MERN Stack Transformation Workbook

Chapter 1: From Static to Stack - Your MERN Makeover Begins!

What You'll Learn in This Chapter:

- What the MERN stack is (and why it's powerful)
- How to prepare your tools
- How to scaffold a React project with Vite
- How to think like a component-based developer

What is the MERN Stack?

MERN stands for:
M - MongoDB: A NoSQL database to store your data.
E - Express.js: A lightweight web framework for Node.js.
R - React: A JavaScript library for building user interfaces.
N - Node.js: A runtime for running JavaScript on the server.

Why MERN? It allows you to use JavaScript everywhere - frontend and backend. Great for solo developers and small teams.

Tools You'll Need:

Make sure you've installed the following:
- Node.js (comes with npm)
- A code editor (VS Code is recommended)
- Git & GitHub (optional, but useful)
- MongoDB Atlas (we'll use this later)

Write-in Area:
Node version:
VS Code extensions I like:
MongoDB setup complete? (Y/N):

_____

_____

_____

_____

Create a New React Project

Open your terminal and run the following commands:
npm create vite@latest my-app-name -- --template react
cd my-app-name
npm install
npm run dev

MERN Stack Transformation Workbook

This starts your app at http://localhost:5173. You should see the Vite splash screen.

Understand Your Project Structure

Here's a quick tour of the new React app:
- public/: Static assets
- src/: Your React components and logic
  - App.jsx: Your main component
  - main.jsx: Entry point for React
- index.html: Your HTML shell
- package.json: Project metadata and dependencies

Activity: Reimagine Your HTML as Components

Example HTML:

```
<header>
  <h1>Welcome to My Portfolio</h1>
</header>
```

Becomes React:

```
function Header() {
  return (
    <header>
      <h1>Welcome to My Portfolio</h1>
    </header>
  );
}
```

Sketch your site as components:
- Header
- _____
- _____
- _____

_____

_____

_____

_____

Challenge: Move Your Homepage Into React

Take your current homepage and:
1. Paste the HTML into App.jsx
2. Replace the default content
3. Update class to className
4. Begin breaking it into components

MERN Stack Transformation Workbook

Reflection

What went well?

_____

_____

_____

What was confusing?

_____

_____

_____

What do I want to learn more about next?

_____

_____

_____

Chapter Summary

In this chapter, you:
- Learned the structure and purpose of the MERN stack
- Installed your development tools
- Created a React app using Vite
- Reimagined your static site using components
- Took the first step toward converting your site into a MERN app

MERN Stack Transformation Workbook

Chapter 2: Break It Down - Components, Props, and Hooks

What You'll Learn in This Chapter:

- Break a React page into reusable components
- Pass data between components using props
- Use basic hooks: useState and useEffect
- Refactor your original HTML for maintainability

Why Components Matter

React apps are built using components. Think of each component as a building block. They can be reused and customized, which keeps your code clean and organized.

A typical website might have a header, navbar, main content, and footer. Each of those can be its own component.

Breaking Down a Page

Here's how you might break a portfolio site into components:
- <Header />
- <Navbar />
- <MainContent />
- <Footer />

Write down your page structure as components:

_____

_____

_____

_____

_____

Passing Props Between Components

Props (short for properties) let you pass data into components. Here's an example:

Parent Component:

```
function App() {
  return <Welcome name='Sam' />;
}
```

Child Component:

```
function Welcome(props) {
  return <h1>Hello, {props.name}!</h1>;
}
```

Try it yourself: What would you pass into a component on your site?

MERN Stack Transformation Workbook

useState: Managing Local State

The useState hook lets your component hold onto information.
Example:
const [count, setCount] = useState(0);
Then use setCount to update the value.

useEffect: Reacting to Changes

useEffect runs code after your component renders. It's great for fetching data or watching changes.
Example:

```
useEffect(() => {
  console.log('Component mounted');
}, []);
```

Activity: Refactor Your Page

Take your current React page and do the following:
- Move your header and footer into components
- Pass at least one prop to a component
- Add useState to store something like a toggle or form input

Reflection

Which parts did you successfully refactor?

What challenges did you face?

What's something you'd like to improve next time?

Chapter Summary

MERN Stack Transformation Workbook

In this chapter, you:
- Learned how to divide your UI into components
- Passed props from parent to child components
- Used useState to manage local data
- Used useEffect to run code after rendering
- Started making your app more dynamic and reusable

MERN Stack Transformation Workbook

Chapter 3: Build the Backend - Express & MongoDB

What You'll Learn in This Chapter:

- Create a backend server using Express
- Connect to a MongoDB database
- Create API routes (GET, POST)
- Test routes with a frontend or API tool

What Is Express?

Express is a lightweight web framework for Node.js. It lets you define routes and serve data through an API. Instead of HTML files, it sends back JSON data that your frontend can use.

Setting Up Your Backend

Create a folder called 'server' inside your project (but outside your React app).

In the terminal:
mkdir server
cd server
npm init -y
npm install express cors mongoose dotenv

Basic Express Server

In server/index.js:

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

app.use(cors());
app.use(express.json());

app.get('/', (req, res) => {
  res.send('Hello from the backend');
});

app.listen(5000, () => console.log('Server running on port 5000'));
```

Connecting to MongoDB

1. Go to mongodb.com and create a cluster.
2. In .env file, add:
MONGO_URI=your_connection_string_here
3. Use mongoose in index.js:
mongoose.connect(process.env.MONGO_URI)

MERN Stack Transformation Workbook

```
  .then(() => console.log('MongoDB connected'))
  .catch(err => console.error(err));
```

Create Your First API Route

Let's build a 'notes' route:

```
app.get('/notes', (req, res) => {
  res.json([{ id: 1, text: 'First note' }]);
});
```

Try it: What kind of data does your app need to store?

Test Your API

You can test your backend using:
- Your React frontend (with fetch or axios)
- A tool like Postman or Thunder Client (VS Code extension)

Activity: Create a New Route

Build a GET and POST route for a resource your app needs. For example:
- Guestbook entries
- Portfolio projects
- Messages or comments

What routes did you build?

Reflection

What went well?

What was confusing?

MERN Stack Transformation Workbook

What do I want to learn more about next?

_____

_____

_____

Chapter Summary

In this chapter, you:
- Set up an Express server
- Connected to MongoDB
- Built basic GET and POST routes
- Tested your API using tools or React frontend
- Began structuring your full-stack application

MERN Stack Transformation Workbook

Chapter 4: Connect the Dots - Frontend Meets Backend

What You'll Learn in This Chapter:

- Connect your React frontend to your Express backend
- Use fetch or axios to call your API
- Display backend data in your components
- Handle loading and errors in the UI

Making the Connection

Now that you have a backend server, it's time to bring your frontend and backend together.
Your React app can make HTTP requests to your Express API to get and send data.

Using fetch in React

Here's how you can use fetch to get data from your backend:

```
useEffect(() => {
  fetch('http://localhost:5000/notes')
    .then(res => res.json())
    .then(data => setNotes(data))
    .catch(err => console.error(err));
}, []);
```

Handling State

You can store your fetched data using useState:

```
const [notes, setNotes] = useState([]);
```

Displaying the Data

Inside your component:

```
{notes.map(note => (
  <div key={note.id}>{note.text}</div>
))}
```

Try it: What endpoint will you fetch in your app?

Sending Data with POST

Example using fetch to send a new note:

```
fetch('http://localhost:5000/notes', {
  method: 'POST',
```

```
  headers: { 'Content-Type': 'application/json' },
  body: JSON.stringify({ text: 'New note' })
})
```

What form or input will send data in your app?

_____

_____

_____

Activity: Build a Full Connection

Use your frontend to:
- Fetch a list of items from your backend
- Display them in a component
- Add a form to POST a new item to the backend

Reflection

What worked when connecting frontend to backend?

_____

_____

_____

What didn't go as expected?

_____

_____

_____

What would you like to build next?

_____

_____

_____

Chapter Summary

In this chapter, you:
- Connected your React app to your Express server
- Used fetch to GET and POST data
- Displayed backend data in your frontend
- Managed loading state and error handling
- Completed your first full-stack feature!

MERN Stack Transformation Workbook

Chapter 5: Lock It Down - User Authentication with JWT

What You'll Learn in This Chapter:

- Create user registration and login routes
- Store hashed passwords securely
- Issue and verify JWT tokens
- Protect backend routes with authentication

Why Authentication?

Authentication lets users securely log in and access their data. With JWT (JSON Web Tokens), your server can verify a user's identity on every request-without storing sessions.

Install Dependencies

In your server directory:
npm install bcryptjs jsonwebtoken

Set Up User Model

Create models/User.js:

```
const mongoose = require('mongoose');
const UserSchema = new mongoose.Schema({
  email: { type: String, required: true, unique: true },
  password: { type: String, required: true }
});
module.exports = mongoose.model('User', UserSchema);
```

Create Register Route

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

app.post('/register', async (req, res) => {
  const { email, password } = req.body;
  const hashedPassword = await bcrypt.hash(password, 10);
  const user = await User.create({ email, password: hashedPassword });
  res.json(user);
});
```

Create Login Route

```
app.post('/login', async (req, res) => {
  const { email, password } = req.body;
  const user = await User.findOne({ email });
  if (!user) return res.status(404).json('User not found');
```

```
  const isMatch = await bcrypt.compare(password, user.password);
  if (!isMatch) return res.status(401).json('Invalid password');

  const token = jwt.sign({ id: user._id }, process.env.JWT_SECRET);
  res.json({ token });
});
```

Protect a Route

```
function authMiddleware(req, res, next) {
  const token = req.headers.authorization?.split(' ')[1];
  if (!token) return res.sendStatus(401);
  jwt.verify(token, process.env.JWT_SECRET, (err, user) => {
    if (err) return res.sendStatus(403);
    req.user = user;
    next();
  });
}
```

What routes or features do you want to protect in your app?

___

___

___

Activity: Add Auth to Your App

- Create a register form in React
- Add a login form that stores the JWT
- Protect a component or route based on auth state

Reflection

What was easy about setting up authentication?

___

___

___

What was hard or confusing?

___

___

___

What kind of user experience do you want to improve?

___

Chapter Summary

In this chapter, you:

- Created a secure user model with hashed passwords
- Built register and login routes
- Used JWT to issue authentication tokens
- Added middleware to protect routes
- Started creating a secure, user-specific experience

MERN Stack Transformation Workbook

Chapter 6: Go Live! Deploying Your MERN App

What You'll Learn in This Chapter:

- Deploy your frontend to Netlify or Vercel
- Deploy your backend to Render or Railway
- Use environment variables safely
- Connect frontend to backend in production

Frontend Deployment: Vercel or Netlify

To deploy your React frontend:

1. Push your code to GitHub
2. Go to vercel.com or netlify.com
3. Import your project from GitHub
4. Set build command to `npm run build`
5. Set output directory to `dist` (Vite default)

Deployment platform chosen: _____

Backend Deployment: Render or Railway

1. Push your `server` folder to a new GitHub repo
2. Go to render.com or railway.app
3. Connect your repo and follow the setup steps
4. Set your server port and MongoDB URI in the environment settings
5. Set the start command to `node index.js` or `npm start`

Deployed backend URL: _____

Connect Frontend to Backend

Change API base URLs in your React app:
Instead of:
`http://localhost:5000/notes`
Use your live backend URL:
`https://your-backend.onrender.com/notes`

Store URLs in `.env`:
VITE_API_URL=https://your-backend-url

Access it in your code:
`import.meta.env.VITE_API_URL`

MERN Stack Transformation Workbook

Test in Production

- Test GET and POST requests from the live site
- Confirm login and registration still work
- Fix CORS issues by updating your backend with:
`app.use(cors({ origin: 'https://your-frontend.com' }))`

Activity: Go Live!

- Deploy both frontend and backend
- Connect them together using env variables
- Test all your features on the live version

What worked smoothly?

_____

_____

What did you have to troubleshoot?

_____

_____

_____

Reflection

What does 'done' look like for this project?

_____

_____

_____

What would you like to build or improve next?

_____

_____

_____

Chapter Summary

In this chapter, you:
- Deployed your frontend and backend to the web
- Used environment variables for security
- Updated API calls to use live URLs
- Troubleshooted CORS and connection issues
- Took your MERN project live for the world to see!