note => note.id === id);

  if (note) {
    res.json(note);
  } else{
    res.status(404).end();
  }
});

app.delete('/notes/:id', (req, res) => {
  const id = Number(req.params.id);
  notes = notes.filter(note => note.id !== id);
  res.status(204).end();
})
```

Receiving POST object with Express:

```
// registering json middleware to read request body json
app.use(express.json());

app.post('/notes', (req, res) => {
  const note = req.body;

  if (!body.content) {
    // 400 bad request
    return res.status(400).json({ error: 'content missing' });
  }
  notes = notes.concat({ ...note, id: id() });
  res.json(note);
})
```

Using async/await:

```
notesRouter.post('/', async (req, res, next) => {
  const body = req.body;
  const note = new Note({
    content: body.content,
    important: body.important === undefined ? false : body.important,
    date: new Date()
  });

  // wrap in try-catch to use async/await
  try {
    const savedNote = await note.save();
    res.json(savedNote.toJSON());
  } catch (exception) {
    next(exception);
  }
})
```

Other types of Express middleware:

```
const cors = require('cors');

```

```
// set allow cross-origin CORS headers
app.use(cors());

const reqLogger = (req, res, next) => {
  console.log('Method:', req.method);
  console.log('Path:  ', req.path);
  // must be used after express json middleware
  console.log('Body:  ', req.body);
  console.log('-------');
  // yield to next middleware
  next();
};

app.use(reqLogger);

const unknownEndpoint = (req, res) => {
  // final middleware
  res.status(404).send({ error: 'Unknown endpoint'});
};

app.use(unknownEndpoint);
```

Deployment

- use Heroku to deploy web applications:
  - server backend:
    - define `Procfile` for starting the application
    - use environment variables for configuring ports and urls
      - `const PORT = process.env.PORT || 3001`
    - `heroku create` , `git push heroku master`
  - frontend production build:
    - `npm run build` to create a production build
      - creates `build` directory with minified JS code
      - serve these static files from the server using static middleware
      - `app.use(express.static('build'))`
  - set proxy to handle server url in developement mode:
    - `"proxy": "http://localhost:3001"`
- use `.env` files and dotenv package to set environment variables
  - `require('dotenv').config()`
  - can set environment variables on Heroku: `heroku config: set VAR=...`

Databases

- databases store server data indefinitely
- **NoSQL** or document databases
  - loosely structured, schemaless
    - application defines schema
  - eg. MongoDB, online providers such as Mongo Atlas
  - mongoose library for use with Express
    - Object Doucment Mapper saves JS objects as Mongo documents
- **SQL** or relational databases
  - defined structure, organized as columns

Using mongoose:

```
// connecting:
const mongoose = require('mongoose');

// disables error messages for findByIdAndUpdate
mongoose.set('useFindAndModify', false);

const url = process.env.MONGODB_URI;
mongoose.connect(url, { useNewUrlParser: true });

// defining schema:
const noteSchema = new mongoose.Schema({
  content: {
    // defining validators for fields
    type: String,
    minlength: 5,
    required: true
  },
  date: {
    type: Date,
    required: true
  },
  important: Boolean,
});
// formatting objects:
noteSchema.set('toJSON', {
  transform: (doc, obj) => {
    obj.id = obj._id.toString();
    // delete _id and versioning field
    delete obj._id;
    delete obj.__v;
  }
});
const Note = mongoose.model('Note', noteSchema);

module.exports = Note;

// creating new object from model:
const note = new Note({
  content: '...',
  date: new Date(),
  important: false
});

// saving object happens asynchronously:
note.save().then(res => {
  console.log('note saved!');
  // close connection
  mongoose.connection.close();
})
```

Fetching objects from database:

```
// finding all notes:
api.get('/api/notes', (req, res) => {
  Note.find({}).then(notes => {
    res.json(notes);
  });
});

// finding specific notes:
Note.find({ important: true}).then(res => ...);

// finding by id:
api.get('/api/notes/:id', (req, res, next) => {
  Note.findById(req.params.id).then(note => {
    if (note) {
      res.json(note.toJSON());
    } else {
      res.status(404).end();
    }
  })
  // pass errors to custom handler
  .catch(err => next(err));
});
```

Other operations with mongoose:

```
app.post('/api/notes', (req, res, next) => {
  const body = req.body;
  if (!body) {
    return res.status(400).json({ error: 'content missing' });
  }

  const note = new Note({
    content: body.content,
    important: body.important || false,
    date: new Date()
  });

  note.save().then(saved => res.json(saved.toJSON())
    .catch(err => next(err)));
});

app.delete('/api/notes/:id', (req, res, next) => {
  Note.findByIdAndRemove(req.params.id)
    .then(result => res.status(204).end())
    .catch(err => next(err));
});

app.put('/api/notes/:id', (req, res, next) => {
  const body = req.body;
  const note = { content: body.content, important: body.important };

  Note.findByIdAndUpdate(req.params.id, note, { new: true })
    .then(updated => res.json(updated.toJSON()))
    .catch(err => next(err));
});
```

Using a custom error handler:

```
const errorHandler = (error, req, res, next) => {
  console.error(error.message);
  if (error.name === 'CastError' && error.kind === 'ObjectId') {
    return res.status(400).json({ error: 'malformatted id' });
  } else if (error.name === 'ValidationError') {
    return res.status(400).json({ error: error.message });
  } else if (error.name === 'JsonWebTokenError') {
    return res.status(401).json({ error: 'invalid token' });
  }
  // pass to default express error handler
  next(error);
}
app.use(errorHandler);
```

Express Testing and User Administration

- project *structure* conventions:
  - `index.js` simplified to only starting the server
  - `app.js`
  - `build/`
  - `controllers/` routing code
    - can use Express routers to modularize controllers
  - `models/` database models
  - `package.json` and `package-lock.json`
  - `utils/` config, middleware, misc.
    - `config.js` handles .env and environment variables
      - eg. starts with `require('dotenv').config()` , exports env variables
      - other parts can access through `const cfig = require('./utils/config')`

Using Express routers:

```
// controllers/notes.js
const notesRouter = require('express').Router();

// minimal route url
notesRouter.get('/' ...);

module.exports = notesRouter;

// app.js
const notesRouter = require('./controllers/notes');

// using router as middleware
app.use('/api/notes', notesRouter);
```

Testing

- Jest library handles testing the backend
  - `jest --verbose` tests `moduleName.test.js` files
    - `jest -t 'notes' --runInBand` runs only tests with 'notes' sequentially
  - use `test` function and `expect` results to pass assertions
    - use `describe` block to group tests

- define execution mode of the application with `NODE_ENV` env variable
  - allows the usage of a different database url for testing
    - eg. `if (process.env.NODE_ENV === 'test') MONGODB_URI = ...`
  - with a testing script: `"test": "NODE_ENV=test jest --verbose --runInBand"'`
- supertest package helps write API tests

Using Jest to test a computing average function:

```
describe('average', () => {
  test('of one value is itself', () => {
    expect(average([1])).toBe(1);
  });

  test('of many', () => {
    expect(average([1, 2, 3, 4, 5, 6])).toBe(3.5);
  });

  test('of empty array is zero', () => {
    expect(average([])).toBe(0);
  });
});
```

Backend tests with Jest and supertest:

```
const supertest = require('supertest');
const app = require('../app');
const api = supertest(app);

const initNotes = [ ... ];

// reset database notes before each test
beforeEach(async () => {
  await Note.deleteMany({});
  const promiseArr = initNotes.map(note => note.save());
  // awaiting suite of promises to initialize database
  await Promise.all(promiseArr);
})

test('notes are returned as json', async () => {
  await api
    .get('/api/notes')
    // expect correct status code
    .expect(200)
    // expect correct content type header
    .expect('Content-Type', /application\/json/);
});

test('there are four notes', async () => {
  const res = await api.get('/api/notes');
  expect(res.body.length).toBe(4);
});

test('first note content matches', async () => {
  const res = await api.get('/api/notes');
  expect(res.body[0].content).toBe('This is the first note.');
});

test('adding a valid note', async () => {
  const newNote = { ... };
  await api
    .post('/api/notes')
    .send(newNote)
    .expect(200)
    .expect('Content-Type', /application\/json/);

  const res = await api.get('/api/notes');
  const contents = res.body.map(r => r.content);

  expect(res.body.length).toBe(initNotes.length + 1);
  expect(contents).toContain(...);
})

test('deleting a note', async () => {
  const deleteMe = initNotes[0];
  await api
    .delete(`/api/notes/${deleteMe.id}`)
    .expect(204);

  const res = await api.get('/api/notes');
  const contents = res.body.map(r => r.content);

  expect(res.body.length).toBe(initNotes.length - 1);
  expect(contents).not.toContain(deleteMe.content);
})

// execute at end of tests
```

```
afterAll(() => mongoose.connection.close());
```

Silencing logger on testing environment:

```
const info = (...params) => {
  // silence logging information
  if (process.env.NODENV !== 'test') {
    console.log(...params);
  }
}
const reqLogger = (req, res, next) => {
  info(...)
  ...
}
```

User Administration

- *users* stored as their own model in databases
  - eg. users creating notes
  - in a *relational* database, both resources would have separate tables:
    - the user id who creates a note would be stored in the notes table as a foreign key
  - in a *document* database, there are more options for modeling the situation:
    - store just the id unidirectionally or bidirectionally
      - or nest entire notes model within the users collection
    - MongoDB supports ObjectID *references*

Updating the schema for users:

```
const uniqueValidator = require('mongoose-unique-validator');

const userSchema = new mongoose.Schema({
  username: {
    type: String,
    // mongoose-unique-validator
    unique: true
  },
  name: String,
  // must store a hashed password
  passwordHash: String,
  notes: [
    {
      type: mongoose.Schema.Types.ObjectId,
      ref: 'Note'
    }
  ]
});

userSchema.plugin(uniqueValidator);

userSchema.set('toJSON', {
  transform: (doc, obj) => {
    obj.id = obj._id.toString();
    delete obj._id;
    delete obj.__v;
    // hide password hash
    delete obj.passwordHash;
  }
});

const User = mongoose.model('User', userSchema);

module.exports = User;

const noteSchema = new mongoose.Schema({
  ...
  // storing reference in both collections
  user: {
    type: mongoose.Schema.Types.ObjectId,
    ref: 'User'
  }
})
```

Updating references in both collections:

```
notesRouter.post('/', async (req, res, next) => {
  ...
  const user = await User.findById(body.userId);
  const note = new Note({
    ...
    user: user._id
  });

  try {
    const saved = await note.save();
    // updating user as well
    user.notes = user.notes.concat(saved._id);
    await user.save();
    res.json(saved);
  } ...
});
```

Populating queries using Mongoose join queries: (not *transactional*, ie. state of the database can change between queries)

```
userRouter.get('/', async (req, res) => {
  // populating the result of the query using ID from the notes fields
  const users = await User.find({}).populate('notes', {
    // specifying which fields to populate
    content: 1, date: 1
  });
  res.json(users);
});
```

- passwords should be *hashed*, eg. using bcrypt package
  - `saltRounds` of bcrypt determines strength of hashing

New user functionality on server using bcrypt:

```
const bcrypt = require('bcrypt');
const usersRouter = require('express').Router();
const User = require('../models/user');

usersRouter.post('/', async (req, res, next) => {
  try {
    const body = req.body;
    const saltRounds = 10;
    const passwordHash = await bcrypt.hash(body.password, saltRounds);

    const user = new User({
      username: body.username,
      name: body.name,
      passwordHash
    });
    const saved = await user.save();
    res.json(saved);
  } catch (err) {
    next(err);
  }
});

module.exports = usersRouter;
```

- **token authentication** secures certain API actions
  - eg. restricting deleting or creating new notes to logged in users only
  - after users log in through a POST request:
    - the server generates a signed **token** that identifies the user
  - browser saves the token, and send the token with requests that require id
  - server uses the token to identify the user
  - use jsonwebtoken package for generating *JSON web tokens* with a randomized secret key
- to send token from the browser to server, use **Authorization** header
  - multiple schema to interpret credentials
  - jsonwebtoken uses the *Bearer* schema
    - eg. `Bearer elkdlLKjdofwlKLAjsf98dfLSDj`
- HTTPS also increases security, but Heroku server already uses HTTPS in production

Login functionality on server using jsonwebtoken:

```
const jwt = require('jsonwebtoken');
const bcrypt = require('bcrypt');

loginRouter.post('/', async (req, res) => {
  const body = req.body;
  const user = await User.findOne({ username: body.username });
  // check hashed password
  const correctPw = !user ? false :
    await bcrypt.compare(body.password, user.passwordHash);

  if (!(user && correctPw)) {
    // 401 unauthorized
    return res.status(401).json({ error: 'invalid username or password' });
  }

  const jsonToken = {
    username: user.username,
    id: user._id
  };
  const token = jwt.sign(jsonToken, SECRET_KEY);
  res.status(200).send({ token, username: user.username, name: user.name });
});

module.exports = loginRouter;
```

Limiting API actions with token authentication:

```
const getTokenFrom = (req) => {
  const auth = req.get('authorization');
  // extract token from header
  if (auth && auth.toLowerCase().startsWith('bearer ')) {
    return auth.substring(7);
  }
  return null;
}

notesRouter.post('/', async (req, res, next) => {
  const token = getTokenFrom(req);
  try {
    const decoded = jwt.verify(token, SECRET_KEY);
    if (!token || !decoded.id) {
      return res.status(401).json({ error: 'token missing or invalid' });
    }

    const user = await User.findById(decoded.id);
    const note = new Note({ ... });
    ...
  } catch (err) {
    next(err);
  }
})
```

Frontend User Administration

- users log in through the browser with a POST request form with username and password
  - need to save the returned token in state to be used later for secured actions
    - can save and retrieve from the browser's *local storage*
      - persists across page rerendering
    - can clear local storage:

- `window.localStorage.removeItem()`
- `window.localStorage.clear()`

Updating general note service:

```
// note service:
let token = null;

const setToken = (token) => token = `bearer ${token}`;

const create = async (obj) => {
  // configuring headers with token
  const config = {
    headers: { Authorization: token }
  };
  const res = await axios.post(baseUrl, obj, config);
  return res.data;
}

export default { getAll, create, update, setToken };

// corresponding login event handler:
const handleLogin = async (e) => {
  e.preventDefault();
  try {
    // using a controlled form
    const res = await axios.post(baseUrl, { username, password });
    const user = res.data;

    window.localStorage.setItem('loggedUser', JSON.stringify(user));
    noteService.setToken(user.token);
    setUser(user.data);
    setUsername('');
    setPassword('');
  } catch (err) {
    ...
  }
}
```

Retrieving from local storage with effect hook:

```
useEffect(() => {
  const loggedUser = window.localStorage.getItem('loggedUser');
  if (loggedUser) {
    const user = JSON.parse(loggedUser);
    setUser(user);
    noteService.setToken(user.token);
  }
}, []);
```

Frontend Testing and Custom Hooks

Advanced React Props

- can create **Higher Order Components** (HOC) that extends the functionality of child components
  - can access child components through `props.children`
    - children props is automatically added by React
    - empty array if component is defined with auto closing tag

A Toggleable HOC:

```
const Toggleable = (props) => {
  const [visible, setVisible] = useState(false);

  // css display property
  const hideWhenVisible = { display: visible ? 'none' : '' };
  const showWhenVisible = { display: visible ? '' : 'none' };

  const toggleVisible = () => setVisible(!visible);

  return (
    <div>
      <div style={hideWhenVisible}>
        <button onClick={toggleVisible}>{props.buttonLabel}</button>
      </div>
      <div style={showWhenVisible}>
        // child component
        {props.children}
        <button onClick={toggleVisible}>cancel</button>
      </div>
    </div>
  )
};
```

- React *refs* are component references to access component props/state from outside
  - `createRef` to make refs
  - `useImperativeHandle` hook shares component attributes with refs

Toggling Toggleable visibility from outside:

```
import { useImperativeHandle } from 'react';

const Toggleable = React.fowardRef((props, ref) => {
  ...
  // make function available outside of component
  useImperativeHandle(ref, () => {
    return {
      toggleVisible
    };
  });
  ...
});

const App = () => {
  ...
  const noteFormRef = React.createRef();

  const noteForm = () => {
    <Toggleable buttonLabel="new note" ref={noteFormRef}>
      ...
    </Toggleable>
  };

  const addNote = (e) => {
    noteFormRef.current.toggleVisible();
    ...
  };
  ...
};
```

- can force required props on a component using prop-types package

Force required props on Toggleable:

```
import PropTypes from 'prop-types';

const Toggleable = ... {
  ...
  Toggleable.propTypes = {
    buttonLabel: PropTypes.string.isRequired
    // also PropTypes.func
  };
};
```

Frontend Testing

- can expand Jest for use with frontend:
  - react-testing-library and jest-dom packages
  - components should have css classes/id's to select during testing
  - render components, check their text content, click buttons
  - create *stub* components, eg. mock objects and functions

Simple frontend Jest test:

```
import 'jest-dom/extend-react';
import { render, cleanup, fireEvent } from '@testing-library/react';
import { prettyDOM } from '@testing-library/dom';

afterEach(cleanup);

test('renders content', () => {
  const note = { content: 'testing', important: true };
  // special render method does not render to DOM
  const component = render(<Note note={note} />);

  // container property contains all renderd HTML
  expect(component.container).toHaveTextContent('testing');

  const elem = component.getByText('testing');
  expect(elem).toBeDefined();

  // using css selectors
  const div = component.container.querySelector('.note');
  expect(div).toHaveTextContent('testing');

  // printing DOM fragments for debugging
  console.log(prettyDOM(div));
})

test('clicking button calls event handle once', async () => {
  const note = { content: 'testing', important: true };
  // mock function
  const mockHandler = jest.fn();

  const { getByText } = render(<Note note={note} toggleImportance={mockHandler});

  const button = getByText('toggle importance');
  // click button and call handler
  fireEvent.click(button);

  expect(mockHandler.mock.calls.length).toBe(1);
})
```

Testing forms with a wrapper component:

```
const Wrapper = (props) => {
  // custom wrapper HOC to synchronize state with its parent
  const onChange = (e) => props.state.value = event.target.value;

  return (
    <NoteForm
      value={state.props.value}
      onSubmit={props.onSubmit}
      handleChange={onChange}
    />
  );
};

test('Form updates parent state and calls onSubit', () => {
  const onSubmit = jest.fn();
  const state = { value: '' };

  const component = render(<Wrapper onSubmit={onSubmit} state={state} />);

  const input = component.container.querySelector('input');
  const form = component.container.querySelector('form');

  fireEvent.change(input, { target: { value: 'new text' }});
  fireEvent.submit(form);

  expect(onSubmit.mock.calls.length).toBe(1);
  expect(state.value).toBe('new text');
});
```

- **integration** tests of the application as a whole can be more comprehensive:
  - to replace server requests, can use Jest *manual mocks* to replace modules with hardcoded data
  - eg. to replace the `noteService` module, a hardcoded `getAll` function in `__mocks__` directory
    - returns a list of hardcoded notes wrapped in a promise

- **snapshot** testing does not require any defined tests:
  - simply compare HTML code defined by the component after changes
- **end-to-end** tests completely simulate a browser
  - inspect application through same interface as real users
- check test *coverage* of tests using `CI=true npm test -- --coverage` :
  - gives breakdown of untested lines of code in a component

Example integration test:

```
import { waitForElement } from '@testing-library/react';
// module to mock
jest.mock('./services/notes');

describe('App component', () => {
  test('renders all notes from backend', async () => {
    const component = render(<App />);

    // rerender to catch all effect hooks
    component.rerender(<App />);

    // fetching notes is async, wait for App to render all notes
    await waitForElement(() => component.conainer.querySelector('.note'));

    const notes = component.cotnainer.querySelectorAll('.note');
    expect(notes.length).toBe(3);
    expect(component.container).toHaveTextContent('note 1');
    expect(component.container).toHaveTextContent('note 2');
    expect(component.container).toHaveTextContent('note 3');
  })
})
```

Custom Hooks

- **custom hooks** extract component logic into resuable functions
  - follow general *hook rules*:
    - don’t call hooks inside loops, conditionals, or nested functions
    - only call from React function components or other custom hooks
  - names start with `use` by convention

Counter custom hook:

```
const useCounter = () => {
  const [val, setVal] = useState(0);
  const increase = () => setVal(val+1);
  const decrase = () => setVal(val-1);
  const zero = () => setVal(0);

  return { val, increase, decrease, zero };
}

const App = () => {
  // two separate counters
  const left = useCounter();
  const right = useCounter();


  ...
  <button onClick={right.increase}>add to right</button>
  ...
}
```

Form-field custom hook:

```
const useField = (type) => {
  const [value, setValue] = useState('');

  const onChange = (e) => setValue(e.target.value);

  /* matching methods to property names
     allows spread syntax to be used */
  return { type, value, onChange };
}

const App = () => {
  const name = useField('text');
  const born = useField('date');

  return (
    <form>
      name:
      <input {...name} />
      birthdate:
      <input {...born} />
    </form>
  )
}
```

Resource service custom hook:

```
const useResource = (baseUrl) => {
  const [token, setToken] = useState('');
  const [resource, setResource] = useState([]);

  const setAuthToken = (newToken) => setToken(`bearer ${newToken}`);

  useEffect(() => {
    const getAll = () =>
      axios.get(baseUrl).then((init) => setResource(init.data));

    getAll().then(console.log('Initialized resource.'));
  }, [baseUrl]);

  const create = (newResource) => {
    const config = {
      headers: { Authorization: token }
    };
    axios.post(baseUrl, newResource, config).then((created) => {
      setResource(resource.concat(created.data));
      return created.data;
    });
  };

  const update = (id, newResource) =>
    axios.put(baseUrl + '/' + id, newResource).then((updated) => {
      setResource(
        resource.map((r) => (r.id === updated.data.id ? updated.data : r))
      );
      return updated.data;
    });

  const remove = (id) => {
    const config = {
      headers: { Authorization: token }
    };
    axios.delete(baseUrl + '/' + id, config).then((removed) => {
      setResource(resource.filter((r) => r.id !== removed.data.id));
      return removed.data;
    });
  };
  return [
    resource,
    {
      setAuthToken,
      create,
      update,
      remove
    }
  ];
};

// usage:
const [notes, noteService] = useResource('http://localhost:3001/notes');

const handleNoteSubmit = (e) => {
  e.preventDefault();
  noteService.create({ content: content.value });
};

return (
  <div>
    {notes.map(n => <p key={n.id}>{n.content}</p>)}
  </div>
);
```

Redux

- **Flux** is a *state-management* alternative
    - previously, state was stored in the root component
    - passed down other components through props
- state is separated from components into a **store**
- the store is changed with **actions**
    - objects with at least a type field
    - actions are *dispatched* to the store
    - can abstract actions with functions, called **action creators**
- the impact of the action on the store is defined with a **reducer**
    - function taking current state and action as parameters
    - returns a new state (with immutable objects)
        - test immutability with `deep-freeze` module
- get current state with `store.getState()`
- call callback functions on store change with `store.subscribe(callbackFunc)`
    - react will *not* automatically re-render on store change
- *note*: **uncontrolled** forms do not have the state of the form fields bound to the component state
    - limitations include no dynamic errors or disabling submit button

Counter with Redux:

```
import { createStore } from 'redux';

const counterReducer = (state = 0, action) => {
  switch (action.type) {
    case 'INCREMENT':
      return state+1;
    case 'DECREMENT':
      return state-1;
    case 'ZERO':
      return 0;
    default:
      return state;
  }
}

// reducer is never called directly
const store = createStore(counterReducer);

store.dispatch({type: 'INCREMENT'});
```

```
console.log(store.getState())
```

Notes app with Redux:

```
const noteReducer = (state = [], action) => {
  switch(action.type) {
    case 'NEW_NOTE':
      // use immutable array methods (ie. concat, spread syntax)
      return [...state, action.data];
    case 'TOGGLE_IMPORTANCE':
      const id = action.data.id;
      const noteToChange = state.find(n => n.id === id);
      const changedNote = {
        ...noteToChange,
        important: !noteToChange.important
      };

      return state.map(note => note.id !== id ? note : changedNote);
    default:
      return state;
  }
}

const createNote = (content) =>
  { type: 'NEW_NOTE', data: { content, important: false, id: generateId() }};

const createNote = (id) =>
  { type: 'TOGGLE_IMPORTANCE', data: { id }};

store.dispatch(createNote(content));

const render = () => {
  ReactDOM.render(...);
};

// first initiol render, required
render();
// re-render on store update
store.subscribe(render);
```

Complex Redux Stores

- options for sharing the store among components:
  - pass the store as a prop
  - use `connect()` from React-Redux library
    - components must be a child of `Provider` component
      - ie. a *connected component*
      - `mapStateToProps()` and `mapDispatchToProps()` allow store to be manipulated through props

Using `Provider` HOC:

```
import { Provider } from 'react-redux';
...
ReactDOM.render(
  <Provider store={store}>
    <App />
  </Provider>,
  document.getElementById('root')
);
```

Using `connect()` :

```
import { connect } from 'react-redux';

const Notes = ...

const mapStateToProps = (state) => {
  return {
    // accessing reducers' state from props directly
    notes: state.notes,
    filter: state.filter
  };
};

// simplifying mapping the state with a selector function
const notesToShow = ({ notes, filter }) => {
  if (filter === 'ALL') return notes;

  return filter === 'IMPORTANT' ? notes.filter(note => note.important)
                                : notes.filter(note => !note.important);
}
```

```
const mapStateToProps = (state) => {
  return {
    visibleNotes: notesToShow(state)
  }
}

// automatically dispatches action from action creator
const mapDispatchToProps = {
  // dispatching action from action creator from props directly
  toggleImportanceOf
};

// alternative, explicit function syntax for mapping dispatch
const mapDispatchToProps = (dispatch) => {
  return {
    toggleImportanceOf: (id) => dispatch(toggleImportanceOf(id))
  };
};

const ConnectedNotes = connect(mapStateToProps, mapDispatchToProps)(Notes);
export default ConnectedNotes;
```

- combine multiple stores / reducers together:
  - `combineReducers(combinedObj)`
- *note*: *presentational* components are simple, their event handlers are abstracted
  - visual, DOM markup and styles, no dependencies, receive data and callbaks exclusively through props
- while *container* components contain application logic, such as defining event handlers
  - no DOM markup, data handling, stateful, HOC's

Combining multiple reducers:

```
import { createStore, combineReducers } from 'redux';
import noteReducer from './reducers/noteReducer';
import filterReducer from './reducers/filterReducer';

const reducer = combineReducers({
  notes: noteReducer,
  filter: filterReducer
});
```

```
const store = createStore(reducer);

// access a reducer through store.getState().notes
```

Asynchronous Actions

- *redux-thunk* library allows for action creators to be asynchronous functions
  - eg. communicate / update data from a database
  - previously not possible to implement within an action creator

Using Redux thunk:

```
import { applyMiddleware } from 'redux';
import thunk from 'redux-thunk';

...

const store = createStore(reducer, applyMiddleware(thunk));

// action creators can now have asynchronous operations

export const initializeNotes = () => {
  return async (dispatch) => {
    const notes = await noteService.getAll();
    dispatch({ type: 'INIT_NOTES', data: notes });
  }
}

export const createNote = (content) => {
  return async (dispatch) => {
    const newNote = await noteService.createNew(content);
    dispatch({ type: 'NEW_NOTE', data: newNote })
  }
}
```

React Router and Styling

React Router

- *routing* is the navigation management of an application
  - React router from `react-router-dom` is a routing solution
- `Link` component modifies the url in address bar
- url-based component rendering defined with `Route` component
  - match exact paths to only catch parent components
  - access `match` parameter for url variables

Using the React BrowserRouter:

```
import {
  BrowserRouter as Router,
  Route, Link, Redirect, withRouter
} from 'react-router-dom';

const App = () => {
  return (
    <Router>
      <div>
        // navbar elements
        <Link to="/">home</Link>
        <Link to="/notes">notes</Link>
        <Link to="/users">users</Link>
      </div>

      // rendering components based on url
      <Route exact path="/" render={() => <Home />} />
      <Route exact path="/notes" render={() => <Notes />} />
      <Route path="/notes/:id" render={({ match }) =>
        <Note note={noteById(match.params.id)} />
      } />
      <Route path="/users" render={() => <Users />} />

      // conditional rendering
      {user
       ? <em>{user} logged in</em>
       : <Link to="/login">login</Link>
      }
    </Router>
  )
}

const Notes = (props) => (
  ...
    <Link to={'/notes/' + note.id}>{note.content}</Link>
  ...
)
```

Using `withRouter` and `history` to change pages:

```
import {
  withRouter
} from 'react-router-dom';

const Login = (props) => {
  const onSubmit = (e) => {
    e.preventDefault();
    ...
    // render home after login
    props.history.push('/');
  }

  return ...
}

// add history prop to component
const LoginWithHistory = withRouter(Login);
```

Using `redirect` to redirect routes:

```
<Route path="/users" render={() =>
  user ? <Users /> : <Redirect to="/login" />
} />
```

Styles

- **UI Frameworks** are predefined style themes and components
  - eg. Boostrap, Semantic UI, reactstrap, react-bootstrap
  - install CSS stylesheet and npm package

- Bootstrap basics:
  - entire application rendered in a `container` class
  - provides response designs
  - react-bootstrap offers:
    - `Table` component
      - striped, bordered, hover options
    - `Form` component
      - Group, Control, Label subcomponents
    - `Button` component
      - primary, secondary, success variants
    - `Alert` component (same variants) for notifications
    - `Navbar` component
      - Toggle, Collapse, Link subcomponents
- Semantic UI basics:
  - `Container` component
  - `Table` component
    - striped, celled options
    - Body, Row, Cell subcomponents
  - `Form` component
    - Field subcomponent
  - `Message` component
  - `Menu` component
    - Item subcomponent
- **Styled Components** use template literals for defining styles

Using React styled components:

```
import styled from 'styled-components';

const Navigation = styled.div`
  background: grey;
  padding: 1em;
`


const Input = styled.input`
  margin: 0.25em;
`


<Input type='password' />
```

Webpack

- **Webpack** bundles separate modules into one for the browser
  - `npm run build` bundles source code into build directory
  - also handles *transpiling* to bridge JS versions

Webpack configuration from scratch:

1. set up the following directory tree:

- `build`
- `package.json` (empty dependencies)
- `src`
  - `index.js`
- `webpack.config.js`

2. install webpack:

- `npm install --save-dev webpack webpack-cli`

3. define `webpack.config.js`
4. define new npm script

- `"build": "webpack --mode=developement"`

webpack.config.js:

```
const path = require('path');

const config = {
  // entry point for bundling
  entry: './src/index.js',
  output: {
    // __dirname holds current directory
    path: path.resolve(__dirname, 'build'),
    // bundled code
    filename: 'main.js'
  }
};
module.exports = config;
```

Webpack with minimal React:

1. install react: `npm install --save react react-dom`
2. need minimal `build/index.html` file for react to render on

- link to bundled `./main.js` in script tag

3. install babel and other dependencies:

- `npm install --save-dev @babel/core babel-loader @babel/preset-react`

- need polyfill for promises/async/await in some browsers:
  - `npm install --save-dev @babel/polyfill`
  - using library directly:
    * `import PromisePolyfill from 'promise-polyfill'`
    * `if (!window.Promise) window.Promise = PromisePolyfill;`
- for transpiling preset:
  - `npm install --save-dev @babel/preset-env`
- for css loaders: (injected directly into bundled code)
  - `npm install --save-dev style-loader css-loader`

4. configure config with babel to process JSX

webpack.config.js:

```
const config = {
  entry: './src/index.js',
  // for polyfill dependency
  entry: ['@babel/polyfill', './src/index.js'],
  output: {
    path: path.resolve(__dirname, 'build'),
    filename: 'main.js'
  },
  module: {
    rules: [
      {
        // specifying .js files
        test: /\.js$/,
        // specifying loader
        loader: 'babel-loader',
        // specifying loader parameters
        query: {
          presets: ['@babel/preset-react'],
          // transpiling preset
          presets: ['@babel/preset-env', '@babel/preset-react'],
        }
      },
      // css loaders
      {
        test: /\.css$/,
        loader: `babel-loader`,
        query: {
          presets: ['style-loader', 'css-loader'],
        }
      }
    ]
  }
};
```

Improved webpack developement workflow:

1. install webpack server:

- `npm install --save-dev webpack-dev-server`

2. define npm script for server:

- `"start": "webpack-dev-server --mode=developement"`

3. add config for server

webpack.config.js:

```
const config = {
  output: ...,
  devServer: {
    contentBase: path.resolve(__dirname, 'build'),
    compress: true,
    port: 3000
  },
  // map errors to original source code
  devTool: 'source-map',
  ...
}
```

Minifying the code:

1. UglifyJS plugin automatically configured with webpack:

- significantly reduces bundled code size
- modify npm script mode:
  - `"build": "webpack --mode=production"`

Configuring backend integration (eg. server url):

```
const webpack = require('webpack');

const config =  (env, argv) => {
  const BACKEND_URL = argv.mode === 'production'
    ? '...'
    : 'localhost...';

  return {
    entry: ...,
    output: ...,
    devServer: ...,
    ...
    plugins: [
      // defining global default constraints in bundled code
      new webpack.DefinePlugin({
        // BACKEND_URL can be used directly in code
        BACKEND_URL: JSON.stringify(BACKEND_URL)
      })
    ]
  }
}
```

Class Components

- React *class* components:
  - use a constructor
    - initializes state (single object composed of multiple parts)
    - state can be set with `setState`
  - implement a render function
  - have access to React **lifecycle** methods
    - eg. `componentDidMount` is executed after first render

Class component example:

```
class App extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      anecdotes: [],
      current: 0
    };
  };

  componentDidMount = () => {
    axios.get(url).then(res => this.setState({ anecdotes: res.data }));
  }

  handleClick = () => {
    const current = Math.round(Math.random() * this.state.anecdotes.length);
    this.setState({ current });
  }

  render() {
    if (!this.state.anecdotes.length)
      return <div>no anecdotes...</div>;
    return (
      <div>
        <h1>anecdote of the day</h1>
        <div>{this.state.anecdotes[this.state.current].content}</div>
        <button onClick={this.handleClick}>next</button>
      </div>
    )
  }
}
```

vs. example as a functional component:

```
const App = () => {
  const [aneccdotes, setAnecdotes] = useState([]);
  const [current, setCurrent] = useState(0);

  useEffect(() => {
    axios.get(url).then(res => setAnecdotes(res.data));
  }, [])

  const handleClick = () => {
    setCurent(Math.round(Math.random() * aneccdotes.length));
  }

  if (!anecdotes.length)
    return <div>no anecdotes...</div>;

  return (
    <div>
      <h1>anecdote of the day</h1>
      <div>{anecdotes[current].content}</div>
      <button onClick={handleClick}>next</button>
    </div>
  )
}
```

End to End Testing

- *End-to-End* (E2E) tests inspect the entire system
  - eg. Selenium, puppeter, Cypress
- Cypress start script: `cypress open`
- for controlling the state of database during tests:
  - create router specifically for tests
  - only register the router if app is run in test mode

Cypress test examples:

```
describe('Note app', () => {
  beforeEach(() => {
    const user = {...};
    // add user to db before every test
    cy.request('POST', url, user);
    cy.visit(url);
  });

  it('front page can be opened', () => {
    cy.contains('Notes');
  });

  it('login form can be opened', () => {
    cy.contains('log in').click();
  });

  it('user can login', () => {
    cy.contains('log in').click();
    // css selectors
    cy.get('#username').type('user');
    cy.get('#password').type('pass');
    cy.contains('login').click();
    cy.contains('user logged in');
  });
})
```

Miscellaneous

- Structure Organization in a React App
- frontend can be deployed separately from backend
- options for watching for changes on server from frontend:
  - *poll* on the frontend (repeated requests to API using `setInterval`)
  - **WebSockets** establish a two-way communication bewteen browser and server
    - define callback functions when server updates state
    - Socket.io library provides fallback options if unsupported
- React uses *virtual DOM*:
  - real DOM is never directly manipulated
  - fast, only updates necessary elements on DOM change
- React deals with the *views* in Model-View-Controller (MVC) architecture
  - Flux architecture makes React even more focused on views
- application security:
  - **injection** is text sent through a form
  - **SQL-injection** maliciously modify the database with SQL queries
    - prevented by *sanitizing* the input
  - mongoose automatically santizes its queries
  - **Cross-site scripting** (XSS) injects malicious JS code into app
- current trends:
  - typed JS versions, eg. Typescript
  - *server-side* rendering allows for **Search Engine Optimization** (SEO)
  - *isomorphic* applications are rendered on both front and backend
  - *universal* applications can be executed on both front and backend
  - **Progressive Web Apps** (PWA) work on every platform
    - should work well with limited or no connections
    - offline functionality implemented with *service workers*
  - *monolithic* backend runs on a single server with a few API-endpoints
  - *microservice* architecture composes backend from separate, independent services
  - *serverless* applications use Cloud functions, easily scalable
- other libraries:
  - Immer, immutable.js for immutable data structures
  - Redux-saga alternative for thunk
  - React Google Analytics for SPA analytics
  - React Native for mobile developement

  - Parcel alternative for webpack

GraphQL

- **GraphQL** is an alternative to REST API
  - REST is *resource based*, every resource has an address
  - with GraphQL, browser makes a JSON-like query with a POST request
    - all queries are sent to the same address
    - schemas describe data sent between client and server

Schemas

```
type Person {
  // ! indicates required field
  name: String!
  phone: String
  street: String!
  city: String!
  // unique ID type (string)
  id: ID!
}

// describes what queries can be made
type Query {
  // ! indicated non-null return/parameter types

  // always returns an integer
  personCount: Int!

  // always returns list of Persons, without any null values
  allPersons: [Person!]!

  // requires string paramter, returns person or null
  findPerson(name: String!): Person
}
```

Queries and Responses

```
query {
  personCount
```

```
}

{
  "data": {
    "personCount": 3
  }
}

query {
  allPersons {
    // must describe which fields of Person to return
    name
    phone
  }
}

{
  "data": {
    "allPersons": [
      {
        "name": ...,
        "phone": ...
      },
      ...
    ]
  }
}

query {
  findPerson(name: "R2D2") {
    phone
    city
    street
    id
  }
}

{
  "data": {
    "findPerson": {
      "phone": ...,
      "city": ...,
      "street": ...,
      "id": ...
    }
  }
}

// null response
{
  "data": {
    "findPerson": null
  }
}

// combining queries
query {
  personCount
  allPersons {
    name
  }
}

{
  "data": {
    "personCount": 3,
    "allPersons": [
      { "name": ... },
      { "name": ... },
      { "name": ... }
    ]
  }
}

// renaming queries
query {
  havePhone: allPersons(phone: YES) {
    name
  }
  phoneless: allPersons(phone: NO) {
    name
  }
}
```

```
{
  "data": {
    "havePhone": [
      { "name": ... },
      { "name": ... }
    ],
    "phoneless": [
      { "name": ... }
    ]
  }
}
```

Resolvers

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

let persons = [
  {
    name: ...,
    phone: ...,
    street: ...,
    city: ...,
    id: ...
  },
  ...
];

// GraphQL schema
const typeDefs = gql`
  // schema doesn't necessarily match stored object
  type Address {
    street: String!
    city: String!
    // no id field since address not saved on server
  }

  type Person {
    name: String!
    phone: String
    address: Address!
    id: ID!
  }

  enum YesNo {
    YES
    NO
  }

  type Query {
    personCount: Int!
    // enum for selecting people with phone
    allPersons(phone: YesNo): [Person!]!
    findPerson(name: String!): Person
  }
`;

// object defining how queries are responded to
const resolvers = {
  Query: {
    personCount: () => persons.length,
    // resolvers take root/obj, args, context, info
    allPersons: (root, args) => {
      if (!args.phone) return persons
      const byPhone = (person) =>
        args.phone === 'YES' ? person.phone : !person.phone;
      return persons.filter(byPhone);
    },
    findPerson: (root, args) =>
      persons.find(p => p.name === args.name)
  }

  /* Apollo defines the following
     default resolvers for Person automatically*/
  Person: {
    name: (root) => root.name,
    phone: (root) => root.phone,
    street: (root) => root.street,
    city: (root) => root.city,
    id: (root) => root.id
  }

  // need to redefine the address resolver
  Person: {
    address: (root) => {
      return {
        street: root.street,
        city: root.city
      }
    }
  }
};

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

server.listen().then(({ url }) => {
  console.log(`Server ready at ${url}`)
})
```

Mutations

Operations that change the database are done with **mutations**:

```
const typeDefs = gql`
  ...
  type Mutation {
    // return can be null for invalid operation
    addPerson(
      name: String!
      phone: String
      street: String!
      city: String!
    ): Person
    editNumber(
      name: String!
      phone: String!
    ): Person
  }
`


const resolvers = {
  ...
  Mutation: {
    addPerson: (root, args) => {
      // validating unique name
      if (person.find(p => p.name === args.name)) {
        throw new UserInputerror('Name must be unique', {
          invalidArgs: args.name
        });
      }
      const person = { ...args, id: uuid() };
      persons = persons.concat(person);
      return person;
    },
    editNumber: (root, args) => {
      const person = persons.find(p => p.name === args.name);
      if (!person) return null;
      const updatedPerson = { ...args, phone: args.phone };
      persons = person.map(p => p.name === args.name ? updatedPerson : p);
      return updatedPerson;
    }
  }
}
```

Adding a Person with the mutation:

```
mutation {
 addPerson(
  name: "R2D2"
  street: "La Brea"
  city: "Tatooine"
 ) {
  name
  phone
  address {
    city
    street
  }
  id
 }
}
```

Saved object on the server:

```
{
  name: "R2D2",
  street: "La Brea",
  city: "Tatooine",
  id: "123-234-123-123123"
}
```

Response to the mutation:

```
{
  "data": {
    "addPerson": {
      "name": "R2D2",
      "phone": null,
      "address": {
        "city": "Tatooine",
        "street": "La Brea"
      },
      "id": "123-234-123-123123"
    }
  }
}
```

Frontend

- GraphQL query is a string sent as value of key *query*
- higher order library instead of Axios: Relay or Apollo Client
  - Apollo Client automatically saves queries to *cache* by ID
    - as a result, new objects are not updated to state (but existing objects are)
  - to update the cache:
    - poll server repeatedly:
    - `<Query query={ALL_PERSONS} pollInterval={2000}>`
    - synchronize queries:
    - `<Mutation mutation={CREATE_PERSON} refetchQueries={[{ query: ALL_PERSONS }]}`
  - to clear the cache: (eg. on logout)
    - `const client = useApolloClient()` , `client.resetStore()`
- react-apollo integrates queries with react components

Using Apollo Client and react-apollo:

```
import ApolloClient, { gql } from 'apollo-boost';
import { ApolloProvider } from 'react-apollo;'

```

```
const client = new ApolloClient({ uri: 'https://localhost:4000/graphql' });

const query = gql`
{
  allPersons {
    name,
    phone,
    address {
      street,
      city
    },
    id
  }
}
`

client.query({ query }).then(res => console.log(res.data));

ReactDOM.render(
  <ApolloProvider client={client}>
    <App />
  </ApolloProvider>,
  document.getElementById('root')
);
```

Using the Query component:

```
import { Query } from 'react-apollo';

const ALL_PERSONS = gql`
{
  allPersons {
    name,
    phone,
    id
  }
}
`

const App = () => {
  return (
    <Query query={ALL_PERSONS}>
      {(result) => <Persons result={result} />}
    </Query>
  );
}

const Persons = ({ resut }) => {
  // as query is processing
  if (result.loading) {
    return <div>loading...</div>;
  }

  const persons = result.data.allPersons;

  return (
    <div>
      <h2>Persons</h2>
      {persons.map(p => (
        <div key={p.name}>
          {p.name} {p.phone}
        </div>
      ))}
    </div>
  );
}
```

Using GraphQL variables for dynamic parameters: (ApolloConsumer component gives access to the client's query method)

```
import { Query, ApolloConsumer }

const App = () => {
  return (
    <ApolloConsumer>
      {(client) =>
        <Query ... />
      }
    </ApolloConsumer>
  );
}

const FIND_PERSON = gql`
  query findPersonByName($nameToSearch: String!) {
    findPerson(name: $nameToSearch) {
      name,
      phone,
      id,
      address{
        street,
        city
      }
    }
  }
`

const Persons = ({ result, client }) => {
  const [person, setPerson] = useState(null);
  ...
  const showPerson = async (name) => {
    const { data } = await client.query({
      query: FIND_PERSON,
      variables: { nameToSearch: name }
    });
    setPerson(data.findPerson);
  };

  if (person) {
    return (
      <div>
        <h2>{person.name}</h2>
        <div>{person.address.street} {person.address.city}</div>
        <div>{person.phone}</div>
        <button onClick={() => setPerson(null)}>close</button>
      </div>
    );
  }

  return (
    ...
    <button onClick={() => showPerson(p.name)}>show address</button>
    ...
  )
}
```

Using the Mutation component:

```
const CREATE_PERSON = gql`
  mutation createPerson($name: String!, $street: String!,
              $city: String!, $phone: !string) {
    addPerson(
      name: $name,
      street: $street,
      city: $city,
      phone: $phone,
    ) {
      name,
      phone,
      id,
      address {
        street,
        city
      }
    }
  }
`

const App = () => {
  // error handling
  const handleError = (err) => {
    console.log(error.graphQLErrors[0].message);
  };

  return (
    ...
    <Mutation mutation={CREATE_PERSON} onError={handleError}>
      {(addPerson) => <PersonForm addPerson={addPerson}/>}
    </Mutation>
  );
}

const PersonForm = (props) => {
  ...
  const submit = async (e) => {
    e.preventDefault();
    await props.addPerson({
      variables: { name, phone, street, city }
    });
    ...
  };
}
```

Render-Props vs. Hooks

- the **render-props** principle:
  - where components are given a function defining how the component is rendered
  - eg. React router Route component and corresponding render function
  - eg. ApolloConsumer and Query components

Using hooks with Apollo Client: (offered in react-apollo@3.0.0-beta.2)

```
import { ApolloProvider } from '@apollo/react-hooks';
...
ReactDOM.render(
  <ApolloProvider client={client}>
    <App />
  </ApolloProvider>,
  document.getElementById('root')
);

import { useApolloClient } from '@apollo/react-hooks';

const Persons = ({ result }) => {
  const client = useApolloClient();
  ...
}

import { useQuery, useMutation } from '@apollo/react-hooks';

const App = () => {
  const persons = useQuery(ALL_PERSONS);

  // array: mutation function, loading/error obj
  const [addPerson] = useMutation(CREATE_PERSON, {
    onError: handleError,
    refetchQueries: [{ query: ALL_PERSONS }]
  });

  const [editNumber] = useMutation(EDIT_NUMBER);
  ...
  <Persons result={persons} />
  <PersonForm addPerson={addPerson} />
  <PhoneForm editNumber={editNumber} />
  ...
}
```

Database

- to use Apollo with a *database*:
  - create a corresponding schema to the type definition
  - update the resolver definitions
    - when resolver functions return a promise, Apollo automatically sends back resolved promise

Apollo with MongoDB:

```
const schema = new mongoose.Schema({
  name: {
    type: String,
    required: true,
    unique: true,
    minlength: 5
  },
  ...
});

module.exports = mongoose.model('Person', schema);

const typeDefs = ...

const resolvers = {
  Query: {
    personCount: () => Person.collection.countDocuments(),
    allPersons: (root, args) => {
      // optional filter people with numbers arg
      if (!args.phone) return Person.find({});

      return Person.find({ phone: { $exists: args.phone === 'YES' }});
    },
    findPerson: (root, args) => Person.findOne({ name: args.name })
  },
  Person: {
    address: root => {
      return {
        street: root.street,
        city: root.city
      };
    }
  },
  Mutation: {
    // returning a promise in the resolver
    addPerson: (root, args) => {
      const person = new Person({ ...args });

      // validating mongoose schema
      try {
        await person.save();
      } catch(err) {
        // Apollo error
        throw new UserInputError(err.message, {
          invalidArgs: args
        });
      }
      return person;
    },
    editNumber: async (root, args) => {
      const person = await Person.findOne({ name: args.name });
      person.phone = args.phone;

      try {
        await person.save();
      } catch (err) {
        throw new UserInputError(err.message, {
          invalidArgs: args
        });
      }
      return person;
    }
  }
}
```

User Administration

- setting up user validation with Apollo and MongoDB in backend

Schema:

```
// User mongoose schema
const schema = new mongoose.Schema({
  username: {
    type: String,
    required: true,
    unique: true,
    minlength: 3
  },
  friends: [
    {
      type: mongoose.Schema.Types.ObjectId,
      ref: 'Person'
    }
  ]
});
module.exports = mongoose.model('User', schema);

// User Apollo schema
type User {
  username: String!
  friends: [Person!]!
  id: ID!
}

type Token {
  value: String!
}

type Query {
  ...
  me: User
}

type Mutation {
  ...
  createUser(username: String!): User
  login(username: String!, password: String!): Token
}
```

Updated mutation resolvers:

```
const resolvers = {
  Mutation {
    createUser: (root, args) => {
      const user = new User({ username: args.username });
      return user.save().catch(err => ...)
    },
    login: async (root, args) => {
      const user = await User.findOne({ username: args.username });
      if (!user || args.password !== 'pass') {
        throw new UserInputError('wrong credentials');
      const userToken = {
        username: user.username,
        id: user._id
      };
      return { value: jwt.sign(userToken, SECRET_KEY) };
    }
  }
}
```

Updated constructor and actions with context:

```
const server = new ApolloServer({
  typeDefs,
  resolvers,
  // context is given to all resolver as 3rd parameter
  // use context for shared resolver data
  context: async ({ req }) => {
    const auth = req ? req.headers.authorization : null;
    if (auth && auth.toLowerCase().startsWith('bearer ')) {
      const decoded = jwt.verify(auth.substring(7), SECRET_KEY);
    }
    const currentUser = await User.findById(decoded.id).populate('friends');
    return { currentUser };
  }
});

// Query resolver
Query: {
  ...
  me: (root, args, context) => context.currentUser
}

// authenticated actions
type Mutation {
  ...
  addAsFriend(name: String!): User
}
```

```
addAsFriend: aync (root, args, { currentUser }) => {
  const nonFriendAlready = (person) =>
    !currentUser.friends.map(f => f._id).includes(person._id);

  if (!currentUser) {
    throw new AuthenticationError("not authenticated");
  }

  const person = await Person.findOne({ name: args.name });
  if (nonFriendAlready(person)) {
    currentUser.friends = currentUser.frieds.concat(person);
  }

  await currentUser.save();
  return currentUser;
}
```

User Administration on the Frontend

Saving token on login success:

```
const LoginForm = (props) => {
  ...
  const submit = async (e) => {
    e.preventDefault();

    const res = await props.login({ variables: { username, password }});

    if (res) {
      const token = res.data.login.value;
      // saved in root App component
      props.setToken(token);
      // saved in local storage
      localStorage.setItem('phonenumbers-user-token', token);
    }
  };
  ...
}
```

Clearing storage and cache on logout:

```
const App = () => {
  const client = useApolloClient();
  ...
  const logout = () => {
    setToken(null);
    localStorage.clear();
    client.resetStore();
  };
  ...
}
```

Automatically adding tokens to headers:

```
// using apollo-client instead of apollo-boost for custom configuration
import { ApolloClient } from 'apollo-client';
import { createHttpLink } from 'apollo-link-http';
import { InMemoryCache } from 'apollo-cache-inmemory';
import { setContext } from 'apollo-link-context';

const httpLink = createHttpLink({ uri: ... });

const authLink = setContext((_, { headers }) => {
  const token = localStorage.getItem('phonenumbers-user-token');
  return {
    headers: {
      ...headers,
      authorization: token ? `bearer ${token}` : null
    }
  };
});

const client = new ApolloClient({
  // how client contacts the server
  // httpLink and custom token in header
  link: authLink.concat(httpLink),
  // using cache in main memory
  cache: new InMemoryCache()
});
```

Alternative for updating cache:

```
const [addPerson] = useMutation(CREATE_PERSON, {
  onError: handleError,
  // query always rerun with any updates
  // refetchQueries: [{ query: ALL_PERSONS }]

  // manually updating cache
  update: (store, res) => {
    const dataInStore = store.readQuery({ query: ALL_PERSONS });
    dataInStore.allPersons.push(res.data.addPerson);
    store.writeQuery({
      query: ALL_PERSONS,
      data: dataInStore
    });
  }
})
```

Fragments and Subscriptions

- often useful to define **fragments** for selecting fields
  - fragments are defined in the client, *not* the GraphQL schema itself

Using fragments to automatically grab all fields:

```
const PERSON_DETAILS = gql`
  fragment PersonDetails on Person {
    id
    name
    phone
    address {
      street
      city
    }
  }
`

const ALL_PERSONS = gql`
  {
    allPersons {
      ...PersonDetails
    }
  }
  ${PERSON_DETAILS}
`
```

- GraphQL **subscription** is another operation type (query, mutation)
  - clients can *subscribe* to changes in the server
  - under the hood, Apollo uses WebSockets for this subscriptions
- communication uses the *publish-subscribe* principle with a PubSub interface:
  - adding a new object *publishes* a notification about the operation with `publish`
  - the subscription resolver registers all subscribers by returning them an *iterator* object
- the *n+1* problem appears in database querying:
  - when attempting to load the children of a parent relationship
  - querying the database repeatedly, n+1 times
- solution usually involves using join queries:
  - eg. can use MongoDB join query to populate child fields
  - check query info to only do join queries for n+1 problem queries
    - minimizes execution when query does not raise an n+1 problem

Setting up subscriptions on the server:

```
// updated schema:
type Subscription {
  // when a new person is added,
  // its details are sent to all subscribers
  personAdded: Person!
}

// updated resolvers:
const { PubSub } = require('apollo-server');
const pubsub = new PubSub();

const resolvers = {
  ...
  Mutation: {
    addPerson: async (root, args, context) => {
      ...
      pubsub.publish('PERSON_ADDED', { personAdded: person });
      return person;
    }
  },
  Subscription: {
    personAdded: {
      subscribe: () => pubsub.asyncIterator(['PERSON_ADDED'])
    }
  }
}
```

```
// updated server start to listen for subscriptions:
server.listen().then(({ url, subscriptionsUrl }) => {
  console.log(`Server ready at ${url}`);
  // different url
  console.log(`Subscriptions ready at ${subscriptionsUrl}`);
})
```

Using subscriptions on the frontend: (requires subscriptions-transport-ws and apollo-link-ws)

```
import { split } from 'apollo-link';
import { WebSocketLink } from 'apollo-link-ws';
import { getMainDefinition } from 'apollo-utilities';

// requires websocket as well as HTTP connection
const wsLink = new WebSocketLink({
  uri: ...,
  options: { reconnect: true }
});


...


const link = splilt(
  // splits to different link depending on operation
  ({ query }) => {
    const { kind, operation } = getMainDefinition(query);
    return kind === 'OperationDefinition' && operation === 'subscription';
  },
  wsLink,
  authLink.concat(httpLink)
);

const client = new ApolloClient({
  link,
  cache: new InMemoryCache()
});
```

Using subscriptions with hooks:

```
import { useSubscription } from '@apollo/react-hooks';

const PERSON_ADDED = gql`
  subscription {
    personAdded {
      ...PersonDetails
    }
  }
  ${PERSON_DETAILS}
`;

const App = () => {
  ...
  useSubscription(PERSON_ADDED, {
    onSubscripionData: ({ subscriptionData } => {
      console.log(subscriptionData);
    })
  })
  ...
}
```

Updating cache with subscription:

```
const App = () => {
  ...
  const updateCacheWith = (addedPerson) => {
    const includedIn = (set, object) => {
      set.map(p => p.id).includes(object.id);
    }

    const dataInStore = client.readQuery({ query: ALL_PERSONS });
    if (!includedIn(dataInStore.allPersons, addedPerson)) {
      dataInStore.allPersons.push(addedPerson);
      client.writeQuery({
        query: ALL_PERSONS,
        data: dataInStore
      });
    }
  };

  useSubscription(PERSON_ADDED, {
    onSubscripionData: ({ subscriptionData }) => {
      const addedPerson = subscriptionData.data.personAdded;
      notify(`${addedPerson.name} added`);
      updateCacheWith(addedPerson);
    }
  });
  const [addPerson] = useMutation(CREATE_PERSON, {
    onError: handleError,
    update: (store, res) => {
      updateCacheWith(res.data.addPerson);
    }
  });
  ...
}
```