

Scaling Real-World Programming with ChatGPT

Lambda Prompt

www.lambdaprompt.com

Morten Heine Sørensen

mhs@formalit.dk

Mark Hissink Muller

mark@mhmconsult.dk

ABSTRACT

The software development community is discussing how ChatGPT and related new AI-technologies can help developers. Day by day, new examples, experiments and experiences emerge. Most authors acknowledge ChatGPT as a useful tool that improves productivity, but they also point out limitations and suggest that ChatGPT belongs in the *useful tool* category, not the *new era* category.

On the other hand, there are reports on LinkedIn about developers able to do the work of multiple persons in parallel using ChatGPT. In this paper we argue that this is indeed a new era and suggest a productivity improvement of a factor 2 to 5. We report on several experiments we did, resulting in 25 lessons learned which we believe support the claim, and outline the approach we have created along the way.

BACKGROUND

We review some of the experiments already reported. Gamage[1] uses ChatGPT to generate code snippets (e.g., a React component), write test cases, and fix defects. He concludes that “ChatGPT’s responses are not always 100% accurate, so you need to do thorough research before using it for more advanced tasks.”

Gewirtz[5] uses ChatGPT to write a WordPress plugin. He also attempts to generate an entire application from an elaborate prompt of requirements. This results in a “flood of text and code,” but “none of it was usable.” He finds that “unless you’re writing a very small function [...], ChatGPT isn’t going to be able to write your final code.”

Another attempt at developing a full application is due to Guo[8] who develops part of a browser plugin to play music that matches the publication date of various research papers [1]. He uses ChatGPT to do the initial setup of a Chrome extension, and notices that “wrestling with all these finicky details [...] before I could begin any real coding felt like death by a thousand cuts.” Having overcome this, he co-designs the application with ChatGPT as a “UX design consultant.”

Finally, he proceeds to link the extension with the Spotify API and comments that “the gold standard here is an

expert human programmer who has a lot of experience with the Spotify API and who is good at teaching novices. ChatGPT was alright for getting me started but ultimately didn’t meet this standard” due to lack of context, bird’s eye view, tacit knowledge and looking ahead.

For a last example, Fowler[3] comments on development where Xu Hao adds an awareness feature to an existing Online Whiteboard application based on React [3]. He begins with a detailed description of the existing system, its tech stack, implementation design, some patterns to be followed, the requirements for the new feature and asks ChatGPT for a plan for the implementation.

ChatGPT responds with the plan, Xu Hao asks for the code for each step, to which ChatGPT responds with code.

Fowler observes that the plan is useful to establish context when ChatGPT loses it and concludes that “it’s useful to interact with the LLM like a junior partner, starting them with architectural guidelines, asking them to show their reasoning, and tweaking their outputs as we go.”

Loukides[4] comments that until the appearance of the Fowler paper, “I haven’t seen anyone demonstrate what it takes to do professional development with ChatGPT.”

He speculates that the approach yields a “25 to 50% reduction in the time it would take to generate the code, not 90%.”

He also comments that “after developing a significant application with ChatGPT, what do you have? A body of source code [...] that nobody understands in depth. [...] It’s similar to software that was written 10 or 20 or 30 years ago, by a team whose members no longer work at the company, but that needs to be maintained, extended, and (still) debugged.”

In summary, results so far are promising, but do not report a revolution in how we develop software.

THIS PAPER

In the paper we conversely argue that this marks the beginning of a new era.

We present an approach to developing full stack applications with ChatGPT, in a manner which we believe can drastically reduce the development time and cost by an estimated factor between 2 and 5.

The approach has been tried out with development of three realistic examples, which contain:

- UI built with React.
- API built on Node.js or Java with Spring Boot.
- Data layer with Postgres database.

In addition, devOps was covered this way:

- UI hosted as Azure static web app.
- API hosted as Azure app service.
- Code in GitHub.
- Automatic deployment on merge of each PR.

In our first example, an application was developed from scratch up to production level quality in very short time. The resulting code is well-structured and can subsequently be maintained following the same approach, or by developers that have been part of the process.

In the second example, an existing application with high quality was taken over by an experienced developer, who had limited experience with the technology stack, and the functionality was quickly changed to a different scope.

In a final third experiment the emphasis was on picture-perfect UI.

We first explain the examples and the overall approach, then break it into smaller steps and aspects that can be followed one by one. Each of the sections summarize at least one important lesson that was learned when working with the examples, which may be of general interest.

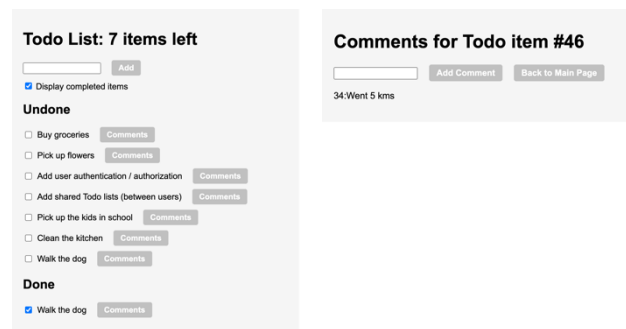
1: THE EXAMPLES

We briefly review the examples, which were implemented using ChatGPT4 in June - July 2023.

Example 1

The first example is the Todo application of [1] extended with an additional page to write comments for a todo, extending the example with routing and state management across pages.

The focus of the example was to drastically reduce development time for the API based on Node.js and for a general structure for React applications dealing with state management, call-backs, calling APIs, etc. Aesthetics of the UI was not a priority.



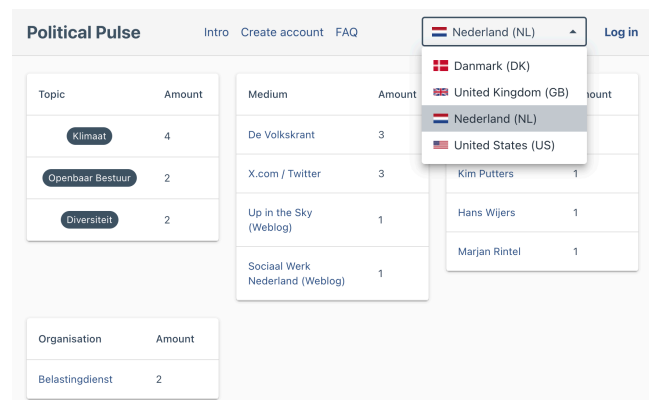
Automatic API testing and UI testing (using Puppeteer) were used to validate code generated by ChatGPT.

Example 2

In the second example, an existing React application with a backend in Java was modified to incorporate several architectural and functional changes. Also, we delivered new end-to-end new application functionality using ChatGPT's help.

The goal of this example was to demonstrate how a developer with limited experience in React may quickly reach an acceptable productivity level. Also, the point was to show that ChatGPT may help maintain applications, not just develop them from scratch.

The resulting application is a multi-tenant social media application allowing people to be involved in the democratic process.



The application's entities are persisted using JPA in a Postgres database managed by Liquibase. REST APIs are implemented by a controller layer (where authentication is handled and access is restricted by role), which accesses core services that contain the business logic, which in turn access repositories that are responsible for data access, in a way common for a Spring Boot application. The application leverages a bean mapper component to limit the entity properties exposed via REST.

Example 3

In this example, the focus was on producing picture-perfect UI, i.e., styling with e.g., CSS, that would match an existing eCommerce application in production to show that also this aspect can be covered by ChatGPT.

2: OVERALL APPROACH

The approach can be briefly explained as follows. Imagine you are senior lead developer for a team of, say, five developers, excluding yourself. The developers are junior, medio or senior and front-end, back-end or full-stack developers.

Imagine you actively manage the team by providing functional requirements to each individual developer where work is organized in sprints. For the UI development it is in the form of screen shots (e.g., Figma) and user stories. For the DB layer, it will be by specifying tables and columns with primary and foreign keys. This carries over to the CRUD part of the API layer, which may also include, among others, validations, business logic, calculations and authentication and authorization.

Also imagine that the application has a skeleton which expresses how to deal consistently with common themes. For the UI, it will be how to manage state, where to place call-back handlers, how to call APIs, etc. For the API it will be, e.g., your selected separation into controller, service, and repo layers and how to manage ORM directly or by 3rd party libraries. The skeleton may come from you or the developers (or as the first step of the approach).

From this point on you delegate features to the developers. You review each pull request (PR) from each developer, possibly asking for revision of the code, e.g., to ensure consistency and completeness. After approving PRs, the API test and UI tests are automatically run and any errors are identified and communicated to the developers, who will fix them and start another cycle of the loop. The tests are extended by the developers to cover their new functionality.

Now take away the five developers and add ChatGPT instead (don't worry, they return later in the paper). The prompts correspond to the delegation to developers, and the generated answers make up the PRs.

In the following sections we go into more details with the approach by outlining individual steps and aspects.

3: SET UP THE PROJECT

The first step is to set up the local development environment, which includes the basic file structure of the application and how you build and run the application.

In our approach we assume that the lead developer is an experienced senior. If he is already expert in the tech stack, he will be capable of such tasks. If he is not, ChatGPT can help him become enough of an expert in short time.

LESSON 1: ChatGPT can provide tutorials on how to accomplish tasks that are tailored to your specific application, making them very easy and fast to use.

In any event, this step is usually required only once for a range of sprints leading to one or more applications in the same company, so it does not have to be super-efficient.

The local development environment must be efficient to work with because it is crucial to have a fast loop of getting pull requests from ChatGPT, testing them with the UI and API tests, and identifying and fixing issues.

In Example 1, we used a mixture of ChatGPT and [1] to provide basic setup of IDE and project files. There were a couple of issues with versions and dependencies, but they were not difficult to fix. We set up the code in a GitHub repository, which made it easy to backtrack when ChatGPT produced undesirable code.

LESSON 2: Spend the time needed to set up the local development environment to efficiently support the iterations with ChatGPT. It does not matter if this take a bit more time than ideal since it will be a one-time cost.

4: GET STARTED WITH DEVELOPMENT

Fowler[2] celebrates an approach, where you first communicate to ChatGPT an overall plan of steps that should be carried out before going into details.

The way we see it, this plan may reside in the mind of the team lead, who delegates parts to the individual developers without each of them necessarily knowing the entire plan.

For instance, for Example 1, we could start with the code to generate the relevant database tables, then create the needed API, and finally create the UI.

In fact, we recommend following the three steps in this order (DB, API, UI) for one feature, and then repeat them for each new feature. There are many reasons why this iterative development is a good idea in conventional projects without ChatGPT, and they carry over to the new way of developing software with ChatGPT. For instance, it avoids the useless code Gewirtz[5] experienced.

LESSON 3: Split the development into sprints and user stories, like you would with a team of developers. Then

proceed sprint by sprint, user story by user story, for the same reasons as you normally do.

LESSON 4: In Sprint 1, establish the fundamental architecture of the application, for instance a UI layer, an API layer, and the DB layer developed well enough to cover a small feature. The fundamental architecture should not only align to functional and non-functional requirements, but also to the team size and team experience.

5: CREATE THE DATA MODEL

In a conventional project we would recommend developing a feature by doing the UI and API simultaneously, because each aspect helps to validate the other and together, they deliver a complete feature. The DB part can probably be done by the API developer, both are part of a typical back-end profile's capabilities.

With ChatGPT and a single development lead, the separate parts UI, API, DB are sequential, and we recommend doing them in the order DB, API, UI, since the UI needs the API, which needs the DB.

So, first you generate the data model. Either you already know the entire data model, or you know only part of it. In any case, as already mentioned, you should start with one or two example entities and create the DDL or migrations, depending on your preferred approach, to confirm that you get the expected results.

Some people recommend postponing the DB layer, for various reasons. For instance, it is argued that the DB layer can change from relational to NoSql. Also, it is emphasized that it is important to solicit quick feedback from product management, which suggests emphasizing the UI.

However, in our experience most applications have a data model, relational or otherwise, and understanding this model is the key to have a solid basis for the entire application.

LESSON 5: Except for mock-ups, UI experimentation and hobby projects, most sprints should start by implementing the data model of the feature.

6: CREATE THE API

Having created the necessary tables for a feature, you can proceed with the API.

As mentioned, you should first generate a small part, verify what ChatGPT provides and make sure that it meets expectations. Although it is impressive how accurate and

correct ChatGPT's answers often are, also in our experience, they are not flawless. Mistakes encountered include basic oversights (for which it always politely apologizes), but occasionally also more fundamental flaws that indicate that it completely misses the point.

LESSON 6: ChatGPT's answers should be evaluated like they come from another (sometimes less experienced) person, rather than from a flawless machine. The human developer who integrates ChatGPT's responses into the application remains liable.

If you are not content with the results that ChatGPT provides, it is often advisable to iterate to get what you want. If you are confident that the code is correctly structured, you could go for the overall thing in a second round after the initial sprint and relevant follow-up shows that the code is functionality correct and well-structured.

Also, if you are delivering to a company that already has policies for how the code should be structured you can ask ChatGPT that the code structured in that way.

For instance, do you have multiple layers with controller, service, and repo, or less than that. In general, keep things as simple as possible. The fewer moving parts and the fewer proprietary add-ons that ChatGPT has no knowledge about, the easier it will be for ChatGPT to get it right. Besides, you can always get ChatGPT to rewrite the files later if you decide.

Remember the API is not just happy scenario, so provide instructions about error handling, return codes, etc. For instance, in Example 1, the API server could initially crash when there was some issue. So we asked ChatGPT to wrap each call in a try catch, and return 200 or 500 depending on whether an error was encountered or not. We should note that on several other interactions, ChatGPT actually suggested that error handling should be part of the implementation and provided examples of how to do it.

LESSON 7: When starting the API, get a simple server running with a simple example and make sure you are happy with it to some level of maturity. There may be some production hardening missing that can be covered later, but the basic structure should be correct and satisfactory.

7: CREATE THE UI

Having established DB and API for the first sprint, proceed to the UI. This is where we see the largest difference working with ChatGPT, compared to the conventional approach.

With a developer you will typically show a screen shot of some sort, maybe Figma wireframes produced by product management, accompanied by some explanations.

Since ChatGPT does not support visual input yet, we need to rely on only explanations. This makes it more difficult to explain details like horizontal and vertical layout, and adds to the possibilities for incorrect interpretation, leading to incorrect or suboptimal results.

We therefore prefer to tap into the developer's fine-grained process of progress. When confronted with a feature with a page with multiple controls and fragments, he should begin with some simple subset and get it to work to create a basis. After this, he can add the remaining elements one by one or in small groups in an iterative way.

Similarly, when we instruct ChatGPT, we first ask for a simple version of the page, confirm it works, and then add further elements.

This process helps keep to the code and the development process – here meaning the interaction with ChatGPT – under control. You get small parts, which can easily be validated and tested. In other words, we do not end up with legacy code, contrary to the claim of Loukides[4].

LESSON 8: When working with ChatGPT on User Interface, ask first for a simple version. Then add remaining controls one by one or in small groups.

We will later discuss how we can scale the development of UI to a more efficient process where the UI is explained in a semi-formal way.

8: INTERMEZZO: EXAMPLE 1 IN DETAIL

In [1] the development of a Todo application proceeds from UI to API (contrary to our recommendation, but the book is about React, so it makes sense). Let's follow that path and see how ChatGPT works it out.

We start with the standard React application obtained by running `npx create-react-app todo`, which was part of the project setup. After this, we ask ChatGPT to create the contents for the files we already have as dummy versions:

Please generate a React application that can maintain a list of todo items.

There should be a single page implemented in App.js.

The page should contain an input field where a text can be entered and a button called Add which creates the item. The input field and button should be in a component TodoCreator.js

The page should also contain a list of all the generated items in a component TodoRow.js

The items should be stored in local storage of the browser. The list is initially empty.

Please indicate name and contents of each file

When you ask ChatGPT to generate code he will often give you examples and leave to you to complete the details. However, it is much faster if ChatGPT simply does it all. So, we ask for full contents of the files.

LESSON 9: Ask for full contents of files. It speeds up the process.

For the above request we get this response:

Okay, we'll need a few files to build this application:

1. `package.json`: This file keeps track of all dependencies and scripts for the project.
2. `index.js`: This file will serve as the entry point for our application.
3. `App.js`: This is the main component of our application which will render the TodoCreator and TodoRow components.
4. `TodoCreator.js`: This component is responsible for creating new todos.
5. `TodoRow.js`: This component is responsible for rendering all todo items.

ChatGPT goes on with the contents of each file.

1. `package.json`

This file contains the minimal necessary packages for your application:

```
json
Copy code
{
  "name": "todo-app",
  "version": "1.0.0",
  "main": "index.js",
  "dependencies": {
    "react": "^17.0.2",
    "react-dom": "^17.0.2"
  },
  "scripts": {
    "start": "parcel index.html"
  },
  "devDependencies": {
    "parcel-bundler": "^1.12.5",
    "babel-preset-env": "^7.15.8",
    "babel-preset-react": "^7.14.5"
  }
}
```

2. `index.js`

```
jsx
Copy code
import React from 'react';
import ReactDOM from 'react-dom';
import App from './App';

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

3. 'App.js'

```
jsx
import React from 'react';
import TodoCreator from './TodoCreator';
import TodoRow from './TodoRow';

class App extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      todos: JSON.parse(localStorage.getItem('todos')) || []
    };
  }

  addTodo = (todo) => {
    this.setState(prevState => {
      return {todos: [...prevState.todos, todo]};
    }, () => localStorage.setItem('todos', JSON.stringify(this.state.todos)))
  }

  render() {
    return (
      <div className="App">
        <TodoCreator addTodo={this.addTodo} />
        <TodoRow todos={this.state.todos} />
      </div>
    );
  }
}

export default App;
```

4. 'TodoCreator.js'

```
jsx
import React, { Component } from 'react';

class TodoCreator extends Component {
  constructor(props) {
    super(props);
    this.state = {
      input: ''
    };
  }

  handleChange = (event) => {
    this.setState({ input: event.target.value });
  }

  handleSubmit = (event) => {
    event.preventDefault();
    this.props.addTodo(this.state.input);
    this.setState({ input: '' });
  }

  render() {
    return (
      <form onSubmit={this.handleSubmit}>
        <input type="text" value={this.state.input} onChange={this.handleChange} />
        <button type="submit">Add</button>
      </form>
    );
  }
}

export default TodoCreator;
```

5. 'TodoRow.js'

```
jsx
import React from 'react';

const TodoRow = ({ todos }) => {
  return (
    <ul>
      {todos.map((todo, index) => (
        <li key={index}>{todo}</li>
      ))}
    </ul>
  );
}

export default TodoRow;
```

To start this application, make sure you have Node.js and npm installed. Put all the above files in the same directory and run `npm install` to install dependencies. Use the `npm start` command to start the application.

Sometimes there can be slight imprecisions in ChatGPT's responses. For instance, about the name of entities, file locations, etc. That's why it's good to have a working dummy application where files are correctly named and stored in locations that are consistent.

LESSON 10: When starting the interaction with ChatGPT, have a dummy application working.

Next, we want to indicate whether todos are done.

In the todo list, there should be a editable check mark indicating whether it is done or not.

Please indicate name and contents of files that need change.

ChatGPT responds with the changes.

1. 'App.js'

In the `'addTodo'` method, change the new todo item to an object:

```
jsx
addTodo = (todo) => {
  this.setState(prevState => {
    return {todos: [...prevState.todos, {text: todo, done: false}]};
  }, () => localStorage.setItem('todos', JSON.stringify(this.state.todos)));
}
```

2. 'TodoCreator.js'

No change is needed in `'TodoCreator.js'` as it simply passes the entered text to the `'addTodo'` method in `'App.js'`.

3. 'TodoRow.js'

We need to add a checkbox input to handle the status of the todo, and make changes to the `'todos.map()'` function:

```
jsx Copy code

import React from 'react';

const TodoRow = ({ todos, toggleDone }) => {
  return (
    <ul>
      {todos.map((todo, index) => (
        <li key={index}>
          <input
            type="checkbox"
            checked={todo.done}
            onChange={() => toggleDone(index)}
          />
          {todo.text}
        </li>
      ))}
    </ul>
  );
}

export default TodoRow;
```

We also need to add the `'toggleDone'` method to the `'App.js'` file to handle the status change of the todos:

```
jsx Copy code

toggleDone = (index) => {
  this.setState(prevState => {
    const todos = prevState.todos.map((todo, i) =>
      i === index ? {...todo, done: !todo.done} : todo
    );
    return {todos};
  }, () => localStorage.setItem('todos', JSON.stringify(this.state.todos)));
}

// Add the toggleDone prop to the TodoRow component
render() {
  return (
    <div className="App">
      <TodoCreator addTodo={this.addTodo} />
      <TodoRow todos={this.state.todos} toggleDone={this.toggleDone} />
    </div>
  );
}
```

We confirm the code is working and move on to the next missing part.

I would like the list of todo items to be split into two lists: those that are done, and those that are not. When I check/uncheck an item, it should move to the right list.

Please indicate name and full contents of files that need change.

ChatGPT responds:

To implement this feature, you need to modify the `'App.js'` and `'TodoRow.js'` files.

and breaks it into parts as usual.

1. 'App.js'

The `'App.js'` will split the todos into two separate lists, `'doneTodos'` and `'notDoneTodos'`, and pass each to a separate `'TodoRow'` component:

```
jsx Copy code

import React from 'react';
import TodoCreator from './TodoCreator';
import TodoRow from './TodoRow';

class App extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      todos: JSON.parse(localStorage.getItem('todos')) || []
    };
  }

  addTodo = (todo) => {
    this.setState(prevState => {
      return {todos: [...prevState.todos, {text: todo, done: false}]};
    }, () => localStorage.setItem('todos', JSON.stringify(this.state.todos))
  )

  toggleDone = (index) => {
    this.setState(prevState => {
      const todos = prevState.todos.map((todo, i) =>
        i === index ? {...todo, done: !todo.done} : todo
      );
      return {todos};
    }, () => localStorage.setItem('todos', JSON.stringify(this.state.todos))
  )

  render() {
    const doneTodos = this.state.todos.filter(todo => todo.done);
    const notDoneTodos = this.state.todos.filter(todo => !todo.done);

    return (
      <div className="App">
        <TodoCreator addTodo={this.addTodo} />
        <h2>Not Done</h2>
        <TodoRow todos={notDoneTodos} toggleDone={this.toggleDone} />
        <h2>Done</h2>
        <TodoRow todos={doneTodos} toggleDone={this.toggleDone} />
      </div>
    );
  }
}

export default App;
```

2. 'TodoRow.js'

The `'TodoRow.js'` file stays the same as the previous version:

```
jsx Copy code

import React from 'react';

const TodoRow = ({ todos, toggleDone }) => {
  return (
    <ul>
      {todos.map((todo, index) => (
        <li key={index}>
          <input
            type="checkbox"
            checked={todo.done}
            onChange={() => toggleDone(index)}
          />
          {todo.text}
        </li>
      ))}
    </ul>
  );
}

export default TodoRow;
```

Now, the todo list is split into two separate lists: those that are done, and those that are not done. When you check/uncheck an item, it will move to the right list.

We are ready for the last part.

There should be an additional section saying "Display completed items" with a check mark. The Not done list is only shown of that check mark is checked.

Please indicate name and full contents of files that need change.

To implement this feature, you will need to add a new state variable in the `App.js` file to track whether the completed items should be displayed or not.

Here's how you can do it:

`App.js`:

```
jsx Copy code

import React from 'react';
import TodoCreator from './TodoCreator';
import TodoRow from './TodoRow';

class App extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      todos: JSON.parse(localStorage.getItem('todos')) || [],
      displayDone: false
    };
  }

  addTodo = (todo) => {
    this.setState(prevState => {
      return { todos: [...prevState.todos, {text: todo, done: false}]};
    }, () => localStorage.setItem('todos', JSON.stringify(this.state.todos)))
  }

  toggleDone = (index) => {
    this.setState(prevState => {
      const todos = prevState.todos.map((todo, i) =>
        i === index ? {...todo, done: !todo.done} : todo
      );
      return { todos };
    }, () => localStorage.setItem('todos', JSON.stringify(this.state.todos)))
  }

  toggleDisplayDone = () => {
    this.setState(prevState => ({ displayDone: !prevState.displayDone }));
  }

  render() {
    const doneTodos = this.state.todos.filter(todo => todo.done);
    const notDoneTodos = this.state.todos.filter(todo => !todo.done);

    return (
      <div className="App">
        <TodoCreator addTodo={this.addTodo} />
        <div>
          <input
            type="checkbox"
            checked={this.state.displayDone}
            onChange={this.toggleDisplayDone}
          />
          <span>Display completed items</span>
        </div>
        <h2>Not Done</h2>
        <TodoRow todos={notDoneTodos} toggleDone={this.toggleDone} />
        {this.state.displayDone && (
          <h2>Done</h2>
          <TodoRow todos={doneTodos} toggleDone={this.toggleDone} />
        )}
      </div>
    );
  }
}

export default App;
```

This code adds a checkbox and associated handler (`toggleDisplayDone`) for controlling the display of completed items. The Done list is now conditionally rendered based on whether `this.state.displayDone` is true.

Rege

The application (focused on UI) now has most of what is shown in Section 1, except the Comments page, the styling, and a few details. Now we move on with the API.

I would like to add a Node server implemented by a file `Server.js` offering a rest API for todos items. It should offer endpoints to add an item, to mark an item done and to mark an item undone. The application just keeps the todos in memory.

In `package.json` there should be a separate script to start the server. It should listen to port 3500.

So far the React application should not use the API, so we only add `Server.js` and `package.json`, we do not alter existing react files.

Please indicate name and full contents of files that need change.

Let's create a simple Node server using Express that stores todos in memory and provides the specified API endpoints.

`Server.js`:

```
javascript Copy code

const express = require('express');
const bodyParser = require('body-parser');

const app = express();
app.use(bodyParser.json());

let todos = []; // Storing todos in memory

// Endpoint to add a todo
app.post('/todos', (req, res) => {
  const newTodo = req.body;
  todos.push(newTodo);
  res.json(newTodo);
});

// Endpoint to mark a todo as done
app.put('/todos/:id/done', (req, res) => {
  const id = req.params.id;
  const todo = todos.find(t => t.id === id);
  if (todo) {
    todo.done = true;
    res.json(todo);
  } else {
    res.status(404).json({ error: 'Todo not found' });
  }
});

// Endpoint to mark a todo as not done
app.put('/todos/:id/undone', (req, res) => {
  const id = req.params.id;
  const todo = todos.find(t => t.id === id);
  if (todo) {
    todo.done = false;
    res.json(todo);
  } else {
    res.status(404).json({ error: 'Todo not found' });
  }
});

app.listen(3500, () => {
  console.log('Server is running on port 3500');
});
```

The `Server.js` file uses the Express web framework to provide three endpoints: one for adding a new todo (a POST request to `/todos`), one for marking a todo as done (a PUT request to `/todos/:id/done`), and one for marking a todo as undone (a PUT request to `/todos/:id/undone`). The todos are stored in an array in memory, so they will be lost when the server is restarted.

You'll also need to update `package.json` to add a script to start the server:

`package.json`:

```
json Copy code
{
  "name": "todo-app",
  "version": "1.0.0",
  "main": "index.js",
  "scripts": {
    "start": "react-scripts start",
    "server": "node Server.js", // add this line
    "build": "react-scripts build",
    "test": "react-scripts test",
    "eject": "react-scripts eject"
  },
  "dependencies": {
    "react": "^17.0.2",
    "react-dom": "^17.0.2",
    "react-scripts": "4.0.3",
    "express": "^4.17.1",
    "body-parser": "^1.19.0"
  }
}
```

With these changes, you can start your Express server by running `npm run server`.

Let's see how to have simple manual test.

Also please indicate the endpoints and how I can test them

Certainly, here are the endpoints you can test:

1. **POST /todos**: This endpoint creates a new todo item. The todo item should be sent in the request body as JSON. For instance:

```
json Copy code
{
  "id": "1",
  "text": "Buy milk",
  "done": false
}
```

2. **PUT /todos/id/done**: This endpoint marks a todo item as done. You need to replace `:id` with the id of the todo item you want to mark as done. There's no need to send a request body for this endpoint.
3. **PUT /todos/id/undone**: This endpoint marks a todo item as not done. You need to replace `:id` with the id of the todo item you want to mark as not done. There's no need to send a request body for this endpoint.

You can test these endpoints using various tools such as [Postman](#) or [curl](#). Here's how you can do it with curl from the command line:

1. **POST /todos**:

```
bash Copy code
curl -X POST -H "Content-Type: application/json" -d '{"id":"1","text":"Buy
```

2. **PUT /todos/id/done**:

```
bash Copy code
curl -X PUT http://localhost:3500/todos/1/done
```

3. **PUT /todos/id/undone**:

```
bash Copy code
curl -X PUT http://localhost:3500/todos/1/undone
```

After running the POST command, you should see the new todo item echoed back in the response. For the PUT commands, you should see the updated todo item in the response, marked as done or not done as appropriate.

After we try out the end points and confirm they are working we can make the UI use them.

The React application should now use the rest API for storing and fetching todo items.

Please indicate name and full contents of files that need change.

To use the REST API from the React application, you will need to make HTTP requests from the React components. This can be done using the Fetch API that is built into modern browsers.

Firstly, update the `TodoCreator.js` file to post the new todo to the server when the form is submitted:

`TodoCreator.js`:

```
jsx Copy code
import React from 'react';
import './TodoCreator.css';

class TodoCreator extends React.Component {
  constructor(props) {
    super(props);
    this.state = {value: ''};
  }

  handleChange = (event) => {
    this.setState({value: event.target.value});
  }

  handleSubmit = async (event) => {
    event.preventDefault();
    const newTodo = { text: this.state.value, done: false };
    const response = await fetch('http://localhost:3500/todos', {
      method: 'POST',
      headers: {
        'Content-Type': 'application/json'
      },
      body: JSON.stringify(newTodo)
    });
    const createdTodo = await response.json();
    this.props.addTodo(createdTodo);
    this.setState({value: ''});
  }

  render() {
    return (
      <div className="add-todo">
        <form onSubmit={this.handleSubmit}>
          <input
            type="text"
            value={this.state.value}
            onChange={this.handleChange}
            required
          />
          <button type="submit">Add</button>
        </form>
      </div>
    );
  }
}

export default TodoCreator;
```

Then, modify `App.js` to fetch the list of todos from the server when the component is mounted and when a todo is added. Also, when a todo's done status is toggled, make a request to the appropriate endpoint on the server:

`App.js`:

```
jsx Copy code

import React from 'react';
import TodoCreator from './TodoCreator';
import TodoRow from './TodoRow';
import './App.css';

class App extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      todos: [],
      showCompleted: false
    };
  }

  componentDidMount = async () => {
    const response = await fetch('http://localhost:3500/todos/undone');
    const todos = await response.json();
    this.setState({ todos });
  }

  addTodo = async (newTodo) => {
    this.setState({
      todos: [...this.state.todos, newTodo]
    });
  }

  toggleTodo = async (id) => {
    const todo = this.state.todos.find(t => t.id === id);
    if (todo) {
      todo.done = !todo.done;
      const response = await fetch(`http://localhost:3500/todos/${id}/${todo.done}`);
      const updatedTodo = await response.json();
      this.setState({
        todos: this.state.todos.map(t => t.id === id ? updatedTodo : t)
      });
    }
  }

  toggleShowCompleted = () => {
    this.setState({ showCompleted: !this.state.showCompleted });
  }

  render() {
    const undoneTodos = this.state.todos.filter(todo => !todo.done);
    const doneTodos = this.state.todos.filter(todo => todo.done);
    return (
      <div className="app-container">
        <h1>Todo List ({undoneTodos.length} items left)</h1>
        <TodoCreator addTodo={this.addTodo} />
        <div>
          <label>
            <input
              type="checkbox"
              checked={this.state.showCompleted}
              onChange={this.toggleShowCompleted}
            />
            Display completed items
          </label>
        </div>
        <h2>Not Done</h2>
        {undoneTodos.map((todo) => (
          <TodoRow key={todo.id} todo={todo} toggleTodo={this.toggleTodo} />
        ))}
        {this.state.showCompleted && (
          <div>
            <h2>Done</h2>
            {doneTodos.map((todo) => (
              <TodoRow key={todo.id} todo={todo} toggleTodo={this.toggleTodo} />
            ))}
          </div>
        )}
      </div>
    );
  }
}

export default App;
```

With these changes, the React application will now make HTTP requests to the server to fetch and modify todos, instead of storing them in local storage.

As mentioned, sometimes ChatGPT will generate code that does not work. In many cases it is because of small details like not naming paths in json correctly (because of lack of context). In general, ChatGPT is pretty good at helping with the issues.

In this case we have this error.

When the front end calls the API I get this error Failed to fetch TypeError: Failed to fetch at _callee

ChatGPT responds:

This issue is most likely happening because of the browser's Cross-Origin Resource Sharing (CORS) policy, which prevents requests to different origins (different domain, protocol, or port) for security reasons.

You can solve this issue by using the CORS middleware in your Node.js server, which will allow the browser to make requests to your server from a different origin.

First, you need to install the `cors` package in your Node.js server. You can do this by running the following command in the terminal:

```
bash Copy code

npm install cors
```

After installing the package, you need to import it in your `Server.js` file and use it as a middleware. Here's how to do it:

`Server.js`:

```
javascript Copy code

const express = require('express');
const bodyParser = require('body-parser');
const cors = require('cors'); // Import the cors package

const app = express();
app.use(bodyParser.json());
app.use(cors()); // Use cors as a middleware

// ... rest of the code
```

This change will add the appropriate Access-Control-Allow-Origin headers to the responses from your server, allowing the browser to make requests to your server.

Please make sure both the Node server and React application are running on different ports, and update the fetch URLs in the React application with the correct port number for your Node server.

If you continue to have problems, please ensure that the server is running and that there are no network issues preventing the React application from communicating with the server. You should be able to see more details about the error in the browser's JavaScript console.

The advice turns out to work. In general, such advice should be confirmed by checking with other sources. Although, code is contributed by ChatGPT, you should see yourself as liable for it. See [7] for more on this aspect and [6] for questions about copyright regarding the code. Also notice that confidential code or information should not be shared with ChatGPT as observed in [3].

LESSON 11: ChatGPT may generate code with issues for various reasons, e.g., because we did not ask explicitly to avoid them or because he made an interpretation that is incorrect in our context. In such cases, simply ask for help; most of the time he will be able to (help) resolve the problem.

At this point, repeated testing reveals some new errors, ChatGPT replies with new versions where the errors are solved, and the ping-pong game continues for a little while until the code eventually works. Again, the trick is to get ChatGPT to do the work and keep providing the updated component in full, rather than to provide instructions for the developer to address these himself.

LESSON 12: As mentioned, ChatGPT may generate code with errors but getting it to fix them is quicker than writing the code yourself, which may also introduce errors that may take time to find and resolve.

Interestingly, at the end, the todo items are back in single list, rather than two different for done and undone. ChatGPT lost some context along the way in the new issues, as others have reported, and we had to get him back on track. Also, we had to instruct him that the API should be called to mark todo items done or not done.

LESSON 13: ChatGPT may generate code with problems that were solved earlier in the dialog. You need to test what he returns at every step and ask errors to be fixed.

We will say more about automatic testing later. It's finally time to get the DB layer done.

I have a postgres database running locally on port 5432. Name of database is todos. User is "postgres", password is "password". Please provide the following:

1. SQL for creating a table that can hold todo items. It should include proper indices.
2. Updated version of Server.js that calls the database to insert items and fetch items

He politely responds.

Sure! Let's go step by step:

1. Here is the SQL to create a "todos" table:

```
sql
CREATE TABLE todos (
  id SERIAL PRIMARY KEY,
  name TEXT NOT NULL,
  done BOOLEAN NOT NULL DEFAULT false
);

CREATE INDEX idx_todos_done ON todos(done);
```

This will create a "todos" table with an "id" column as the primary key, a "name" column for the todo text, and a "done" column to store the status of each todo. The "idx_todos_done" index will improve the performance of queries that filter by the "done" column.

1. We'll need to install "pg" npm package to interact with Postgres database in Node.js. Run "npm install pg" in your terminal.

And he continues:

1. We'll need to install "pg" npm package to interact with Postgres database in Node.js. Run "npm install pg" in your terminal.

Then here is the updated "Server.js":

```
javascript
const express = require('express');
const { Pool } = require('pg');
const app = express();
const port = 3500;

const pool = new Pool({
  host: 'localhost',
  port: 5432,
  user: 'postgres',
  password: 'password',
  database: 'todos'
});

app.use(express.json());

app.get('/todos/done', async (req, res) => {
  const result = await pool.query('SELECT * FROM todos WHERE done = true');
  res.json(result.rows);
});

app.get('/todos/undone', async (req, res) => {
  const result = await pool.query('SELECT * FROM todos WHERE done = false');
  res.json(result.rows);
});

app.post('/todos', async (req, res) => {
  const { name } = req.body;
  const result = await pool.query('INSERT INTO todos (name) VALUES ($1) RETURNING *');
  res.json(result.rows[0]);
});

app.post('/todos/:id/done', async (req, res) => {
  const { id } = req.params;
  const result = await pool.query('UPDATE todos SET done = true WHERE id = $1');
  res.json(result.rows[0]);
});

app.post('/todos/:id/undone', async (req, res) => {
  const { id } = req.params;
  const result = await pool.query('UPDATE todos SET done = false WHERE id = $1');
  res.json(result.rows[0]);
});

app.listen(port, () => {
  console.log('Todo API listening at http://localhost:${port}');
});
```

In the updated "Server.js", we create a connection pool to the Postgres database with "new Pool()". This pool handles creating and managing multiple client connections to the database. We then use this pool to execute SQL queries in each endpoint.

Please ensure that the "pg" package is installed in your project by running "npm install pg" and ensure the postgres server is running and the "todos" table exists in the "todos" database.

Then follows some more ping-pong about

- The user the API should connect to the DB with.
- Handling "done" information in post endpoint.

LESSON 14: Be as precise as you can in stating what you need from ChatGPT. Whenever you omit details, ChatGPT may do something else than you expect.

Not long after this we have an application that works with all three layers. However, there are still many details

missing before the application is appropriate for production. For instance, the server code is in one big file which does not scale for a bigger API. Also, there is no error handling. We return to these and other issues later.

9: FURTHER LESSONS FROM EXAMPLE 2

As advocated by Fowler[3], before we started to work with ChatGPT on Example 2, we provided an elaborate description of the technology stack and the system architecture. Based on that, we turned the question around and asked ChatGPT which additional information he needed to best help us during our development task.

This resulted in a long list of questions, which we answered and which – as confirmed by ChatGPT – provided a good basis for our collaboration.

LESSON 15: If ChatGPT is lacking context, he may make assumptions instead of asking for clarifications. You can circumvent this by explicitly directing him initially to ask clarifications.

Areas where ChatGPT's help proved useful or invaluable:

- Rewriting the security and authentication layer, by replacing a proprietary component with an approach based on JWT tokens.
- Migrating from simple sequence-based primary keys to primary keys that follow the Twitter Snowflake pattern.
- Implementing new frontend views based on the existing application structure.
- Hunting down and resolving issues.
- Providing feedback and structuring the pros and cons for engineering tradeoffs.

One example where ChatGPT significantly sped up issue resolution is the following. After modifying the primary keys in the database from natural numbers (1, 2, 3...) to Twitter Snowflake style (e.g., 1139459444921196544), in certain situations the frontend could not find the data required, even though it was in the database.

When we presented this issue to ChatGPT, within seconds he explained that in the frontend our ids (modelled as long in our backend) should be represented as string to avoid rounding for the large numbers that our indexes are. It is our estimation that isolation of such an issue without ChatGPT could easily have taken a full day for an experienced developer.

Also, confronted with situations that required an engineering trade off, ChatGPT provided valuable

overviews of the pros and cons. It should be noted that on several instances – following the DRY principle – we deviated from the suggestions, e.g., by implementing a common feature in an `AbstractService` class, rather than in each service itself.

10: CREATE THE STYLING

If the DB layer is seen as the nitty-gritty details below the API, the styling are the nitty-gritty details above the UI.

Explaining styling to ChatGPT is even worse than explaining the basic layout of pages.

There are various offerings that can generate artifacts from drawings or even Figma's. They may gain traction onwards. We tried out one with not the results we needed. For instance, it was not possible to generate React code. We do not rule out that such tools exist, but the feature had been removed from the tool we tried out.

Instead, we tried to get ChatGPT to generate CSS for Example 3. He correctly used an HTML grid and got the basic layout right, but adjusting the details turned out to be impossible after 2-3 iterations.

At this point we reverted to classical techniques and studied HTML and CSS documentation and manually did the needed adjustments.

LESSON 16: Sometimes, several iterations do not bring you closer to a solution. In these cases, consider reverting to classical techniques, like Googling, Stack Overflow, YouTube demos, reading the documentation, etc.

As the discipline of coding with ChatGPT evolves, it can be expected that efficient approaches will be developed to deal with UI specification, CSS, SASS, etc.

By the way, the approach to styling is in general evolving. Initially you did the styling in HTML, then came CSS, and these days you have SASS or even technologies such as Tailwind CSS. You should instruct ChatGPT to deal with styling at the level fit for your application needs.

Also, you can request that controls be used from standard libraries such as `material-ui`¹ or more recent approaches like Tailwind CSS, etc. However, notice that so far, ChatGPT is not trained on material to the current date, so he may prefer approaches that are not fully up to date.

For instance, In Example 2, when we later created a basic application based on Next.js, he was expecting a file

¹ Material UI, via <https://mui.com>.

structured based on the classical router with the pages directory. In contrast, the default option when creating Next.js applications these days uses the new App router.

LESSON 17: ChatGPT currently does not know about knowledge published after September 2021.

11: CREATE THE TESTS

As mentioned, ChatGPT – like most developers – generates code with errors, even errors that were previously fixed.

The most efficient approach to dealing with these is to have two slim test suites, one that tests the API and one that tests the UI.

For Example 1, the UI test was produced with Puppeteer. It has these steps:

- Open browser and go to main page.
- Click Show completed todos.
- Type into the item input field.
- Click the "Add" button.
- Confirm the new item is there.
- Click the new item's check mark.
- Check the new item is now in the done list.

The test ends by deleting the new item using the API.

When adding new functionality from ChatGPT we check locally that the UI test runs before merging the pull request. We also manually check the new feature works and add a UI test for it. For the new feature, we prefer to first check it manually, so that if the automatic test fails, we know the problem is with the test.

The API test was a sequence of API calls in code using axios (a HTTP client library) imitating a UI interaction. Again, the test ends by deleting the data it has added using the API.

LESSON 18: ChatGPT generates several errors. It is valuable to have a slim UI test and API test that can be run locally before merging pull requests.

We do not bother to mock database or service layer. Since we have the development environment running locally, we can rely on it as long as the tests:

- Do not make any assumptions about the condition of the database on entry.
- Leave the database in the same condition on exit as it was in on entry.

LESSON 19: By keeping the tests slim, lightweight, and direct, they do not need to be time consuming and can be a good way of testing pull requests from ChatGPT.

The tests were written manually, because we wanted to make sure the code from ChatGPT did what we asked. Some people prefer to let ChatGPT generate the tests, but we see a risk that when ChatGPT gets something wrong in code, he will get the same thing wrong in tests. Indeed, errors were frequently about aspects that were not sufficiently well communicated, e.g., field names.

Nevertheless, we are open to the approach that you start by having ChatGPT create the tests and review that they capture what you want.

12: REFACTOR OUT COMMON PARTS

When you ask ChatGPT to generate some code, he will try to satisfy your requirements, but in many cases, he will refrain from satisfying your untold requirements, even if you regard them as common sense.

To be fair, when he generates some code, he will sometimes tell you that this is just an example, and you also need to take into account some list of considerations. But at the end of the day, you cannot assume he satisfies non-functional requirements that you do not mention.

An obvious example is when the same code occurs in multiple places. In this case we want to factor it out.

For instance, in Example 1 we added the possibility to have some fields of the todo encrypted in the database (GDPR, you know the drill), so that developers cannot see the contents of the fields when they access the database.

The code to encrypt when storing and decrypt when fetching should not be replicated to the handler of each end point. There should rather be utility functions that are called from the API endpoints handlers:

```
const crypto = require('crypto');
const IV_LENGTH = 16;

function encrypt(text, key) {
  let iv = crypto.randomBytes(IV_LENGTH);
  let cipher = crypto.createCipheriv('aes-256-cbc', Buffer.from(key), iv);
  let encrypted = cipher.update(text);
  encrypted = Buffer.concat([encrypted, cipher.final()]);
  return iv.toString('hex') + ':' + encrypted.toString('hex');
}

function decrypt(text, key) {
  let textParts = text.split(':');
  let iv = Buffer.from(textParts.shift(), 'hex');
  let encryptedText = Buffer.from(textParts.join(':'), 'hex');
  let decipher = crypto.createDecipheriv('aes-256-cbc', Buffer.from(key), iv);
  let decrypted = decipher.update(encryptedText);
  decrypted = Buffer.concat([decrypted, decipher.final()]);
  return decrypted.toString();
}

module.exports = {
  encrypt,
  decrypt
};
```

You can handle these requirements up front when asking code to be generated or refactor along the way by explaining to Chat GPT the current files and their contents and the required refactoring.

LESSON 20: As you review the PRs from ChatGPT, keep an eye on parts of the code that could or should be refactored. Either do the refactoring yourself or ask ChatGPT to do it.

In general, we want to limit refactoring to a reasonable limit and only allow it when it has a tangible result and good balance between effort and benefit not based solely on speculation.

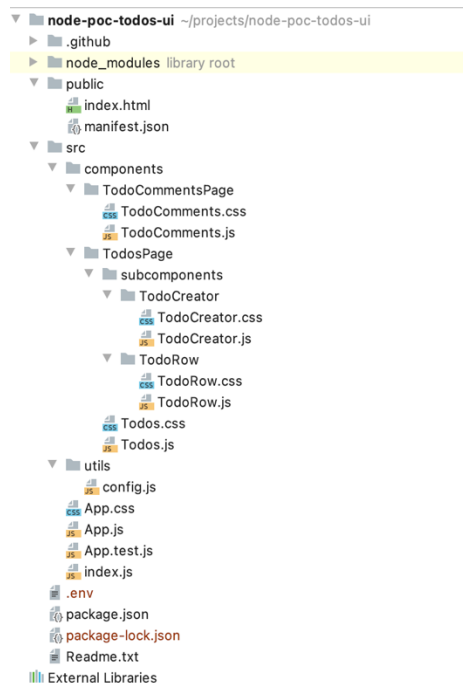
But some refactoring just has to be done. Like having significant chunks of code that is replicated verbatim in many methods pulled out to a method.

13: DESIGN THE FILE STRUCTURE

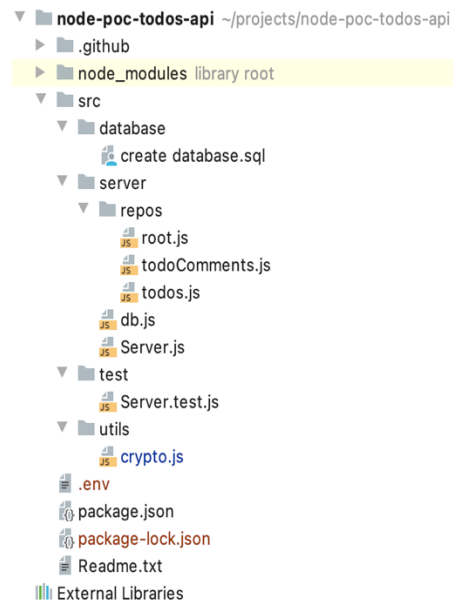
When you design your application with ChatGPT there are typically two different approaches:

- You did this many times and have a clear idea about the major design and architecture choices.
- You are doing this in an exploratory manner.

Both approaches have their merits. If you work in the explanatory way, you should expect to do refactoring along the way. ChatGPT can provide advice on the structure and even generate the files. For instance, at the end of Example 1, the UI looked like this:



Similarly, the API looked like this:



Have a clear structure for the files of your application in mind from the beginning and communicate it to ChatGPT in your prompts, or be prepared that you or ChatGPT should refactor the code along the way to replace a single file by a set of files each of which covers a feature.

Similarly, be prepared that as more complexity arrives, a single layer may be split into several layers, e.g., for the API as service, controller, repo, etc.

LESSON 21: You are responsible for ensuring that the file structure of the application is manageable. Define it from the outset or refactor it (with the help of ChatGPT) along the way.

14: KEEP THE CODE UNDER CONTROL

As mentioned, you should be prepared to refactor. What was said in the preceding section was about splitting large files into smaller more manageable files, e.g., splitting a single file with the handlers for all end points into separate files grouped by entity or entity groups.

Also, we have mentioned that every response from ChatGPT should be seen as a pull request. That way no code ever enters the code repository without some sanity check. You don't need to read every single line, but you need to gain confidence that the code follows the philosophy and guidelines that you are trying to enforce. What we can add to this is that the code should be *structurally consistent*. This means that how different kinds of functionality or logic is structured or organized should be the same from feature to feature.

For instance, in Example 1, the page fragments were initially classes. However, when we added the page for adding comments on todos, we introduced version 6.14.1 of `react-router-dom` and had to make use of the `useParams` and `useNavigate` hooks for retrieving the URL parameters and for navigation, respectively. And this led to some classes being reformulated as functions. In this case, it is preferable that all the components are functions.

The reason ChatGPT got us into the classes formulation in the first place may be that this was the most widespread method during the period of documents that ChatGPT has been trained with.

LESSON 22: Keep the code consistent. ChatGPT may change structure of the code he generates when new requirements from you become clear. You should refactor the previously generated code (make ChatGPT do it) when this happens. Consider code measurement tooling at the end of the project to confirm and document consistency.

It is claimed in [2] regarding the code generated by ChatGPT that “For all practical purposes, it’s ‘legacy code,’ even if it’s only a few minutes old.” As hinted at earlier, this does not apply to the approach described here:

1. Developers may leave you and are not there to explain their code. ChatGPT and related technologies are here to stay and will improve over time.
2. You must review the code from ChatGPT and adopt it as your own code. So long as you are there, it’s not legacy.

LESSON 23: See the code by ChatGPT as your code. You must be able to account for it. That way it has same value as code developed by you or your developers.

15: GO TO PRODUCTION

So, you have your code generated by ChatGPT and it does what you asked. The UI tests run, the API tests run, all good. But are you ready for production?

If you are an experienced developer, you know many things are missing. For instance, you may need a penetration test to ensure that you did everything to correctly handle security-related HTTP-headers, SQL injection attacks, cross-site scripting etc.

Actually, you can ask ChatGPT what is required from, say, an API before it is mature for production, and he will help you:

When developing an API, it's essential to have clear evaluation criteria to determine when the development process is complete and the API is ready for release. Here are some good evaluation criteria to consider:

1. **Functionality:** Ensure that the API performs all the intended functions as outlined in the requirements and design specifications. Test each endpoint and verify that they return the expected results.
2. **Reliability:** The API should be reliable and available for use when needed. Evaluate its stability and make sure it handles errors gracefully, returning appropriate error codes and messages.
3. **Performance:** Measure the API's response times and throughput under various loads. Check if it meets the performance requirements and is capable of handling the expected user traffic.
4. **Security:** Assess the API's security measures, including authentication, authorization, and data protection. Verify that it's protected against common security vulnerabilities, such as SQL injection, cross-site scripting (XSS), and CSRF attacks.
5. **Scalability:** Test the API's ability to handle increasing loads and growing user base. Ensure it can scale horizontally or vertically to accommodate future growth.
6. **Documentation:** Evaluate the clarity and completeness of the API documentation. Developers should be able to understand how to use the API without unnecessary confusion.
7. **Usability:** Consider the developer experience and usability of the API. Is it intuitive and easy to use? Does it follow RESTful or other design principles that developers are familiar with?
8. **Compatibility:** Verify that the API is compatible with different client environments, programming languages, and platforms.
9. **Error Handling:** Check that the API provides meaningful error messages and appropriate HTTP status codes when errors occur. This helps developers troubleshoot and resolve issues efficiently.
10. **Versioning and Backward Compatibility:** If you anticipate future changes, ensure that the API supports versioning and backward compatibility to avoid breaking existing client implementations.
11. **Testing:** Conduct thorough unit testing, integration testing, and end-to-end testing to identify and resolve any bugs or issues.
12. **Monitoring and Analytics:** Implement monitoring and logging mechanisms to track API usage, performance, and potential issues in real-time.
13. **API Lifecycle Management:** Consider the processes for deploying, updating, and deprecating the API over time.
14. **Governance and Legal Compliance:** Ensure that the API complies with relevant regulations and company governance policies, especially if it deals with sensitive data.
15. **Feedback from Stakeholders:** Seek feedback from potential API consumers, internal stakeholders, and domain experts to gauge their satisfaction and address any concerns.

By evaluating the API against these criteria, you can ensure that it meets the required standards and is ready for deployment and use in a production environment.

This provides a checklist, which can be elaborated upon as needed. If you have any questions, you can ask again. For instance, how do you check if the API is vulnerable to SQL injection, ask and he will help you test it.

For example:

In some penetration tests results I have been instructed to set certain http headers to certain values. Can you remind me what I have to do in my react application?

HTTP headers can be manipulated directly within your React application when making API requests, typically done using an HTTP client library like `'axios'` or `'fetch'`.

Below is an example using axios:

```
javascript Copy code

import axios from 'axios';

const instance = axios.create({
  baseURL: 'https://your-api.com',
  timeout: 1000,
  headers: {
    'X-Custom-Header': 'value',
    'Content-Type': 'application/json'
  }
});
```

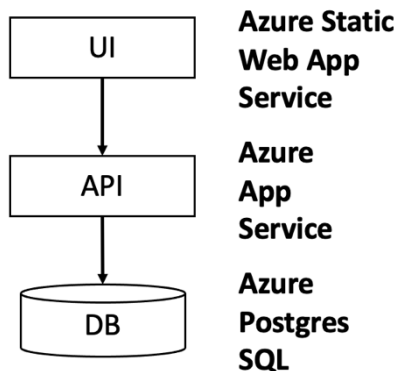

Again, such responses should be checked with alternative sources.

LESSON 24: The code that you get from ChatGPT may not be ready for production, but he can help you understand what needs to be checked and how, and which changes are needed as a result of the checks.

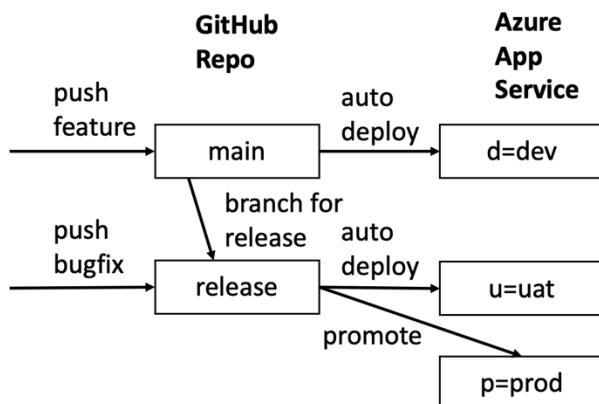
16: SET UP THE PROD ENVIRONMENT

Like with the development environment, we eventually need a production environment.

For Example 1, we used Azure Postgres for the database layer. For the UI and the API, we initially used Azure App service. For the UI, upload took ages and there were some issues getting it to work. After a while we backtracked and looked for demos of how to deploy a React application on Azure and changed to Azure static web app, which ran much smoother.



We need some additional DevOps work. For instance, as mentioned, for Example 1 we set up a GitHub repository and had automatic deploy to the environment when there are commits to the branch.



Depending on your organizational setup, number of applications to develop, etc. you may find it desirable to script provisioning of your environments as often done.

18: SCALE WITH FORMALISM

Once you start doing multiple applications with ChatGPT, it becomes natural to optimize the communication.

One interesting possibility is that you can train ChatGPT with training material yourself. That is, you can give examples of what prompts you provide and at the same time provide the desired answer.

This can be used to train him to deal with formal specifications of UI. For instance, you provide a text representation of a UI by listing all the components of a page including alignment margins etc. and show the HTML or React code you expect.

We have already pursued this approach some way but leave as future work to report on it in detail.

LESSON 25: Formalize your communication with ChatGPT with initial training. That way the communication can hopefully be way more efficient and less ambiguous.

Indeed, it's been a goal in the software engineering community for decades to formalize development. We create a specification in some formal language, and the system generates the code which is provably correct with respect to the specification.

So far this did not become a widespread practice in the community. We can speculate that the following reasons apply:

1. It is very difficult to translate the nature of the problem that a solution will solve to a formal specification in one go. If this was easy, waterfall development would have been more successful.
2. Your formal specifications cannot embrace the needed functionality of your target applications unless they are as complicated as the target language and then what have you won?
3. Due to the complexity of the specification language, you will make errors in your specification that are as severe as errors in the generated application.

What became possible overnight is to provide the specification that you can invent yourself and train ChatGPT in. And the remaining requirements that do not fit conveniently into the specification can be mentioned informally, they will be taken into account by ChatGPT. We look forward to exploring this possibility further.

19: SCALE WITH DEVELOPERS

Remember the five developers that left us? Now they are back.

Today the development lead, as we envisage the role, coaches the developers, and supports them in how they code. Replacing them with ChatGPT you may think they are gone.

But when this shift is made, the development lead becomes the bottleneck. The obvious way out is that the development lead does not coach ChatGPT in how to code. He coaches the developers in how they coach, work, and interact with ChatGPT.

Software development has fundamentally changed since the introduction of ChatGPT. Both the industry as a whole and every practitioner in it is trying to understand the implications in this interesting era.

Our approach assumes that there are experienced developers; it is an interesting question how humans can continue to build and retain knowledge and skills if ChatGPT is to take over the bulk of coding. Will this not also lead to an erosion of understanding? Also, we assume ChatGPT has been trained on material regarding coding in given language. Again, for a new language, how can this happen for a new language if ChatGPT itself is supposed to do the coding? Many questions remain.

20: LIMITATIONS

ChatGPT can help you, but there are also limitations:

- ChatGPT may be periodically unavailable.
- He accepts a limited number of prompts in a fixed amount of time.
- There have been reports of degradation of his responses, at least for ChatGPT 3.5
- ChatGPT is an echo chamber: he will provide what knowledge is agreed by most people. If you have unique shortcuts based on experience, he may not agree.
- ChatGPT does not know about recent developments, such as new languages and frameworks, etc.

In practice we have so far not found these limitations to severely restrict the benefits of using ChatGPT.

CONCLUSION

We have outlined an approach to developing mature, production-ready, maintainable full stack applications from scratch with ChatGPT. Based on our examples we estimate the productivity improvement to be a factor between 2 and 5.

The method does not work out of the box, you must adopt various practices. Most importantly, you should work with ChatGPT the same way as you do today with developers:

- Create the architecture and features in sprints.
- Regard the contributions by ChatGPT as pull requests that you should review.
- Implement lightweight regression testing for UI and API to confirm that refactoring and adding functionality by ChatGPT does not break anything.
- Get ChatGPT to refactor along the way to turn duplicated code into methods and to implement other patterns that you observe the need for in the pull request reviews.

Our hands-on experiments suggest that ChatGPT, guided the right way, can contribute working code, identify relevant libraries and frameworks, identify the source of bugs, suggest methods to mature and secure the application, and contribute in many other ways that save time and money, and improve quality and time-to-market.

ChatGPT can make mistakes, but so can developers, professors, and textbooks. When you receive advice that you cannot confirm satisfactorily yourself, it is always good to double check with an alternative authoritative source. For instance, when putting a new application to production, it is a good idea to get an external company to conduct a penetration test, whether the application development got help from ChatGPT or not.

At the end of the day, we think of ChatGPT as a developer that read all the relevant documents and pieces of code that exist for that language in question and gives the best answer to your question at hand. In our experience, ChatGPT and related technologies should be evaluated and used today, since their potential to reduce time-to-market and increase efficiency, while increasing quality, is vast.

In most cases, ChatGPT can do things faster than you can. Whenever getting caught up in anything that seems remotely routine, ask yourself this question: Shouldn't I get ChatGPT to do this?

A revolution in software development has started. Are organizations ready to adopt and reap the benefits...?

REFERENCES

- [1] Adam Freeman: *Pro React 16*. A Press. 2013.
- [2] Danusha Navod Gamage: *7 Ways ChatGPT Can Help Developers*. <https://cult.honeypot.io/reads/how-can-chatgpt-help-developers/>, March 7, 2023.

- [3] Martin Fowler: *An example of LLM prompting for programming*.
<https://martinfowler.com/articles/2023-chatgpt-xu-hao.html>, April 13, 2023.
- [4] Mike Loukides: *Real World Programming with ChatGPT*. <https://www.oreilly.com/radar/real-world-programming-with-chatgpt/>. April 25, 2023.
- [5] David Gewirtz: How to use ChatGPT to write code
<https://www.zdnet.com/article/how-to-use-chatgpt-to-write-code/>. ZDNet. June 2, 2023.
- [6] David Gewirtz: Who owns the code? If ChatGPT's AI helps write your app, does it still belong to you?
<https://www.zdnet.com/article/who-owns-the-code-if-chatgpts-ai-helps-write-your-app-does-it-still-belong-to-you/>. ZDNet. June 19, 2023.
- [7] David Gewirtz: If you use AI-generated code, what's your liability exposure.
<https://www.zdnet.com/article/if-you-use-ai-generated-code-whats-your-liability-exposure/>. ZDNet. June 20, 2023
- [8] Philp Guo: *Real-Real-World Programming with ChatGPT*. <https://www.oreilly.com/radar/real-real-world-programming-with-chatgpt/>, July 25, 2023.
- [9] W3Schools: ChatGPT Coding.
https://www.w3schools.com/gen_ai/chatgpt-4/chatgpt-4_code.php