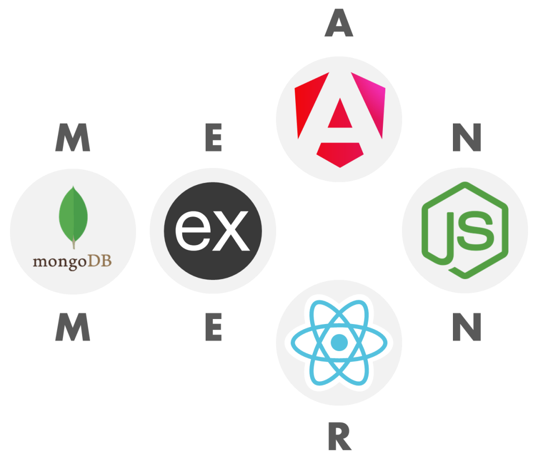


Coding Guidelines

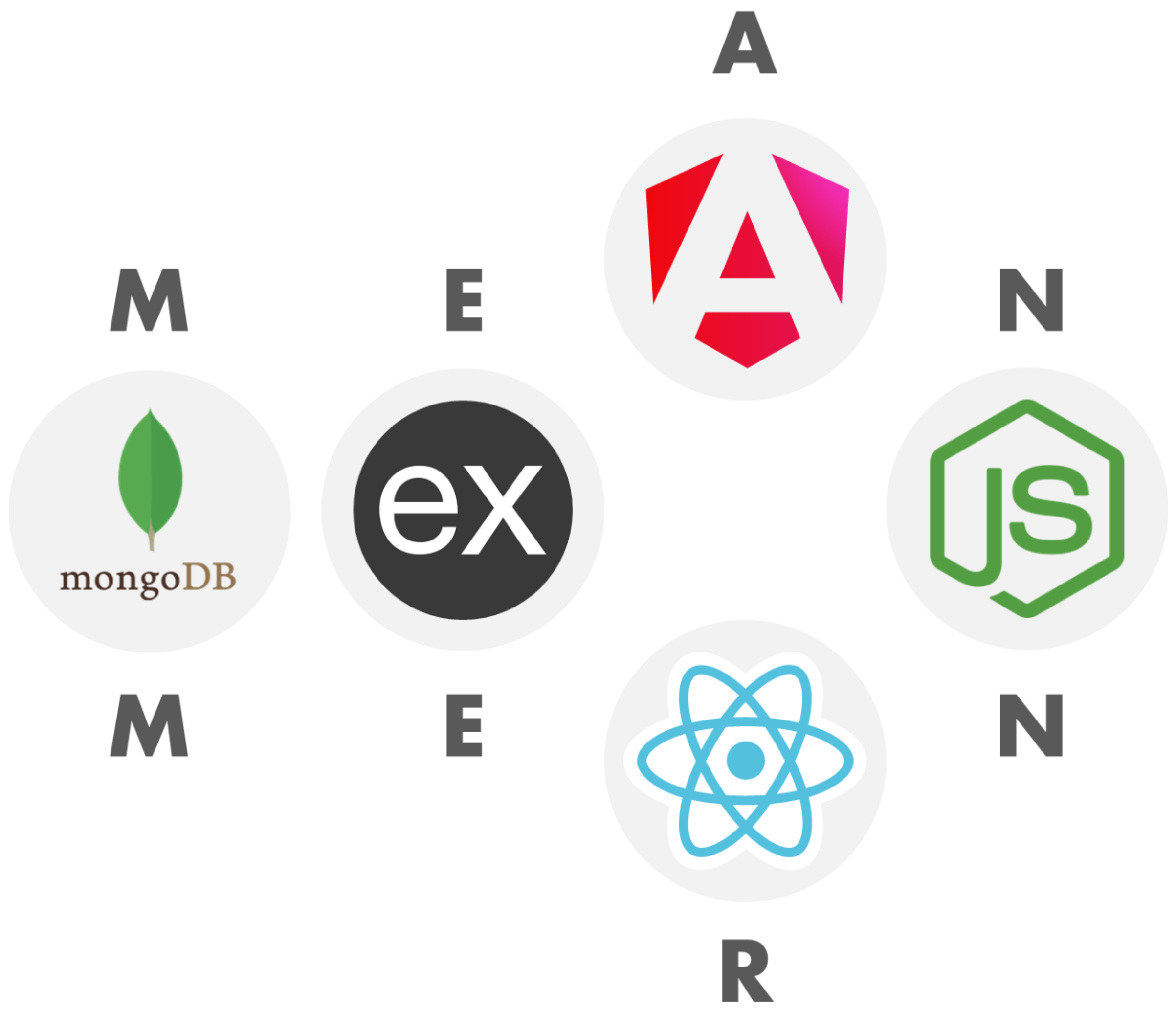
Web Technologies

Prof. Dr. Mohamed Amine Chatti
M.Sc. Shoeb Joarder

Social Computing Group, University of Duisburg-Essen
www.uni-due.de/soco

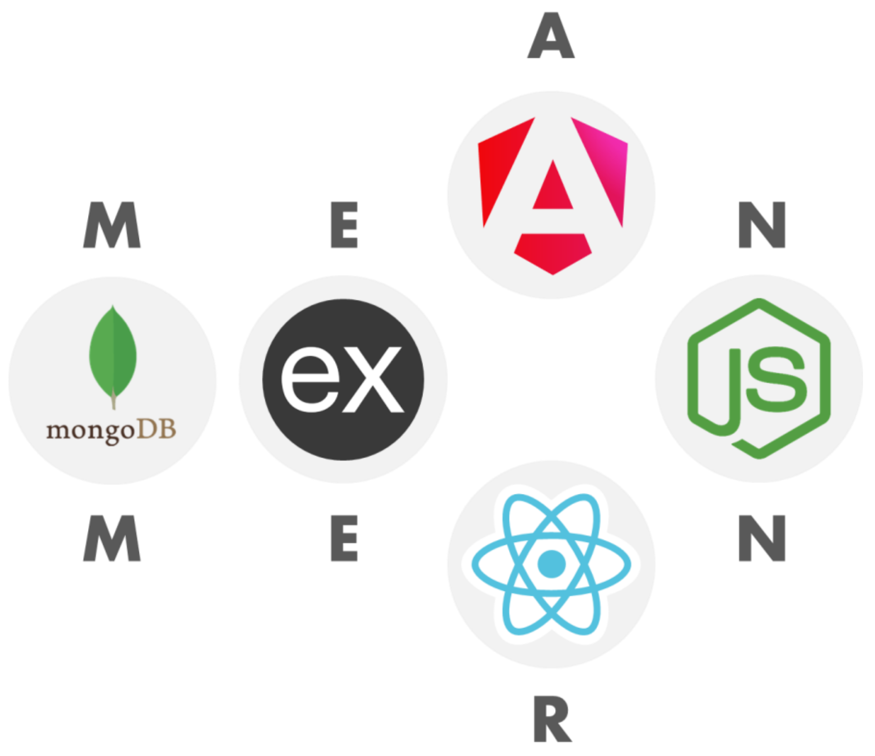
Guidelines For Project

- 10 JavaScript Guidelines
- 8 React.js Guidelines
- 5 Angular Guidelines
- 6 Node.js/Express.js Guidelines



JavaScript

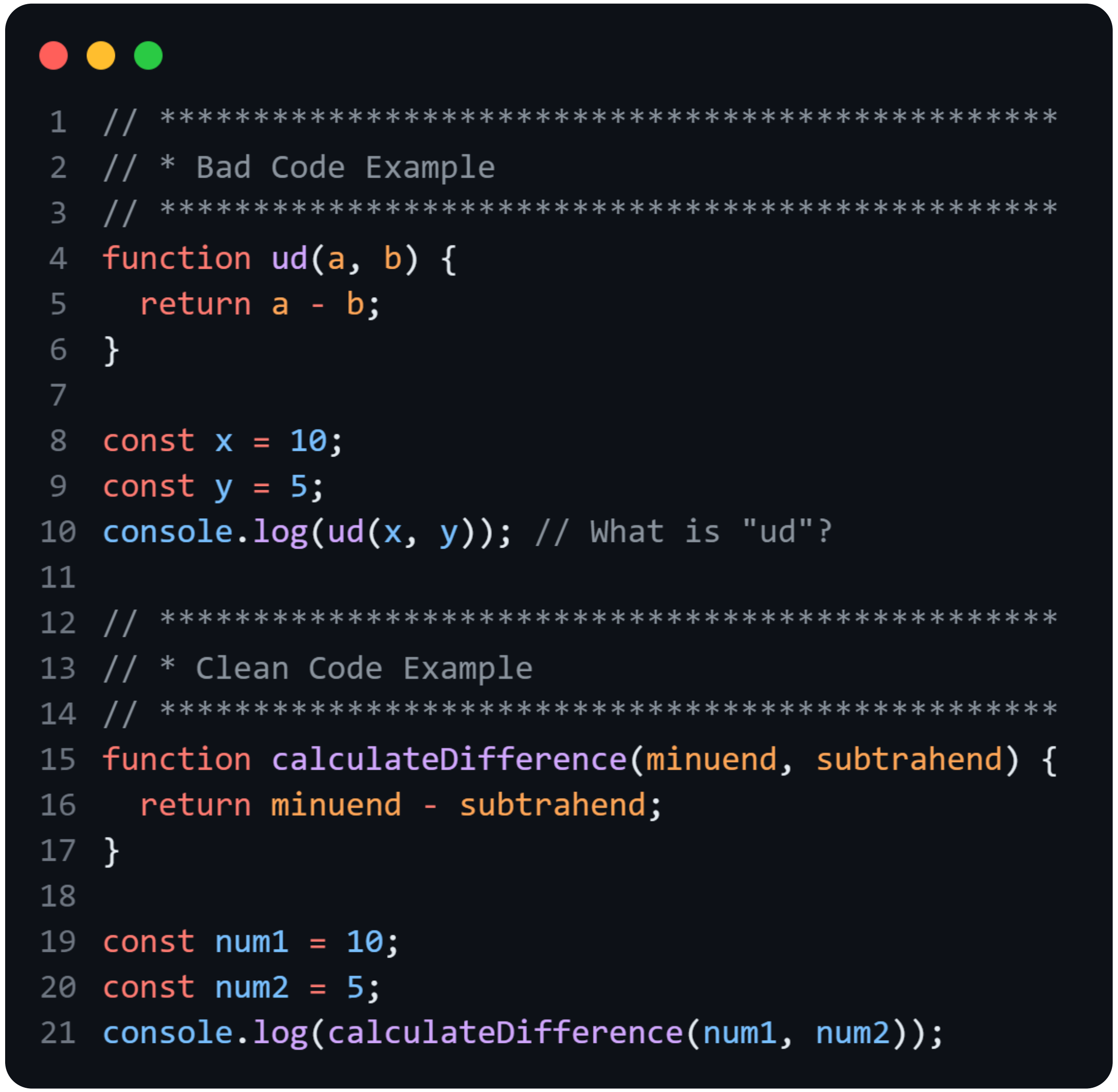
Guidelines For Project



Guideline 1 – Use Meaningful and Descriptive Names

- Description
  - Choose clear and descriptive names for variables, functions, and components to improve readability and maintainability. Avoid abbreviations and cryptic names, as they make the code harder to understand.

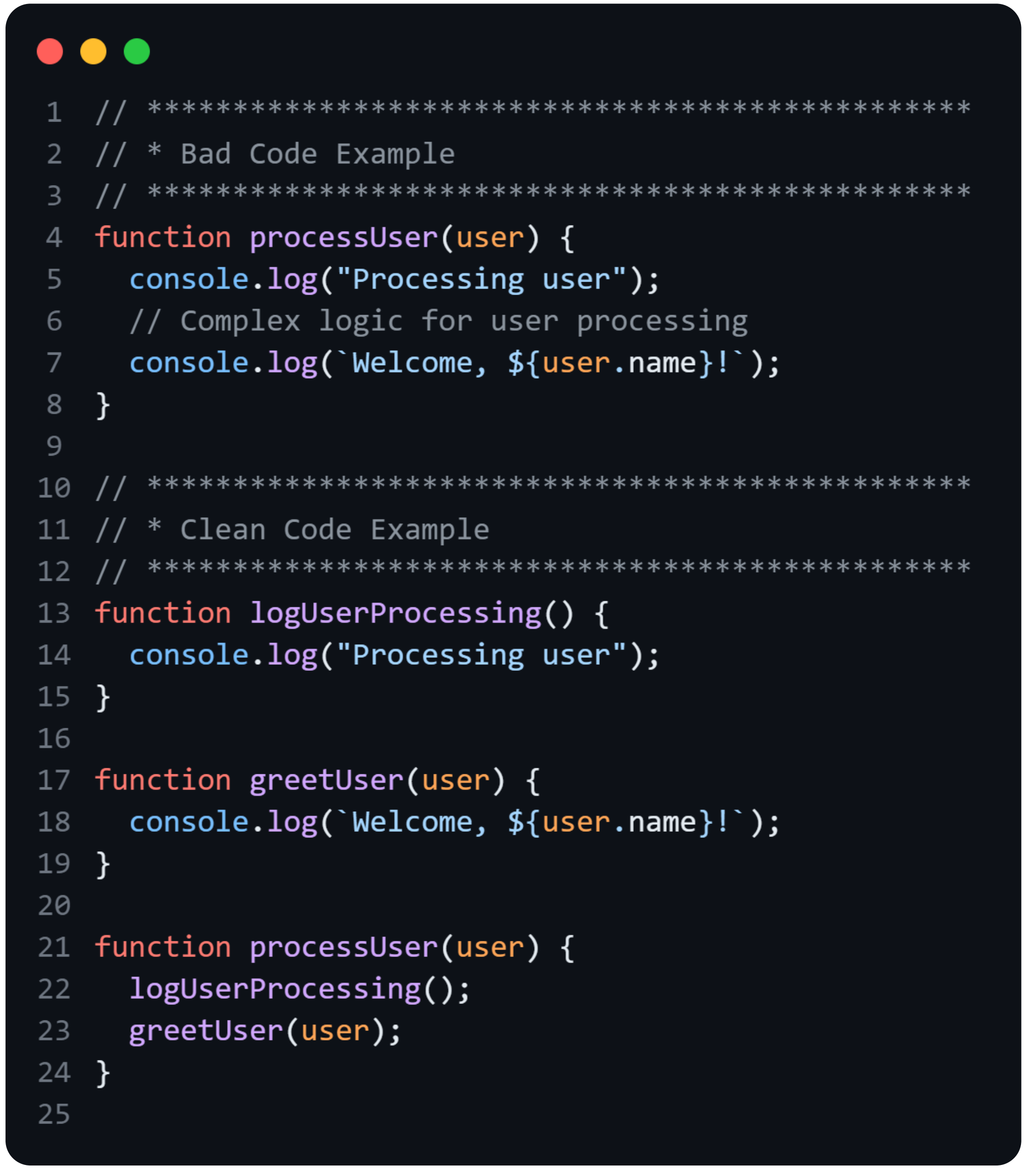
- Explanation
  - In the clean code example, calculateDifference is a meaningful name that explains what the function does. Similarly, minuend and subtrahend are descriptive names for the parameters. This helps future developers (and yourself!) quickly understand the code's purpose.



```javascript
// ***********************************************
// * Bad Code Example
// ***********************************************
function ud(a, b) {
  return a - b;
}

const x = 10;
const y = 5;
console.log(ud(x, y)); // What is "ud"?

// ***********************************************
// * Clean Code Example
// ***********************************************
function calculateDifference(minuend, subtrahend) {
  return minuend - subtrahend;
}

const num1 = 10;
const num2 = 5;
console.log(calculateDifference(num1, num2));
```

Guideline 2 – Keep Functions Small and Focused

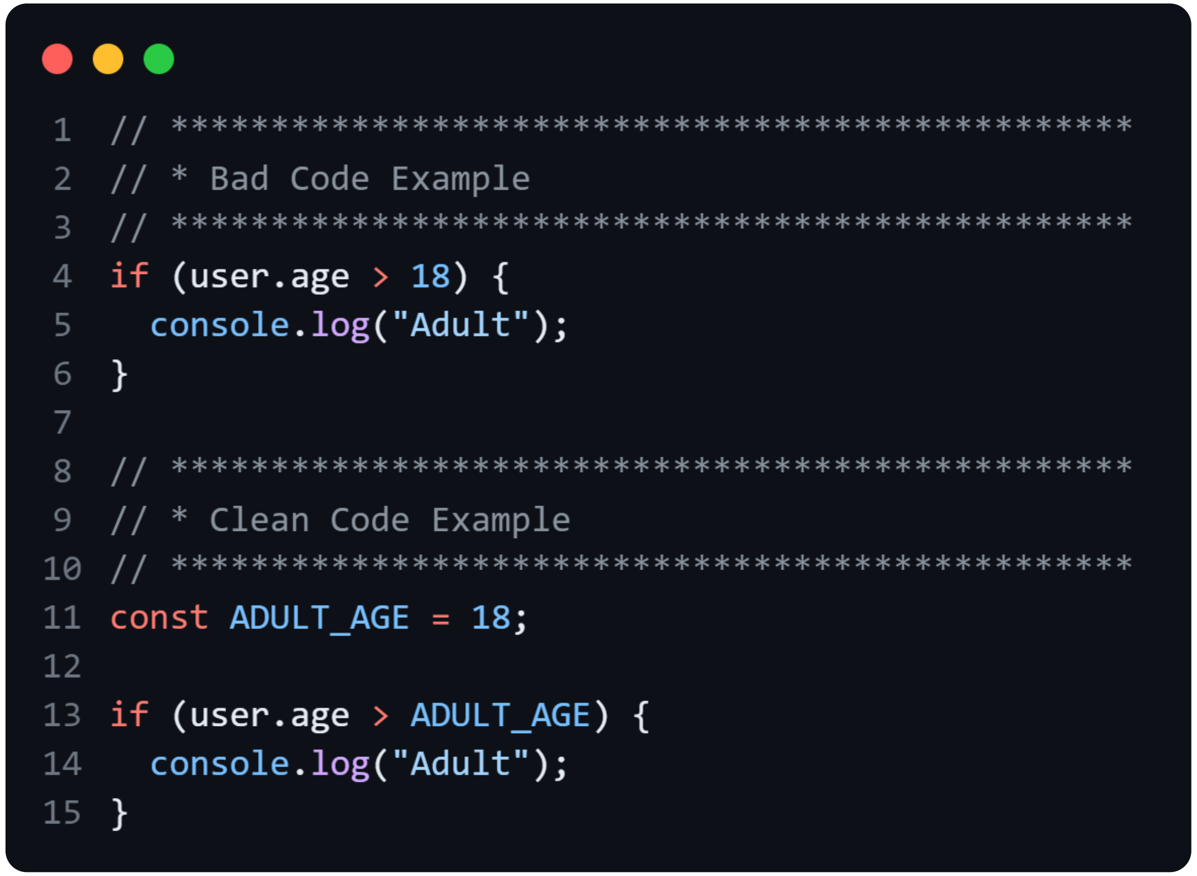
- Description
  - Functions should do one thing and do it well. Breaking down complex tasks into smaller helper functions improves readability and testability.
- Explanation
  - In the clean code example, the logic is split into smaller functions (logUserProcessing and greetUser), making it easier to understand and maintain.



```javascript
// ************************************************
// * Bad Code Example
// ************************************************
function processUser(user) {
  console.log("Processing user");
  // Complex logic for user processing
  console.log(`Welcome, ${user.name}!`);
}

// ************************************************
// * Clean Code Example
// ************************************************
function logUserProcessing() {
  console.log("Processing user");
}

function greetUser(user) {
  console.log(`Welcome, ${user.name}!`);
}

function processUser(user) {
  logUserProcessing();
  greetUser(user);
}

```

Guideline 3 – Avoid Magic Numbers and Strings

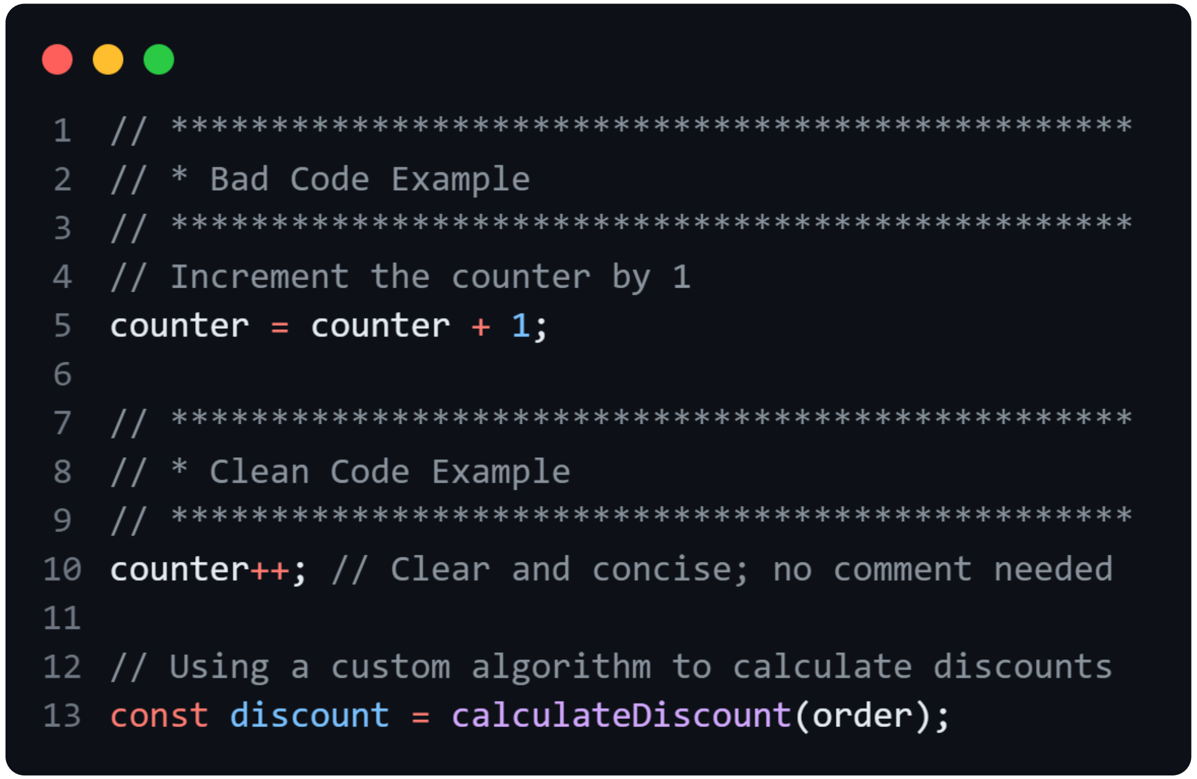
- Description
  - Replace hard-coded values ("magic numbers" or "magic strings") with named constants to improve readability and avoid errors.
- Explanation
  - The named constant ADULT_AGE clearly indicates the purpose of the value 18. This makes the code easier to update and understand.



```
// ***********************************************
// * Bad Code Example
// ***********************************************
if (user.age > 18) {
  console.log("Adult");
}

// ***********************************************
// * Clean Code Example
// ***********************************************
const ADULT_AGE = 18;

if (user.age > ADULT_AGE) {
  console.log("Adult");
}
```

Guideline 4 – Write Descriptive Comments Only When Neces

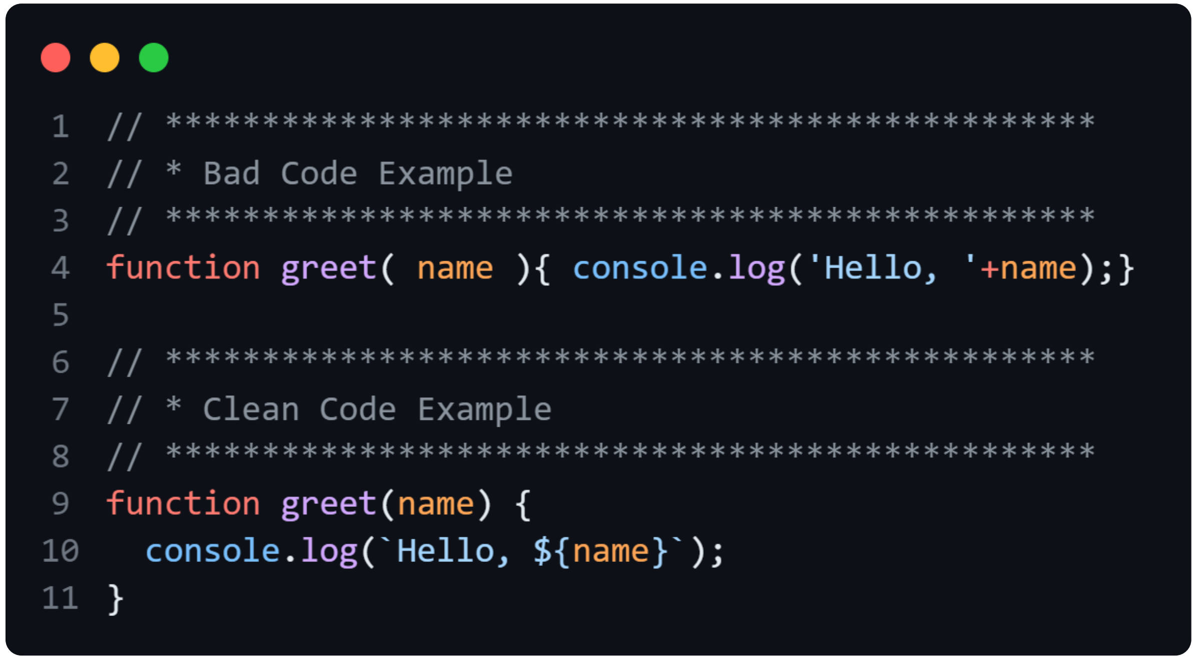
- Description
  - Write comments sparingly, focusing on clarifying complex logic or intentions. Avoid obvious comments that repeat what the code already expresses.
- Explanation
  - The clean code example avoids unnecessary comments by writing self-explanatory code. Only use comments when they provide value (e.g., explaining complex logic).



```
// *************************************************
// * Bad Code Example
// *************************************************
// Increment the counter by 1
counter = counter + 1;

// *************************************************
// * Clean Code Example
// *************************************************
counter++; // Clear and concise; no comment needed

// Using a custom algorithm to calculate discounts
const discount = calculateDiscount(order);
```

Guideline 5 – Consistently Use Code Formatting

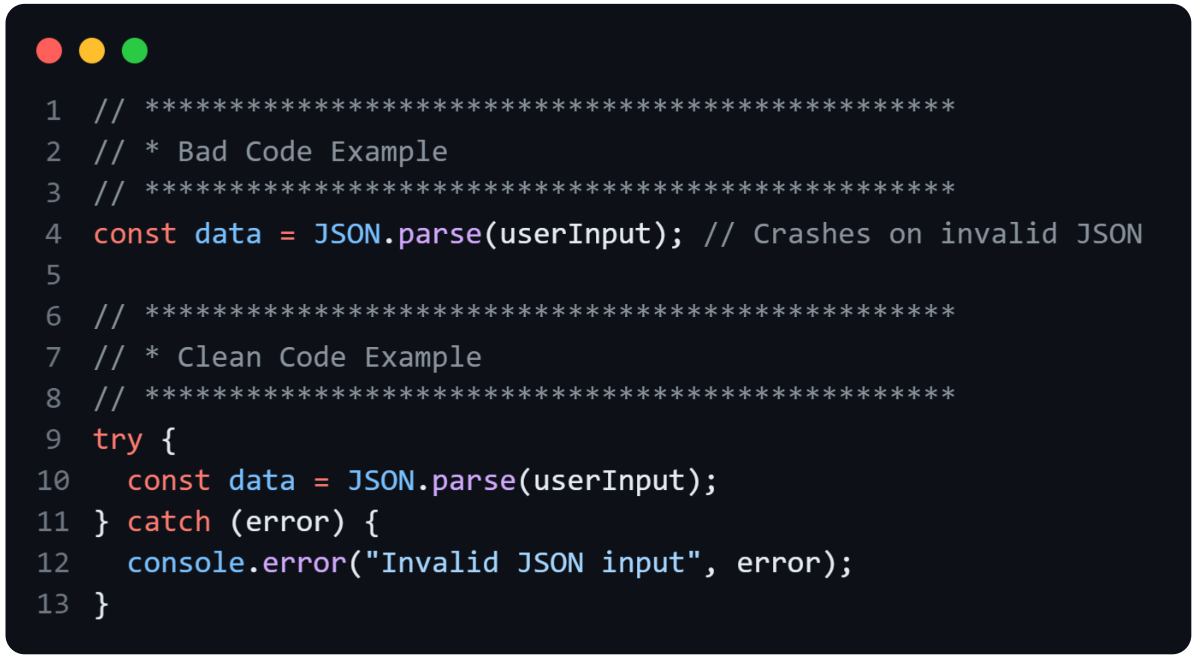
- Description
  - Adopt a consistent formatting style across the codebase to make it easier to read. Use tools like ESLint and Prettier to enforce this automatically.
- Explanation
  - Consistent formatting improves readability and ensures that everyone on the team follows the same style, reducing confusion and errors.



```
// ************************************************
// * Bad Code Example
// ************************************************
function greet( name ){ console.log('Hello, '+name);;}

// ************************************************
// * Clean Code Example
// ************************************************
function greet(name) {
  console.log(`Hello, ${name}`);
}
```

Guideline 6 – Handle Errors Gracefully

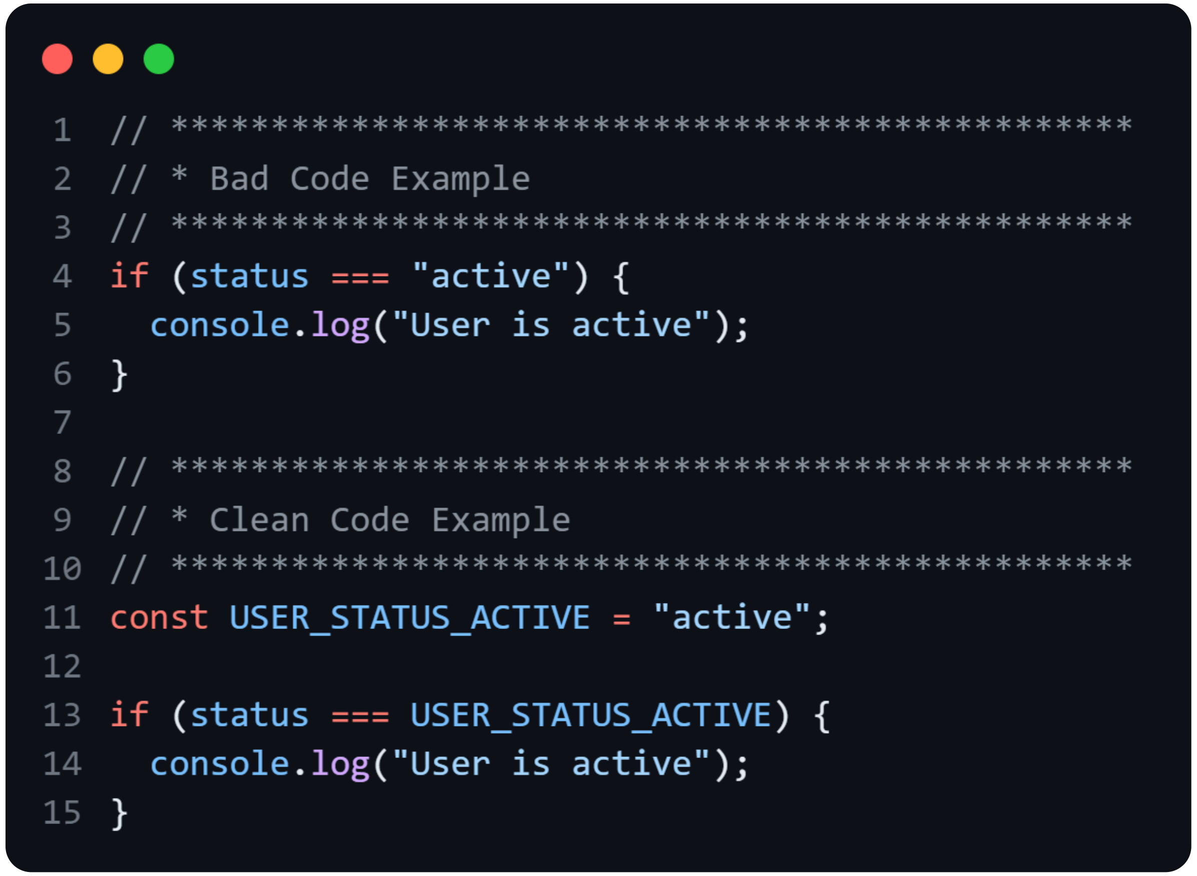
- Description
  - Always handle potential errors to prevent unexpected crashes. Use try-catch or conditionals to anticipate and manage errors.
- Explanation
  - In the clean code example, the try-catch block handles invalid JSON input gracefully, ensuring the application doesn't crash unexpectedly.



```javascript
// ************************************************
// * Bad Code Example
// ************************************************
const data = JSON.parse(userInput); // Crashes on invalid JSON

// ************************************************
// * Clean Code Example
// ************************************************
try {
  const data = JSON.parse(userInput);
} catch (error) {
  console.error("Invalid JSON input", error);
}
```

Guideline 7 – Use Constants for Repeated Values

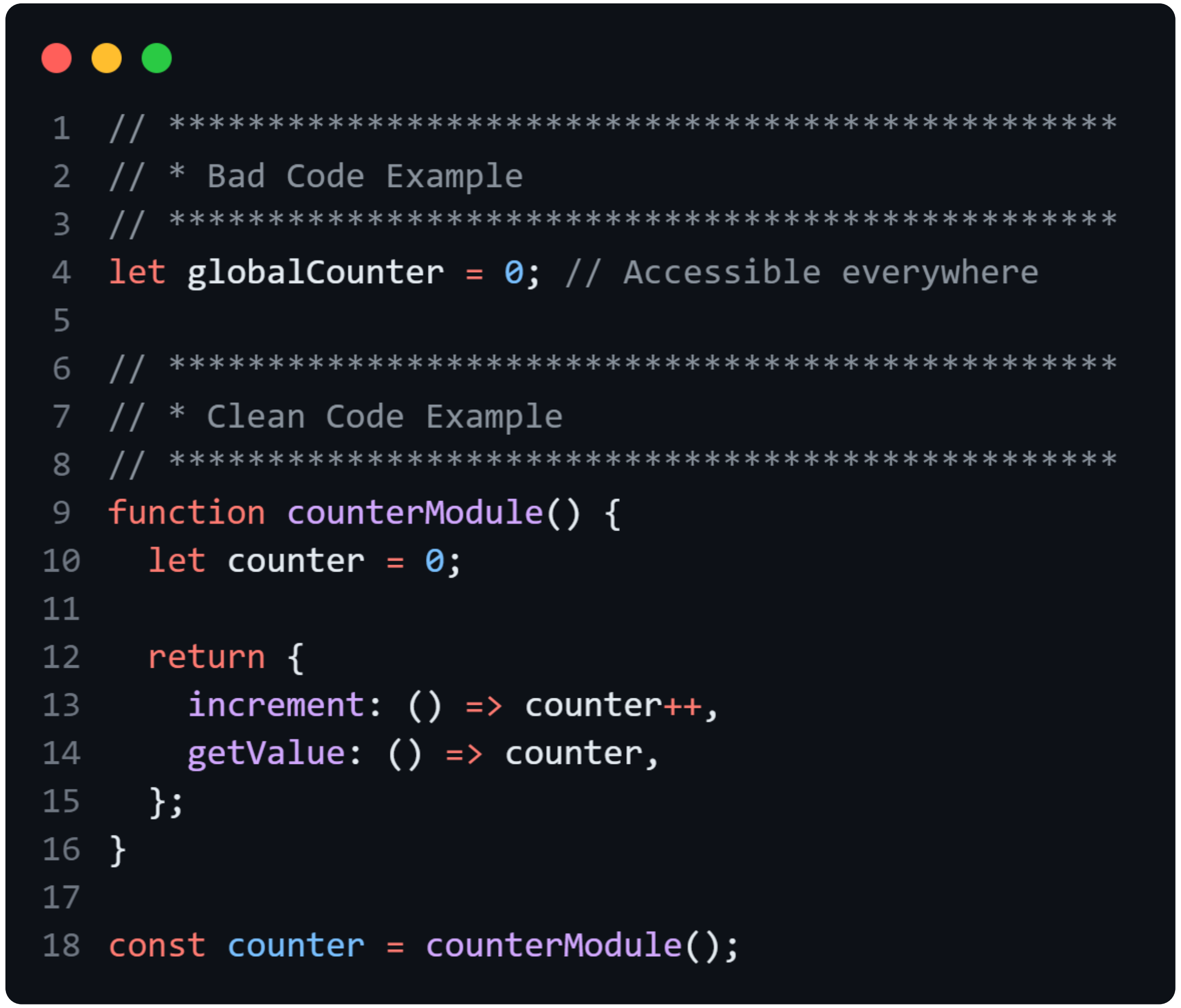
- Description
  - Extract common values into constants to avoid duplication and simplify updates.
- Explanation
  - By using USER_STATUS_ACTIVE, the value "active" is easy to update and ensures consistency throughout the codebase.



```javascript
// ************************************************
// * Bad Code Example
// ************************************************
if (status === "active") {
  console.log("User is active");
}

// ************************************************
// * Clean Code Example
// ************************************************
const USER_STATUS_ACTIVE = "active";

if (status === USER_STATUS_ACTIVE) {
  console.log("User is active");
}
```

Guideline 8 – Avoid Global Variables

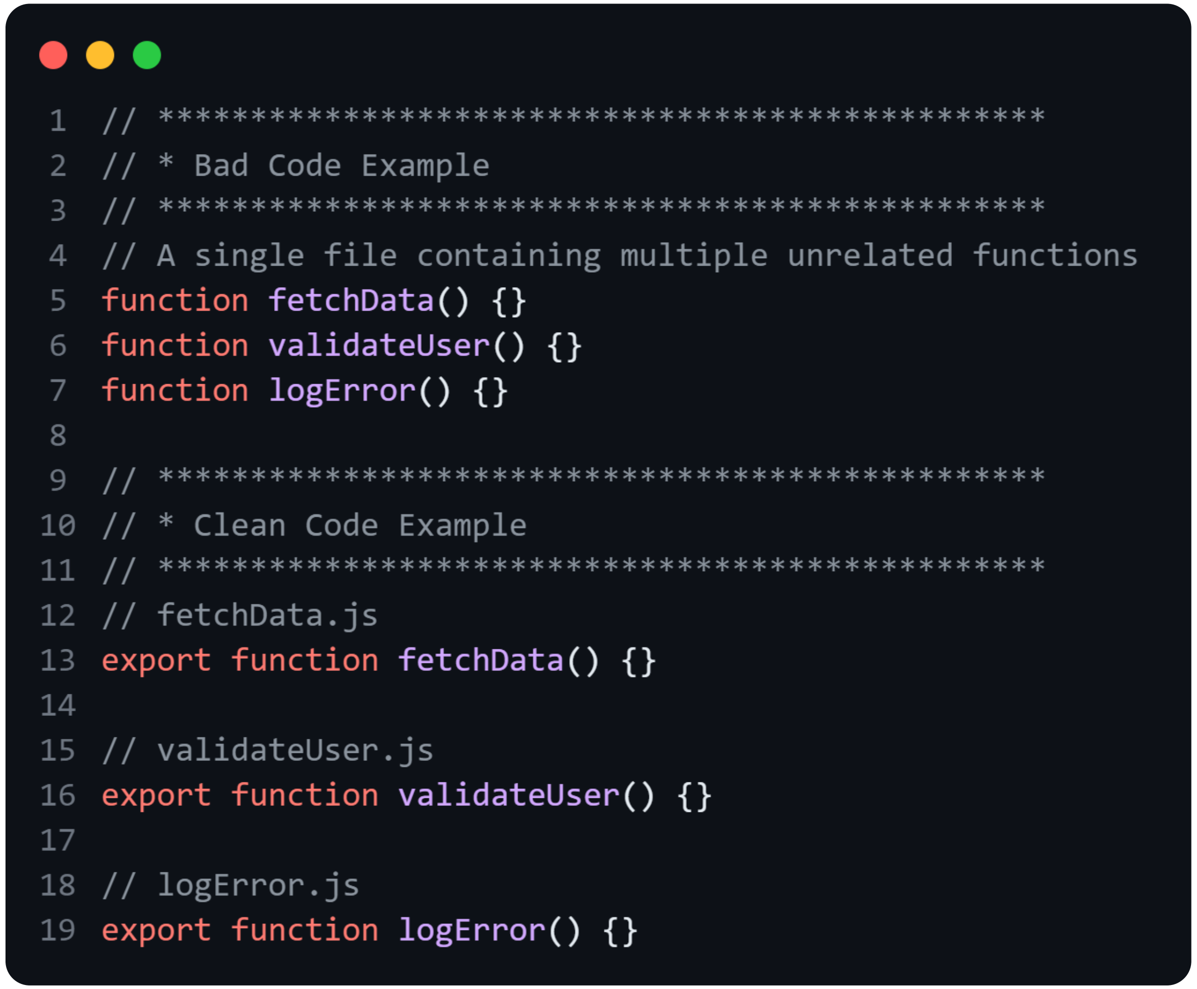
- Description
  - Minimize the use of global variables to avoid conflicts and make code more modular. Use closures or modules to encapsulate state.
- Explanation
  - The clean code example encapsulates the counter within a closure, avoiding pollution of the global namespace.



```javascript
// ************************************************
// * Bad Code Example
// ************************************************
let globalCounter = 0; // Accessible everywhere

// ************************************************
// * Clean Code Example
// ************************************************
function counterModule() {
  let counter = 0;

  return {
    increment: () => counter++,
    getValue: () => counter,
  };
}

const counter = counterModule();
```

Guideline 9 – Refactor Long Files Into Smaller Modules

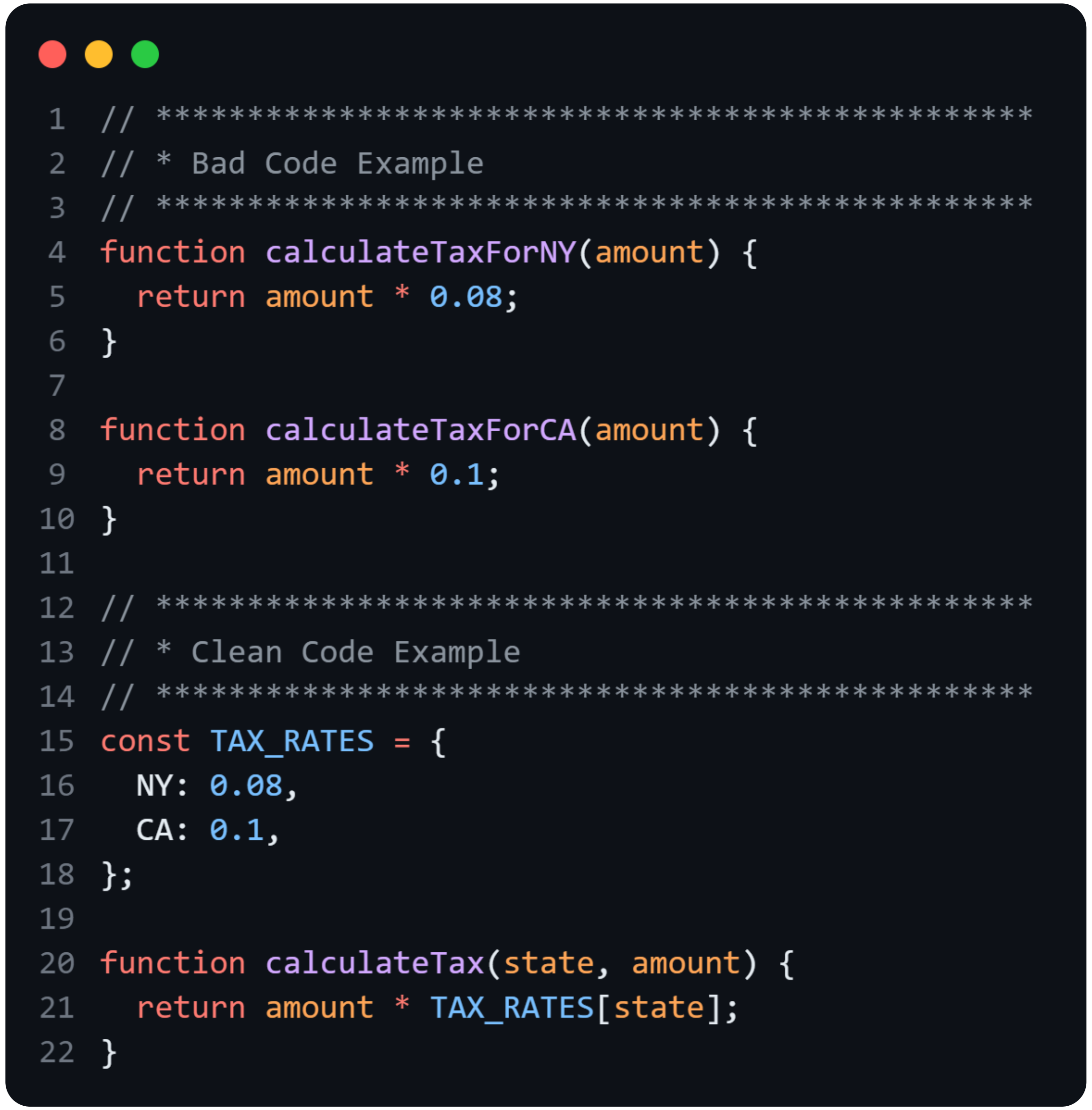
- Description
  - Split large files into smaller modules based on functionality. This improves code organization and makes it easier to find and reuse code.
- Explanation
  - By splitting functions into separate files, the clean code example ensures that each file has a single responsibility, making the code easier to maintain.



```javascript
// ************************************************
// * Bad Code Example
// ************************************************
// A single file containing multiple unrelated functions
function fetchData() {}
function validateUser() {}
function logError() {}

// ************************************************
// * Clean Code Example
// ************************************************
// fetchData.js
export function fetchData() {}

// validateUser.js
export function validateUser() {}

// logError.js
export function logError() {}
```

Guideline 10 – Adopt DRY (Don't Repeat Yourself) Principles

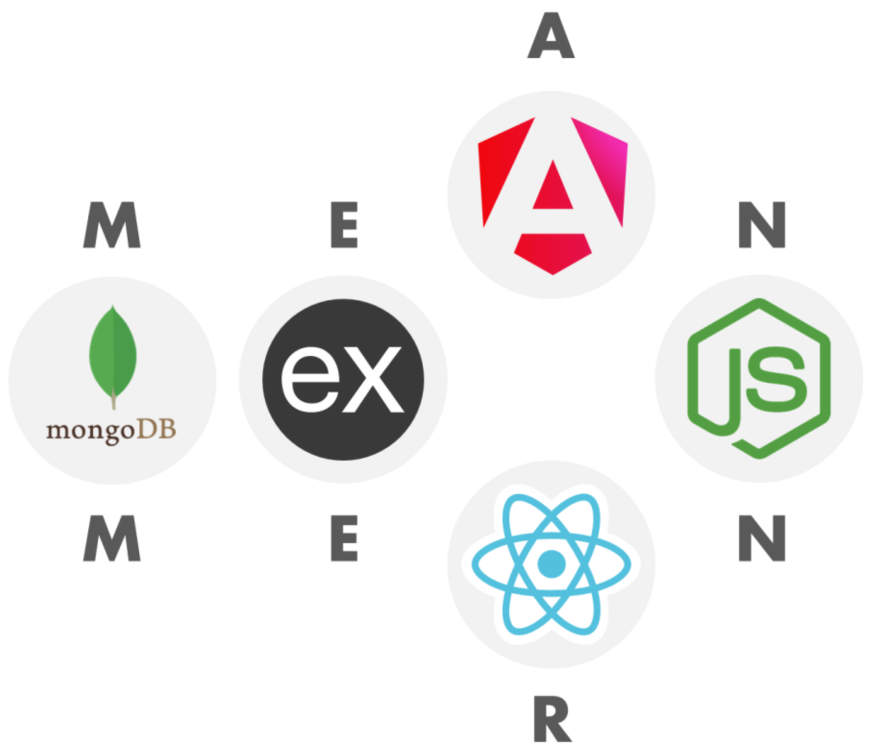
- Description
  - Avoid duplicating logic by abstracting it into reusable functions, constants, or modules.
- Explanation
  - In the clean code example, a single calculateTax function handles all states using a reusable TAX_RATES constant, eliminating redundancy.



```javascript
// ***********************************************
// * Bad Code Example
// ***********************************************
function calculateTaxForNY(amount) {
  return amount * 0.08;
}

function calculateTaxForCA(amount) {
  return amount * 0.1;
}

// ***********************************************
// * Clean Code Example
// ***********************************************
const TAX_RATES = {
  NY: 0.08,
  CA: 0.1,
};

function calculateTax(state, amount) {
  return amount * TAX_RATES[state];
}
```

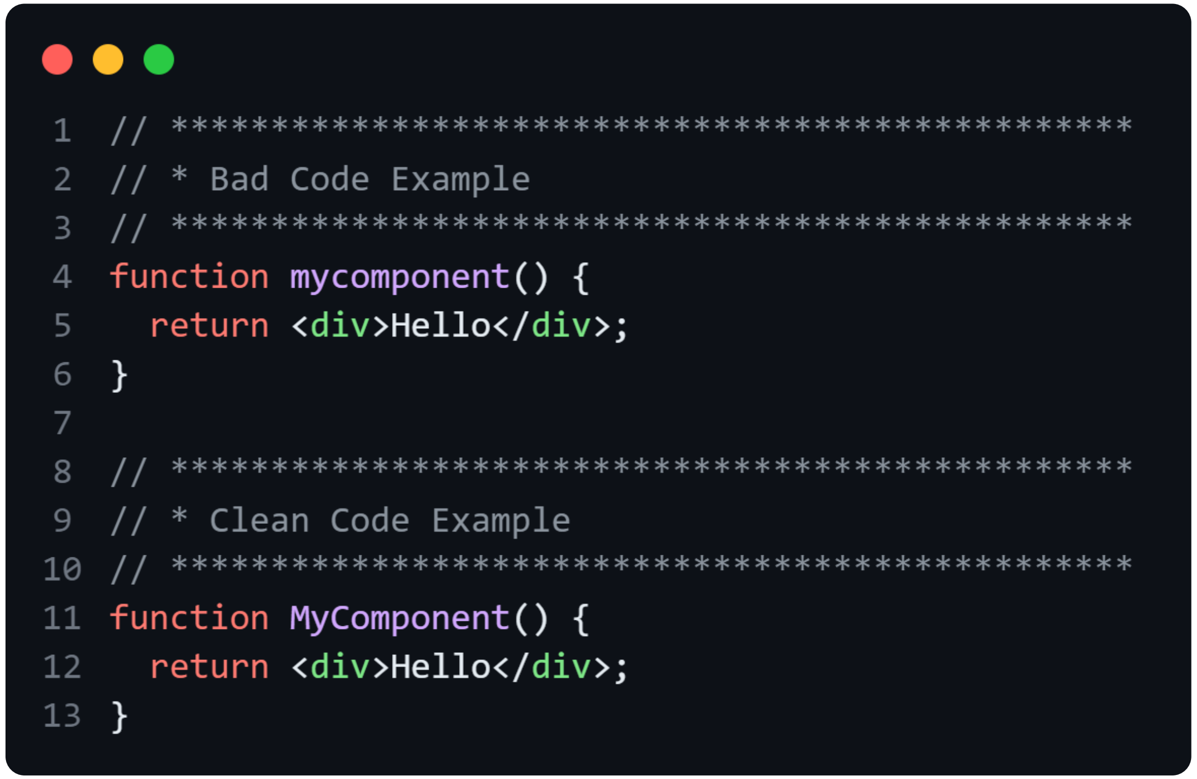
React.js

Guidelines For Project



Guideline 1 – Name Components with PascalCase

- Description
  - React components should use PascalCase naming (e.g., MyComponent) to distinguish them from regular HTML tags, which use lowercase.
- Explanation
  - Using PascalCase aligns with React conventions and makes the component easily identifiable in JSX.

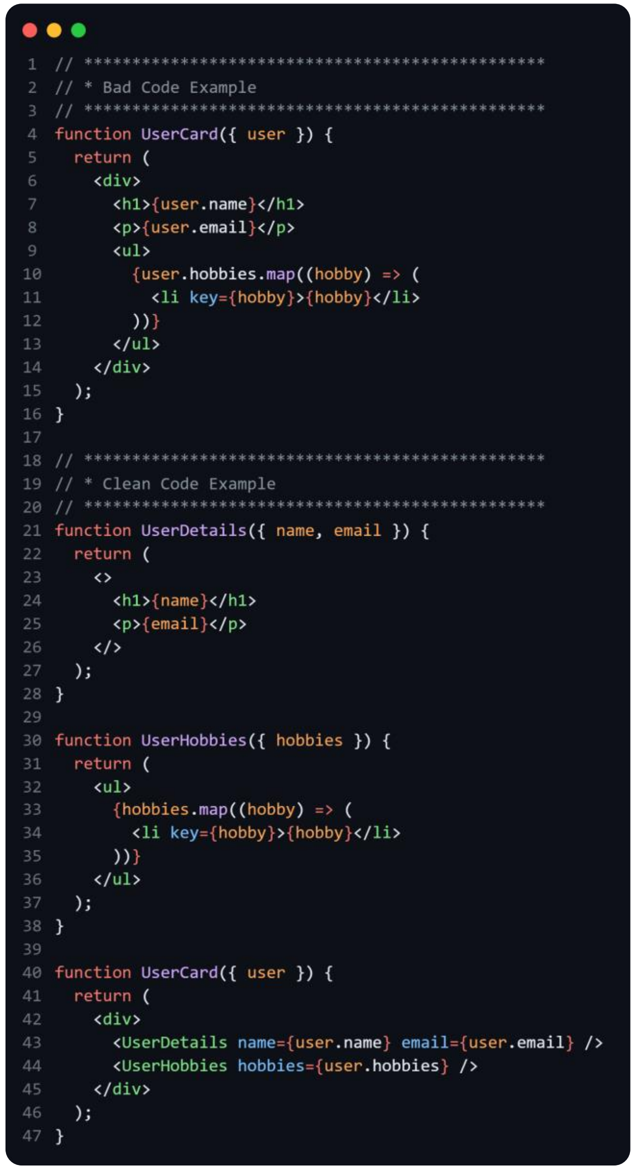


```
// ****************************************************
// * Bad Code Example
// ****************************************************
function mycomponent() {
  return <div>Hello</div>;
}

// ****************************************************
// * Clean Code Example
// ****************************************************
function MyComponent() {
  return <div>Hello</div>;
}
```

Guideline 2 – Keep Components Small and Reusable

- Description
  - Break down large components into smaller, reusable components. Each component should handle a single responsibility.

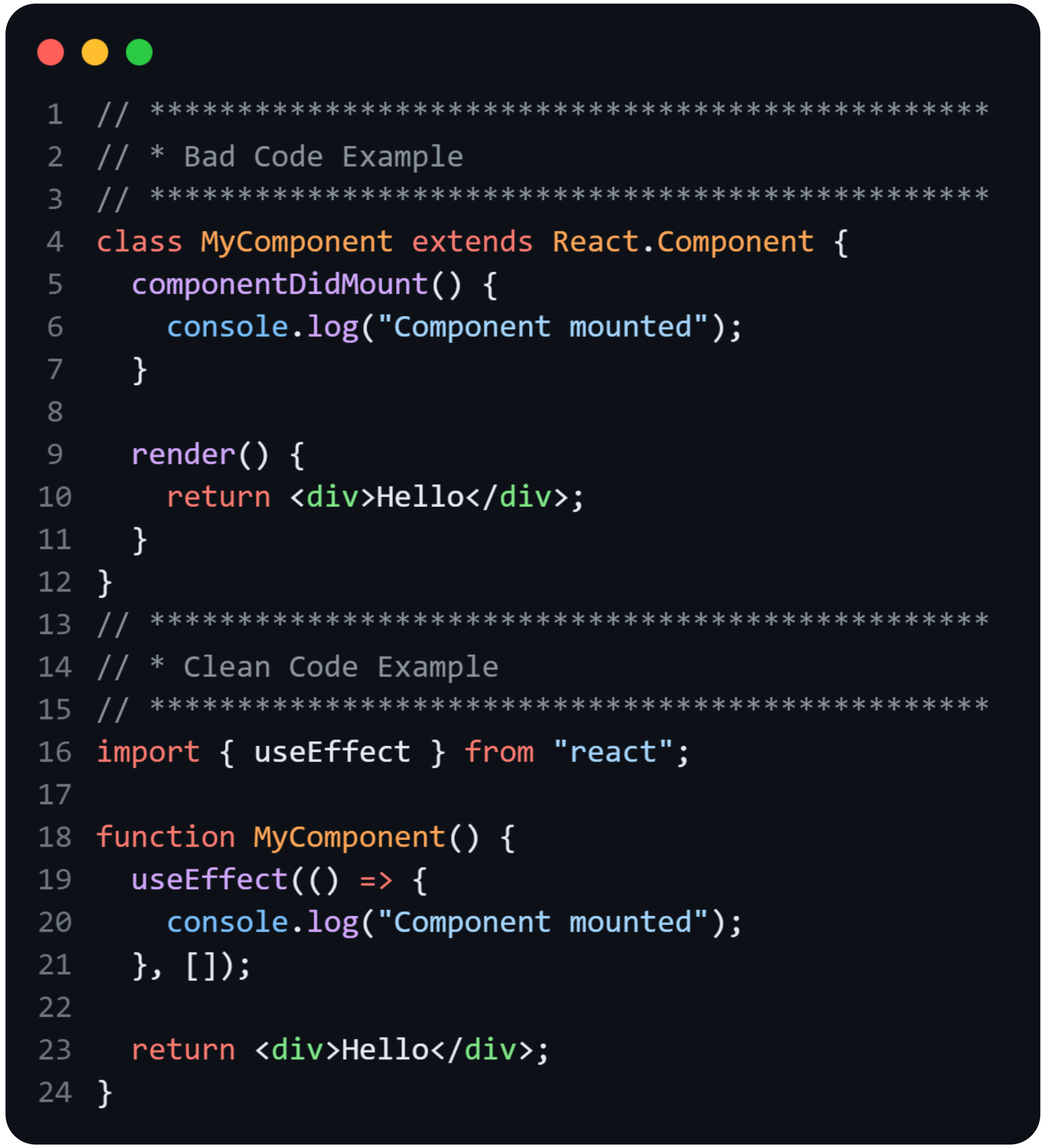
- Explanation
  - The clean code example separates the UserDetails and UserHobbies logic, making them reusable and simplifying the main UserCard component.



```jsx
// **************************************************
// * Bad Code Example
// **************************************************
function UserCard({ user }) {
  return (
    <div>
      <h1>{user.name}</h1>
      <p>{user.email}</p>
      <ul>
        {user.hobbies.map((hobby) => (
          <li key={hobby}>{hobby}</li>
        ))}
      </ul>
    </div>
  );
}

// **************************************************
// * Clean Code Example
// **************************************************
function UserDetails({ name, email }) {
  return (
    <>
      <h1>{name}</h1>
      <p>{email}</p>
    </>
  );
}

function UserHobbies({ hobbies }) {
  return (
    <ul>
      {hobbies.map((hobby) => (
        <li key={hobby}>{hobby}</li>
      ))}
    </ul>
  );
}

function UserCard({ user }) {
  return (
    <div>
      <UserDetails name={user.name} email={user.email} />
      <UserHobbies hobbies={user.hobbies} />
    </div>
  );
}
```

Guideline 3 – Use Hooks for State and Side Effects

- Description
  - Use React Hooks (e.g., useState, useEffect) to manage component state and lifecycle events in functional components.
- Explanation
  - Hooks simplify the code by eliminating the need for class components and directly managing side effects in functional components.

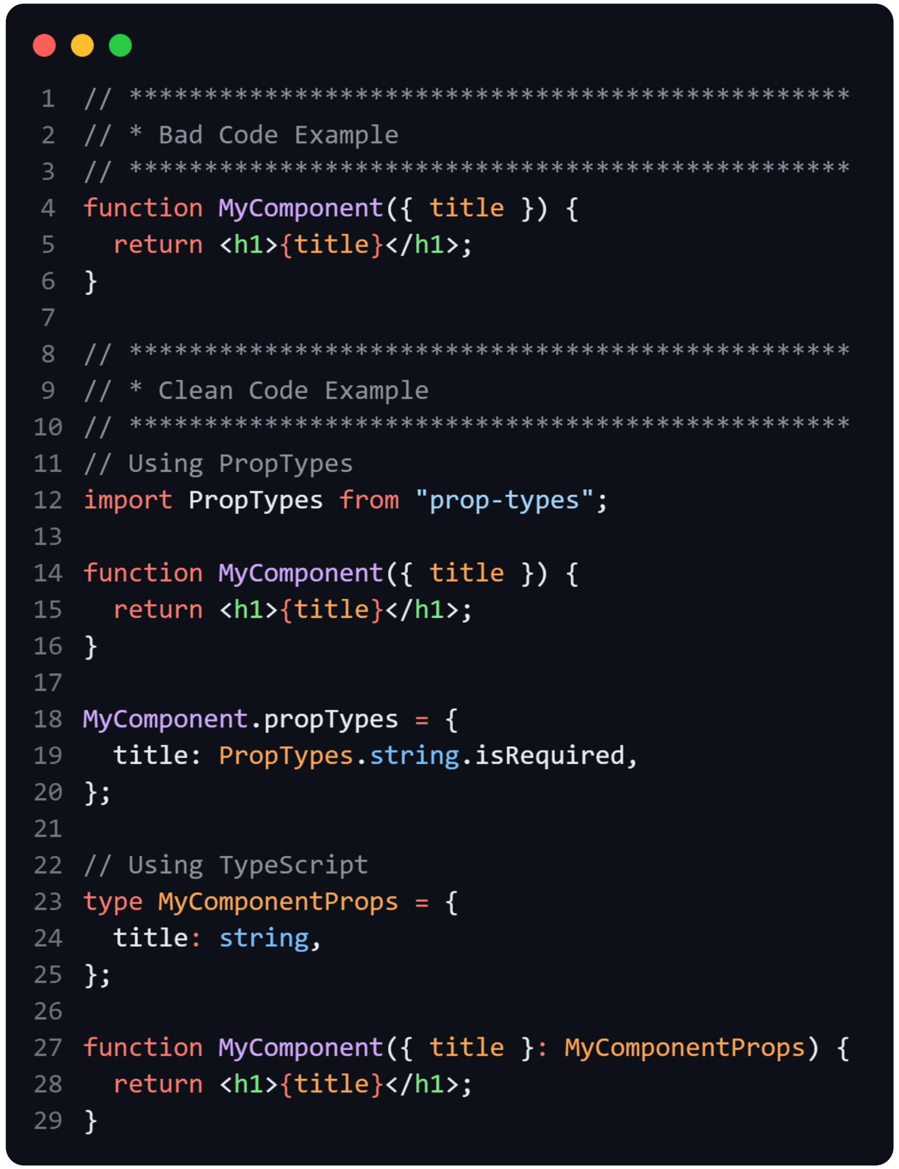


```
// ***********************************************
// * Bad Code Example
// ***********************************************
class MyComponent extends React.Component {
  componentDidMount() {
    console.log("Component mounted");
  }

  render() {
    return <div>Hello</div>;
  }
}
// ***********************************************
// * Clean Code Example
// ***********************************************
import { useEffect } from "react";

function MyComponent() {
  useEffect(() => {
    console.log("Component mounted");
  }, []);

  return <div>Hello</div>;
}
```

Guideline 4 – Use PropTypes or TypeScript for Type Checking

- Description
  - Use PropTypes or TypeScript to enforce type checking for component props, reducing bugs and improving documentation.

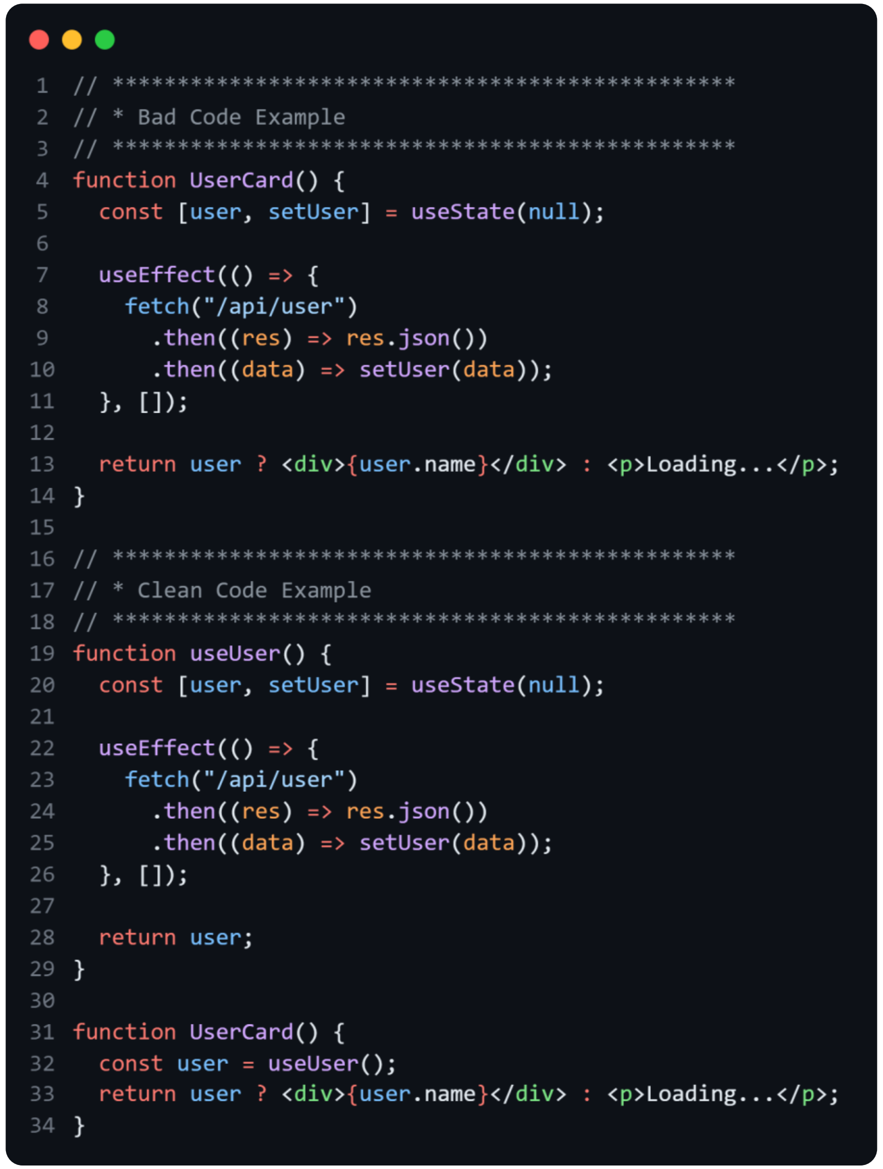
- Explanation
  - Type checking ensures that components receive the correct data, helping to catch issues early during development.



```
// **************************************************
// * Bad Code Example
// **************************************************
function MyComponent({ title }) {
  return <h1>{title}</h1>;
}

// **************************************************
// * Clean Code Example
// **************************************************
// Using PropTypes
import PropTypes from "prop-types";

function MyComponent({ title }) {
  return <h1>{title}</h1>;
}

MyComponent.propTypes = {
  title: PropTypes.string.isRequired,
};

// Using TypeScript
type MyComponentProps = {
  title: string,
};

function MyComponent({ title }: MyComponentProps) {
  return <h1>{title}</h1>;
}
```

Guideline 5 – Separate Presentation and Logic

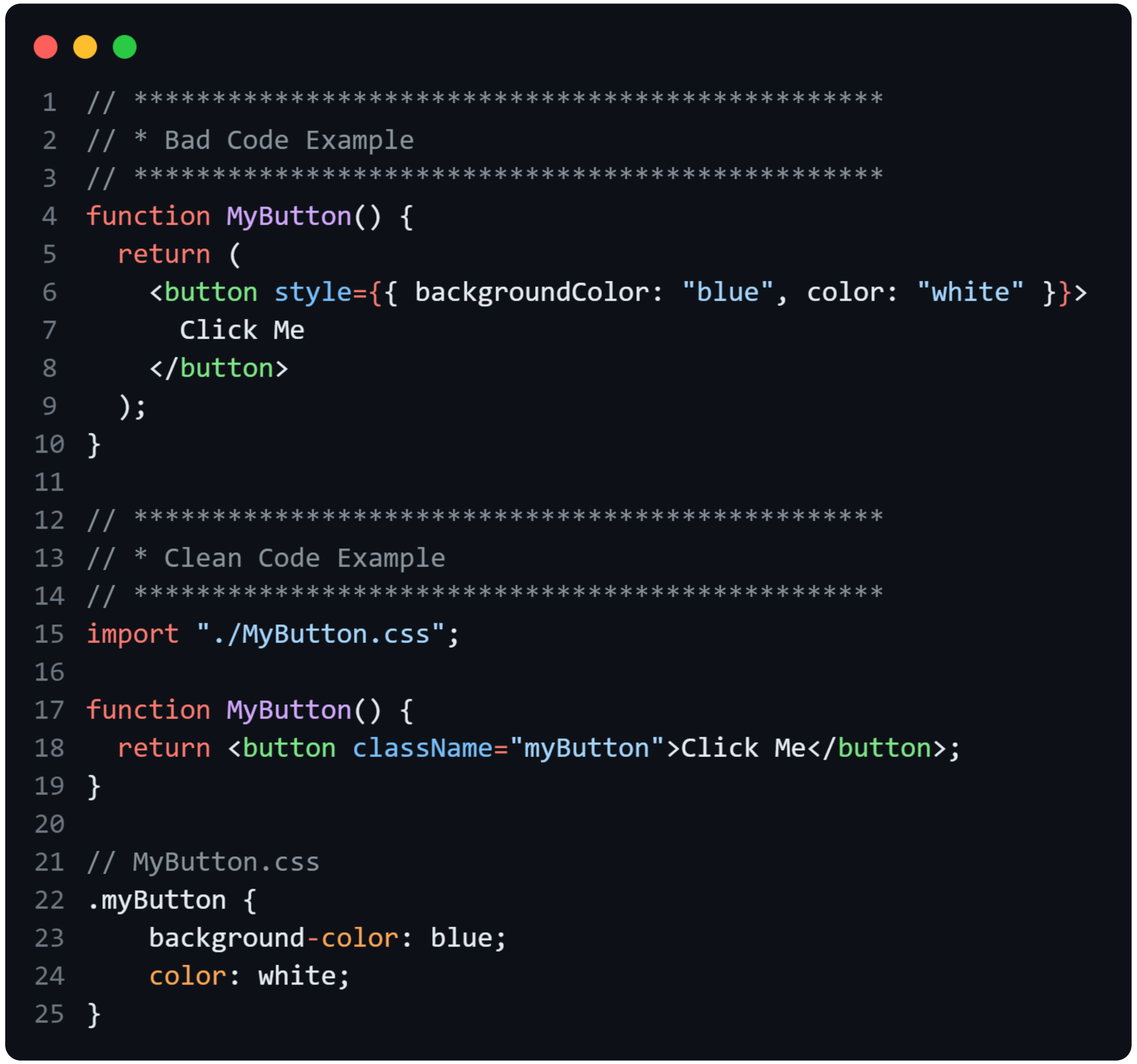
- Description
  - Keep UI (presentation) and logic (data fetching, state management) separate to improve readability and reusability.
- Explanation
  - The clean code example separates the data fetching logic into a custom hook (useUser), making the UserCard component focused on presentation.



```
// ***************************************************
// * Bad Code Example
// ***************************************************
function UserCard() {
  const [user, setUser] = useState(null);

  useEffect(() => {
    fetch("/api/user")
      .then((res) => res.json())
      .then((data) => setUser(data));
  }, []);

  return user ? <div>{user.name}</div> : <p>Loading...</p>;
}

// ***************************************************
// * Clean Code Example
// ***************************************************
function useUser() {
  const [user, setUser] = useState(null);

  useEffect(() => {
    fetch("/api/user")
      .then((res) => res.json())
      .then((data) => setUser(data));
  }, []);

  return user;
}

function UserCard() {
  const user = useUser();
  return user ? <div>{user.name}</div> : <p>Loading...</p>;
}
```

Guideline 6 – Avoid Inline Styles (Use Styled Components or CSS Modu

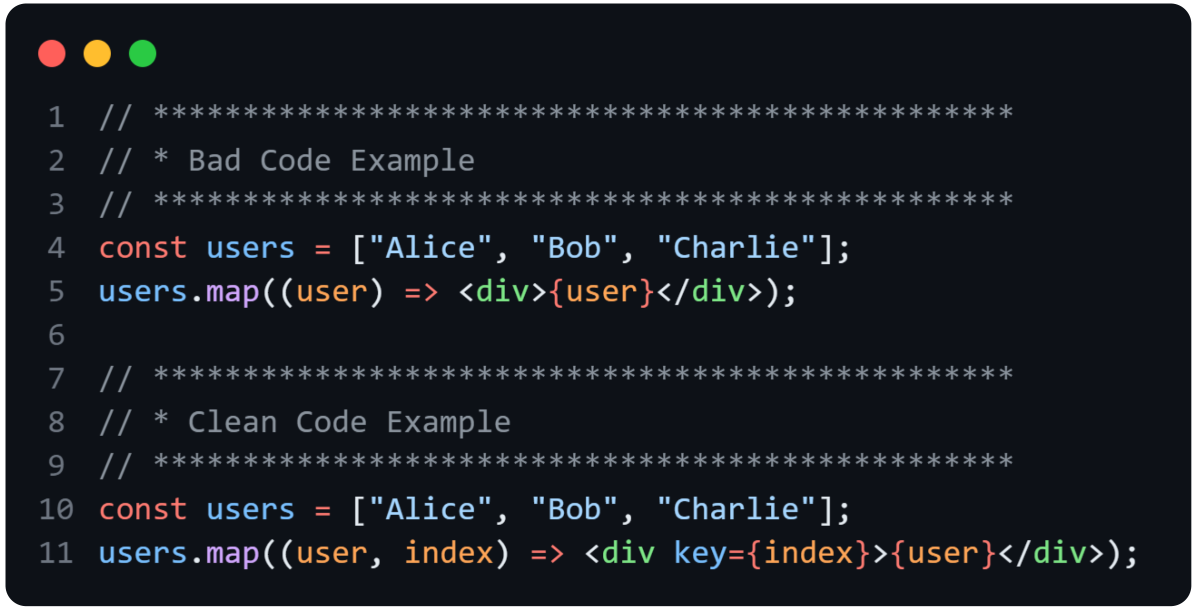
- Description
  - Avoid using inline styles directly in JSX. Use libraries like styled-components or CSS Modules for styling to keep your styles organized.
- Explanation
  - Separating styles into a CSS file or using CSS-in-JS libraries improves readability and reusability of your code.



```
// ************************************************
// * Bad Code Example
// ************************************************
function MyButton() {
  return (
    <button style={{ backgroundColor: "blue", color: "white" }}>
      Click Me
    </button>
  );
}

// ************************************************
// * Clean Code Example
// ************************************************
import "./MyButton.css";

function MyButton() {
  return <button className="myButton">Click Me</button>;
}

// MyButton.css
.myButton {
    background-color: blue;
    color: white;
}
```

Guideline 7 – Use Meaningful Key Props for Lists

- Description
  - When rendering lists, always provide a unique and stable key prop to help React optimize rendering and avoid bugs.
- Explanation
  - The clean code example includes a key prop, ensuring React tracks individual elements correctly during updates.

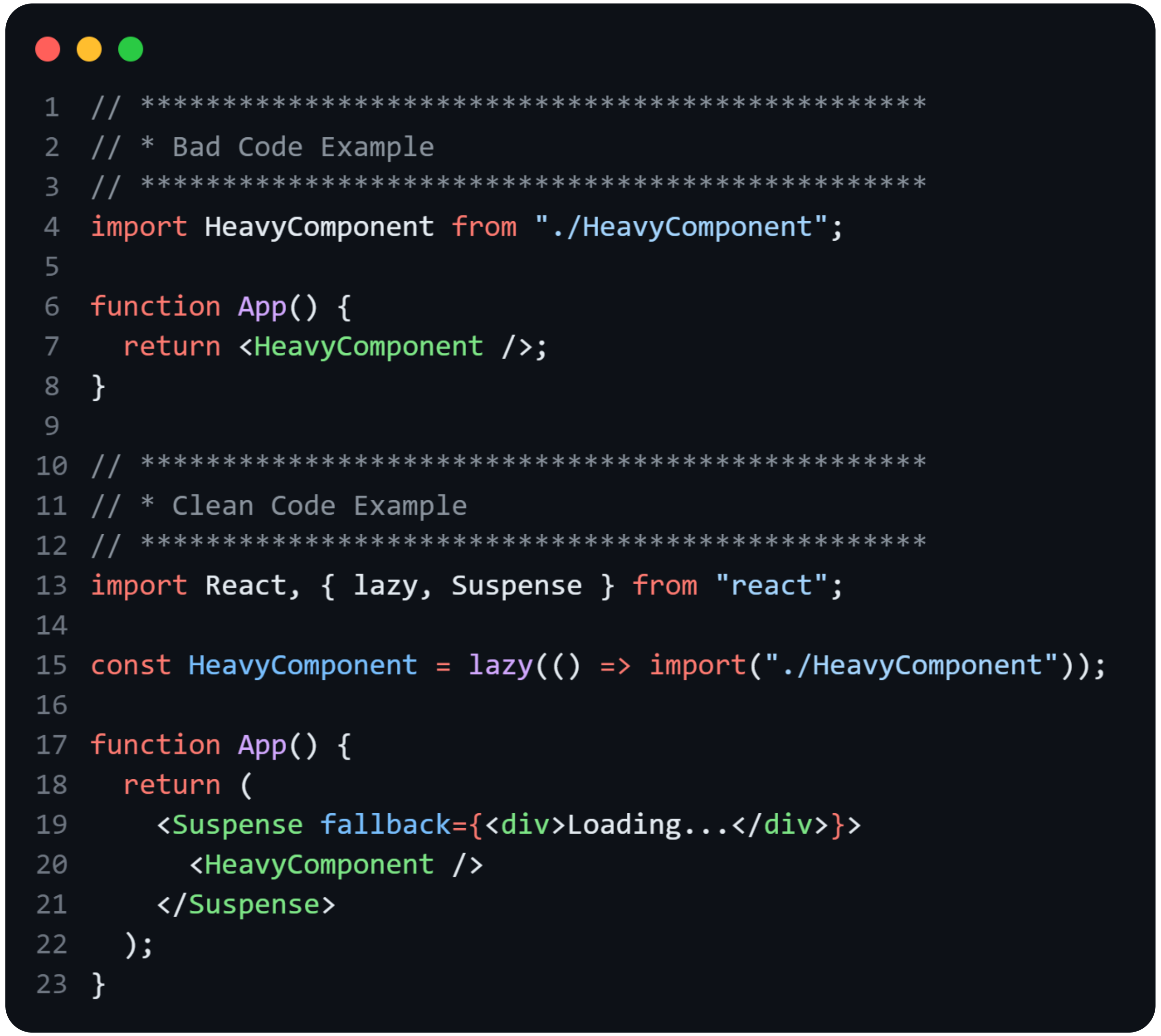


```
// ************************************************
// * Bad Code Example
// ************************************************
const users = ["Alice", "Bob", "Charlie"];
users.map((user) => <div>{user}</div>);

// ************************************************
// * Clean Code Example
// ************************************************
const users = ["Alice", "Bob", "Charlie"];
users.map((user, index) => <div key={index}>{user}</div>);
```

Guideline 8 – Lazy Load Heavy Components

- Description
  - Use React's lazy and Suspense to load large components only when needed, improving initial load time.

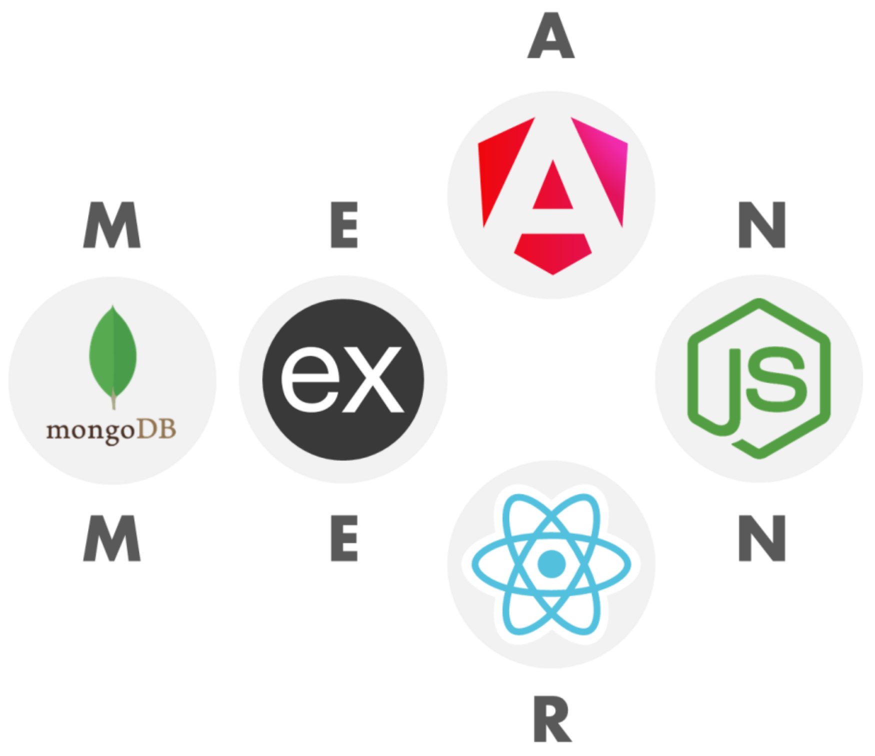
- Explanation
  - The clean code example defers loading HeavyComponent until it is actually rendered, reducing the initial bundle size and improving performance.



```
// ************************************************
// * Bad Code Example
// ************************************************
import HeavyComponent from "./HeavyComponent";

function App() {
  return <HeavyComponent />;
}

// ************************************************
// * Clean Code Example
// ************************************************
import React, { lazy, Suspense } from "react";

const HeavyComponent = lazy(() => import("./HeavyComponent"));

function App() {
  return (
    <Suspense fallback={<div>Loading...</div>}>
      <HeavyComponent />
    </Suspense>
  );
}
```

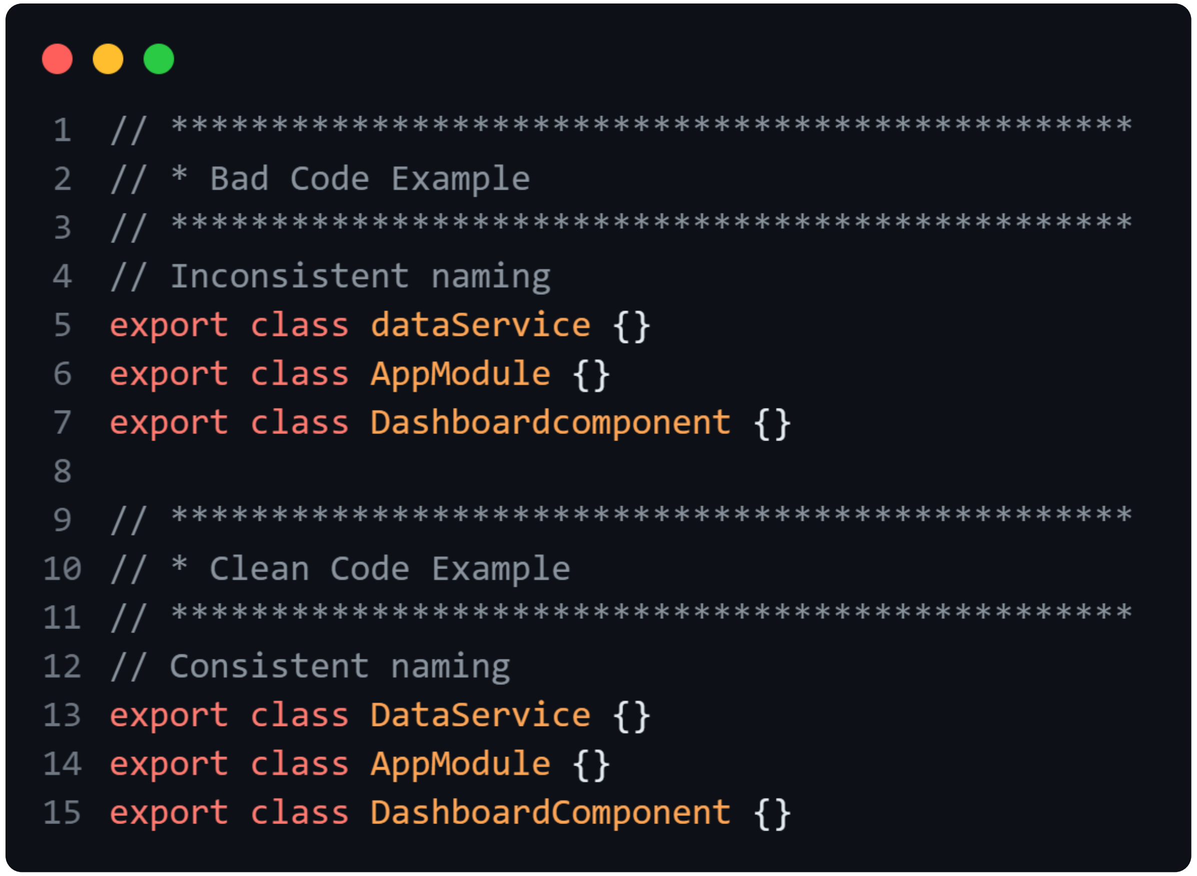
Angular

Guidelines For Project



Guideline 1 – Name Services, Components, and Modules Consistently

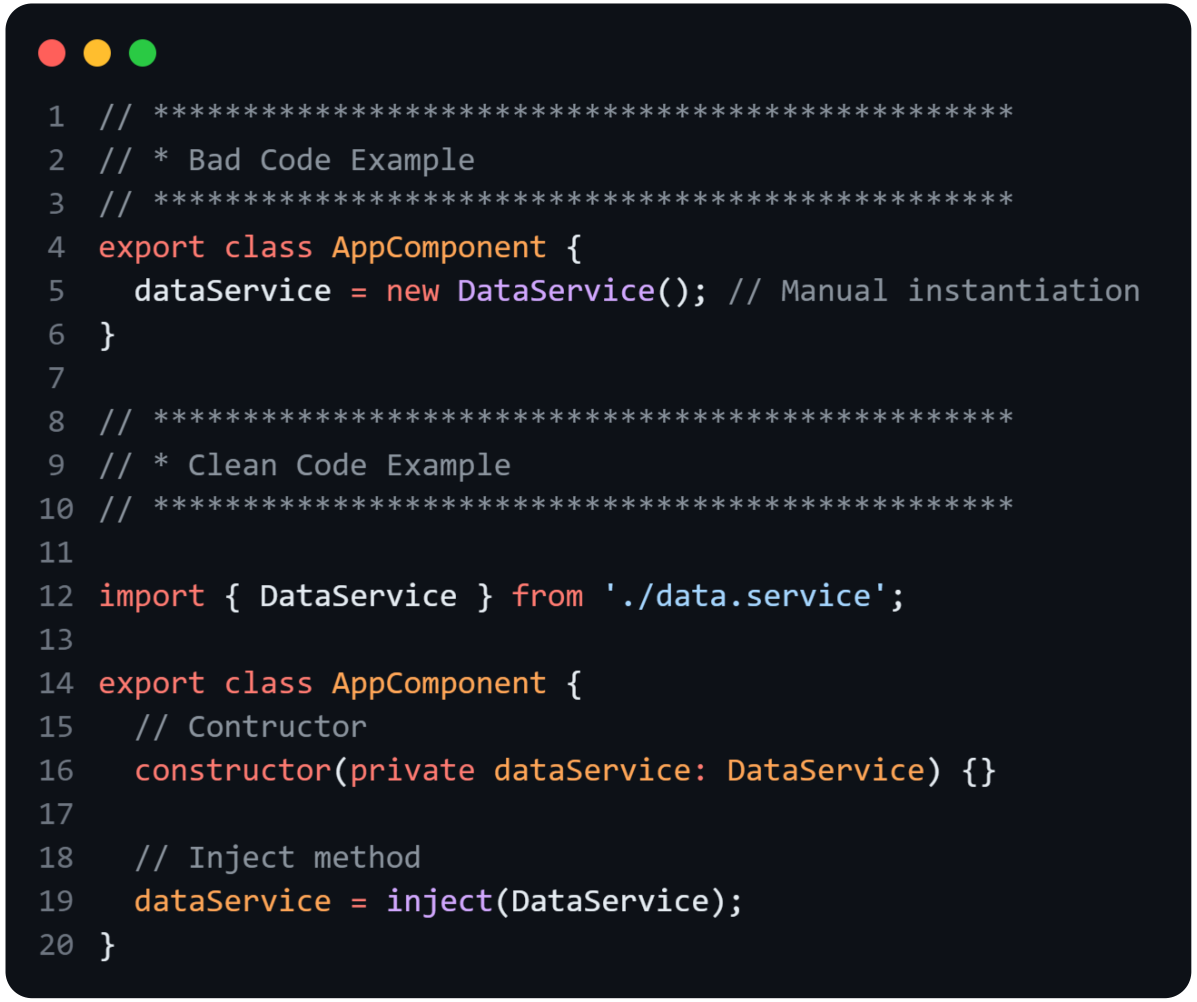
- Description
  - Use consistent and descriptive naming conventions for Angular artifacts like components, services, and modules.
- Explanation
  - The clean code example follows Angular's naming conventions (e.g., PascalCase for classes and modules), making the codebase easier to read and maintain.



```
// ************************************************
// * Bad Code Example
// ************************************************
// Inconsistent naming
export class dataService {}
export class AppModule {}
export class Dashboardcomponent {}

// ************************************************
// * Clean Code Example
// ************************************************
// Consistent naming
export class DataService {}
export class AppModule {}
export class DashboardComponent {}
```

Guideline 2 – Use Dependency Injection Properly

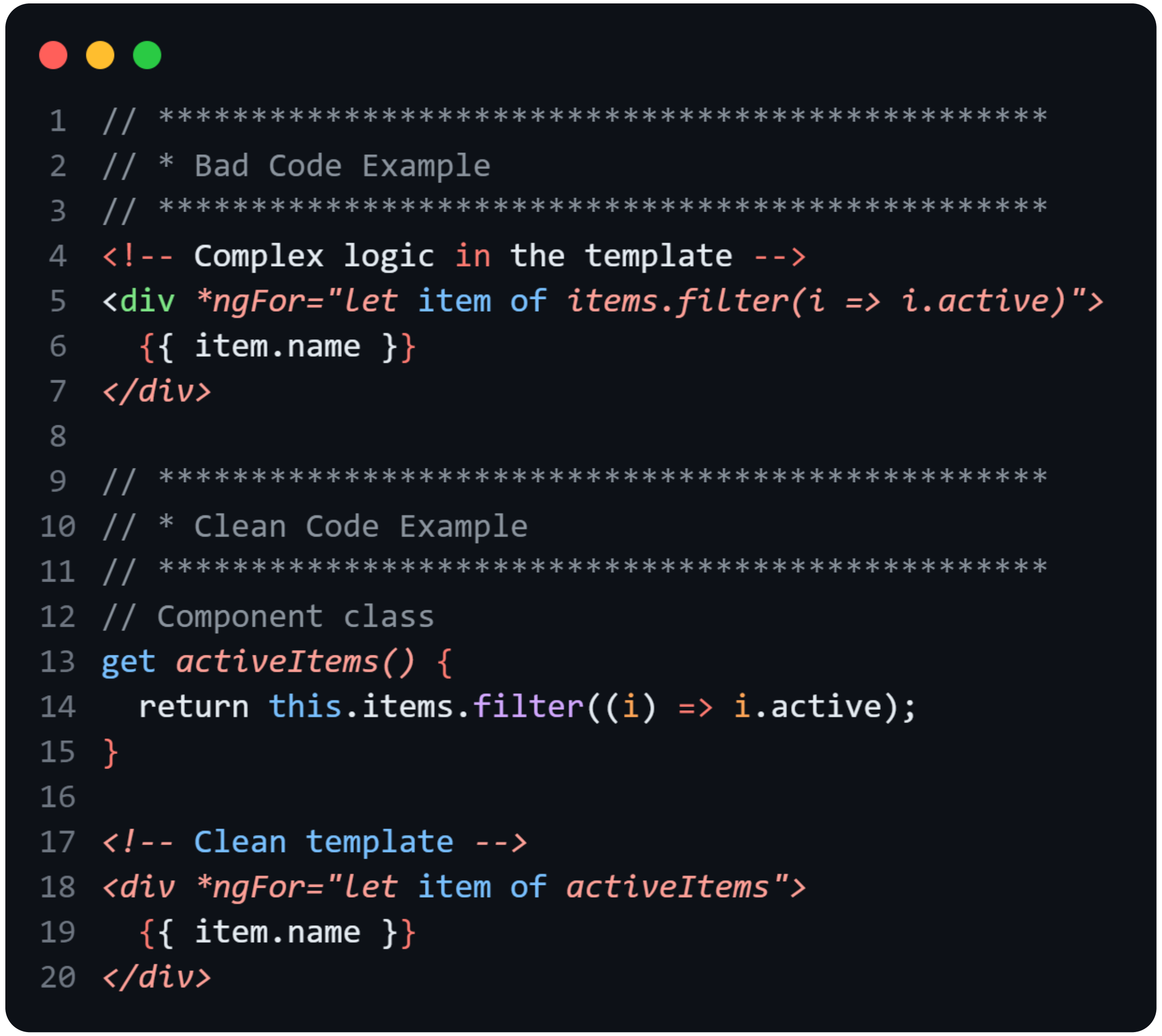
- Description
  - Leverage Angular's built-in dependency injection system for services and avoid creating instances manually.
- Explanation
  - The clean code example uses Angular’s dependency injection (constructor) to provide the service or inject method adhering to best practices for testability and modularity.



```
// ***********************************************
// * Bad Code Example
// ***********************************************
export class AppComponent {
  dataService = new DataService(); // Manual instantiation
}

// ***********************************************
// * Clean Code Example
// ***********************************************

import { DataService } from './data.service';

export class AppComponent {
  // Contructor
  constructor(private dataService: DataService) {}

  // Inject method
  dataService = inject(DataService);
}
```

Guideline 3 – Keep Templates Clean and Minimal

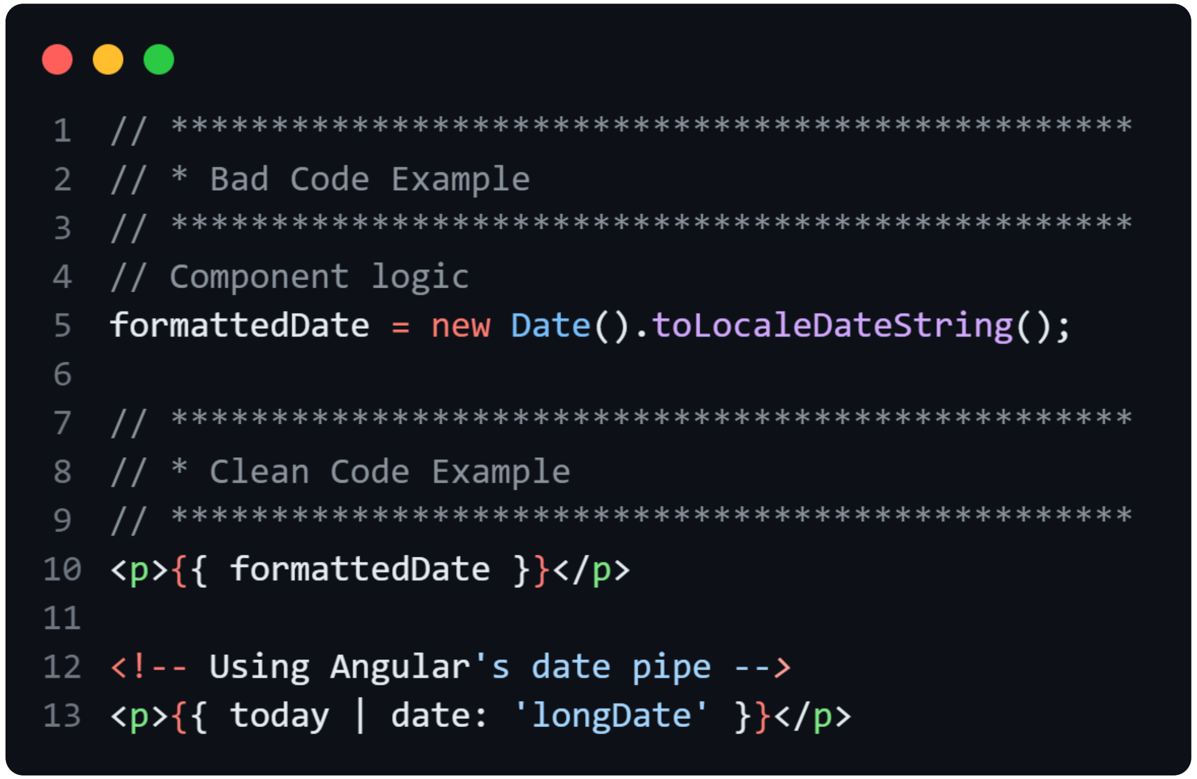
- Description
  - Avoid writing complex logic in templates. Move such logic to the component class or a service for better readability and testability.
- Explanation
  - In the clean code example, filtering logic is moved to a getter in the component class, making the template cleaner and easier to understand.



```
// ************************************************
// * Bad Code Example
// ************************************************
<!-- Complex logic in the template -->
<div *ngFor="let item of items.filter(i => i.active)">
  {{ item.name }}
</div>

// ************************************************
// * Clean Code Example
// ************************************************
// Component class
get activeItems() {
  return this.items.filter((i) => i.active);
}

<!-- Clean template -->
<div *ngFor="let item of activeItems">
  {{ item.name }}
</div>
```

Guideline 4 – Leverage Angular's Built-in Pipes

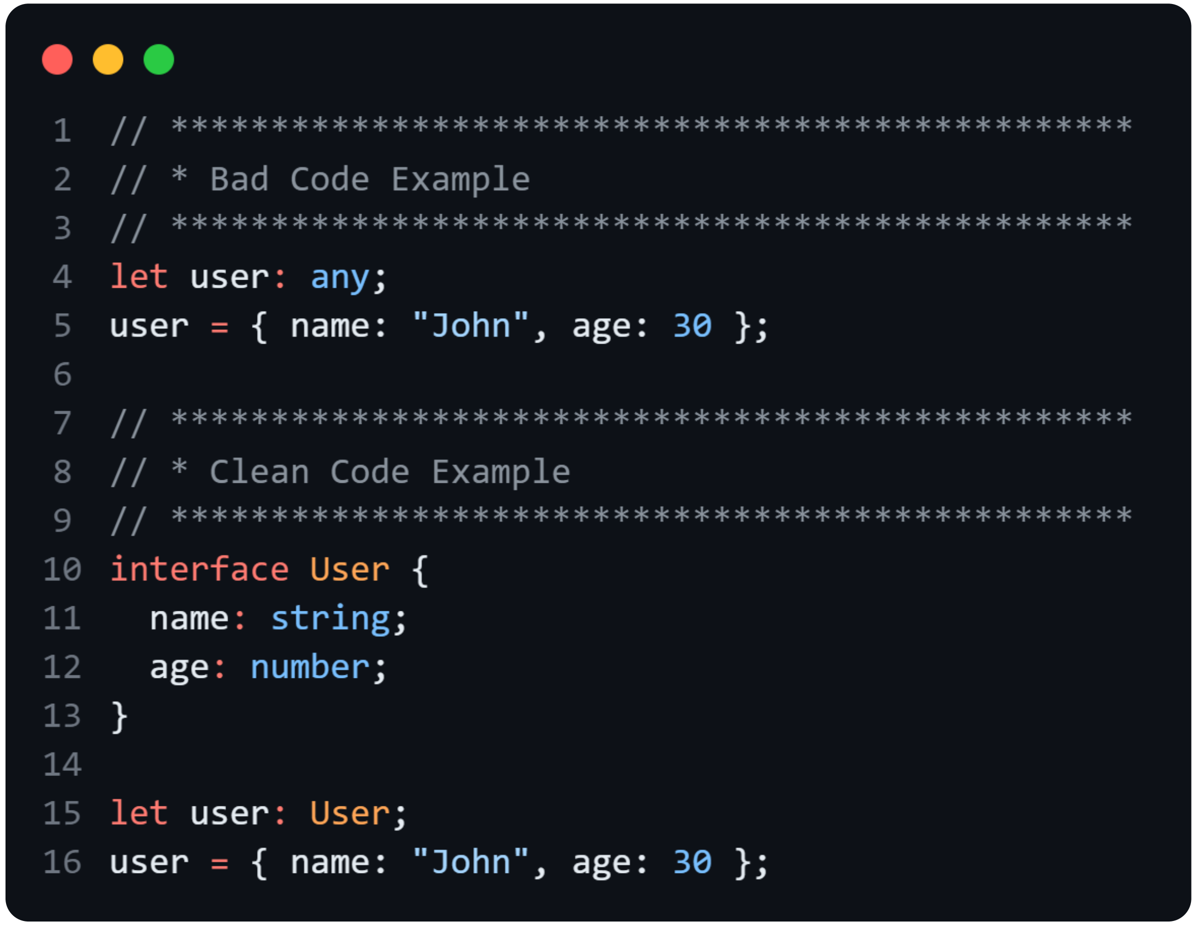
- Description
  - Use Angular's built-in pipes (e.g., date, uppercase) to format data in the template instead of handling it in the component.
- Explanation
  - The clean code example uses Angular's date pipe for formatting, reducing the complexity of the component logic.



```
// *********************************************
// * Bad Code Example
// *********************************************
// Component logic
formattedDate = new Date().toLocaleDateString();

// *********************************************
// * Clean Code Example
// *********************************************
<p>{{ formattedDate }}</p>

<!-- Using Angular's date pipe -->
<p>{{ today | date: 'longDate' }}</p>
```

Guideline 5 – Avoid Using any Type in TypeScript

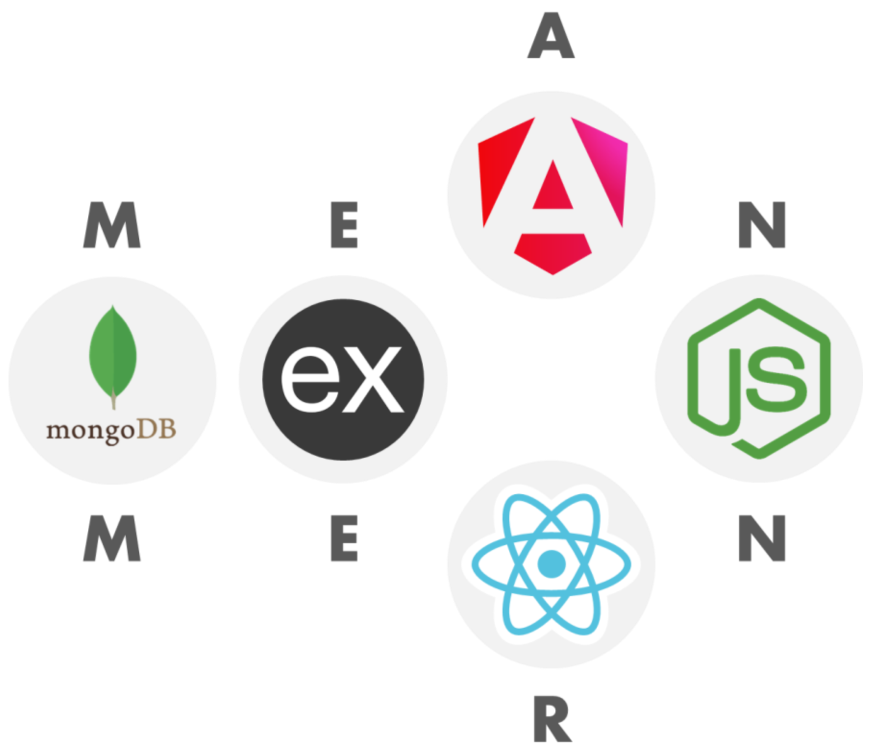
- Description
  - Avoid using the any type as it defeats the purpose of TypeScript's type checking. Use specific types or interfaces.
- Explanation
  - The clean code example defines a User interface, ensuring type safety and improving readability.



```typescript
// ***********************************************
// * Bad Code Example
// ***********************************************
let user: any;
user = { name: "John", age: 30 };

// ***********************************************
// * Clean Code Example
// ***********************************************
interface User {
  name: string;
  age: number;
}

let user: User;
user = { name: "John", age: 30 };
```

Node.js/Express.js

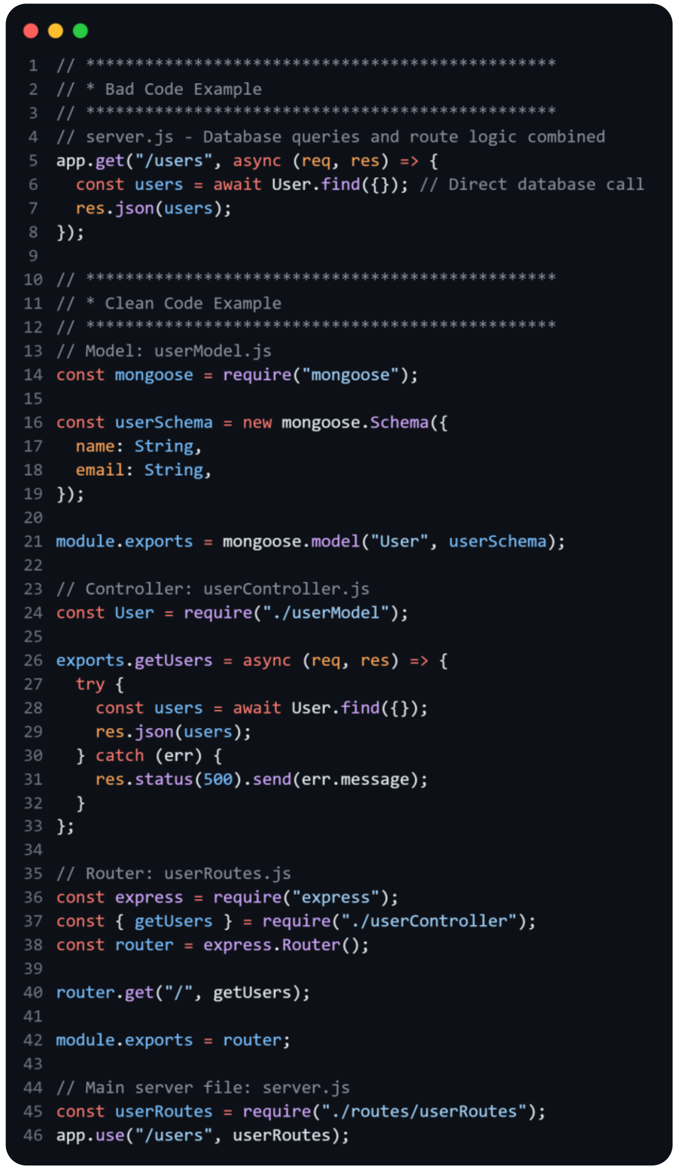
Guidelines For Project



Guideline 1 – Structure Code with MVC Pattern

- Description
  - Separate database logic into models, and keep controllers focused on business logic. Use services to further abstract database interactions if needed.

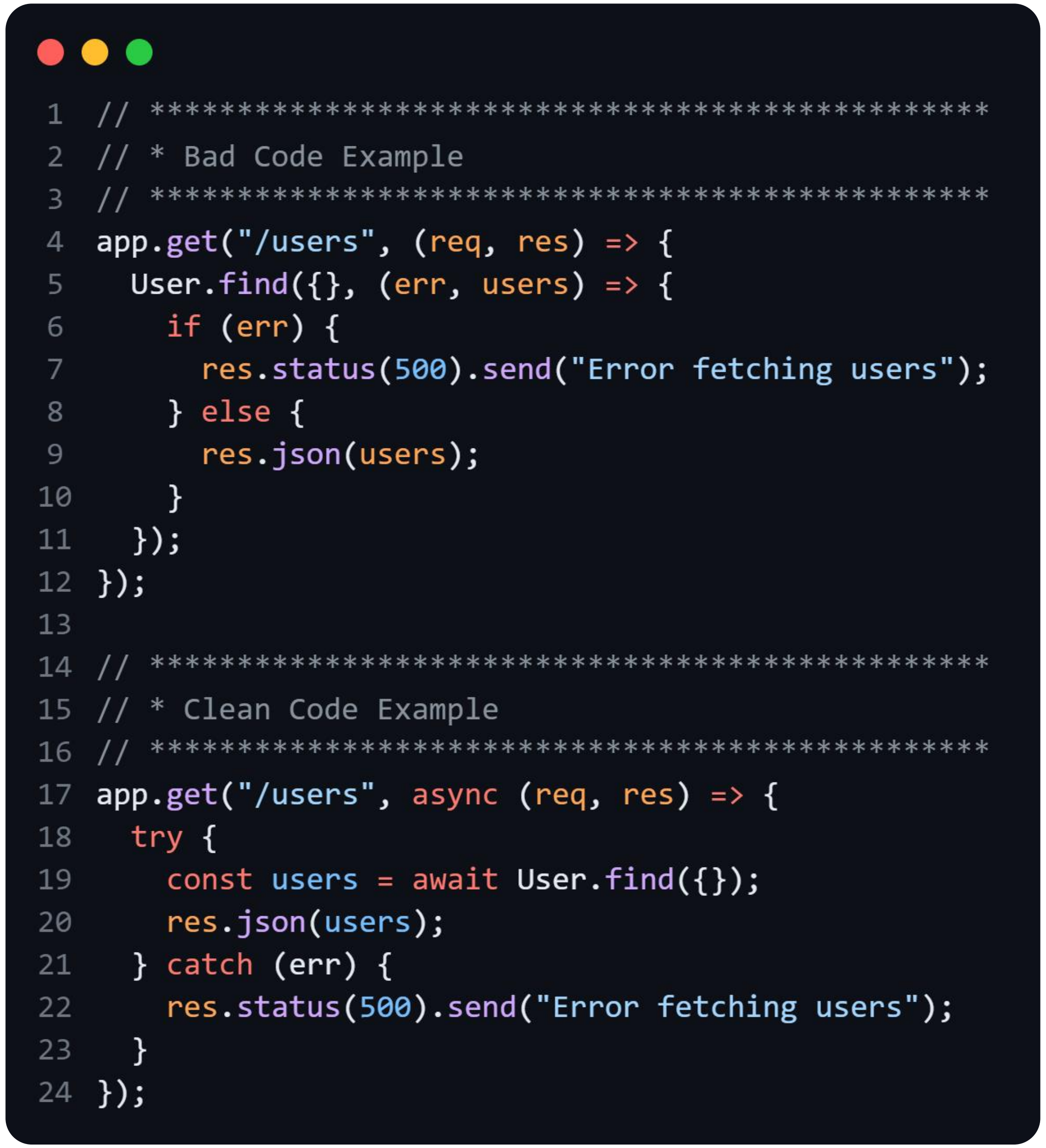
- Explanation
  - The clean code example separates the Mongoose model, the business logic in the controller, and routing logic, promoting modularity.



```javascript
// ************************************************
// * Bad Code Example
// ************************************************
// server.js - Database queries and route logic combined
app.get("/users", async (req, res) => {
  const users = await User.find({}); // Direct database call
  res.json(users);
});

// ************************************************
// * Clean Code Example
// ************************************************
// Model: userModel.js
const mongoose = require("mongoose");

const userSchema = new mongoose.Schema({
  name: String,
  email: String,
});

module.exports = mongoose.model("User", userSchema);

// Controller: userController.js
const User = require("./userModel");

exports.getUsers = async (req, res) => {
  try {
    const users = await User.find({});
    res.json(users);
  } catch (err) {
    res.status(500).send(err.message);
  }
};

// Router: userRoutes.js
const express = require("express");
const { getUsers } = require("./userController");
const router = express.Router();

router.get("/", getUsers);

module.exports = router;

// Main server file: server.js
const userRoutes = require("./routes/userRoutes");
app.use("/users", userRoutes);
```

Guideline 2 – Use Async/Await for Asynchronous Code

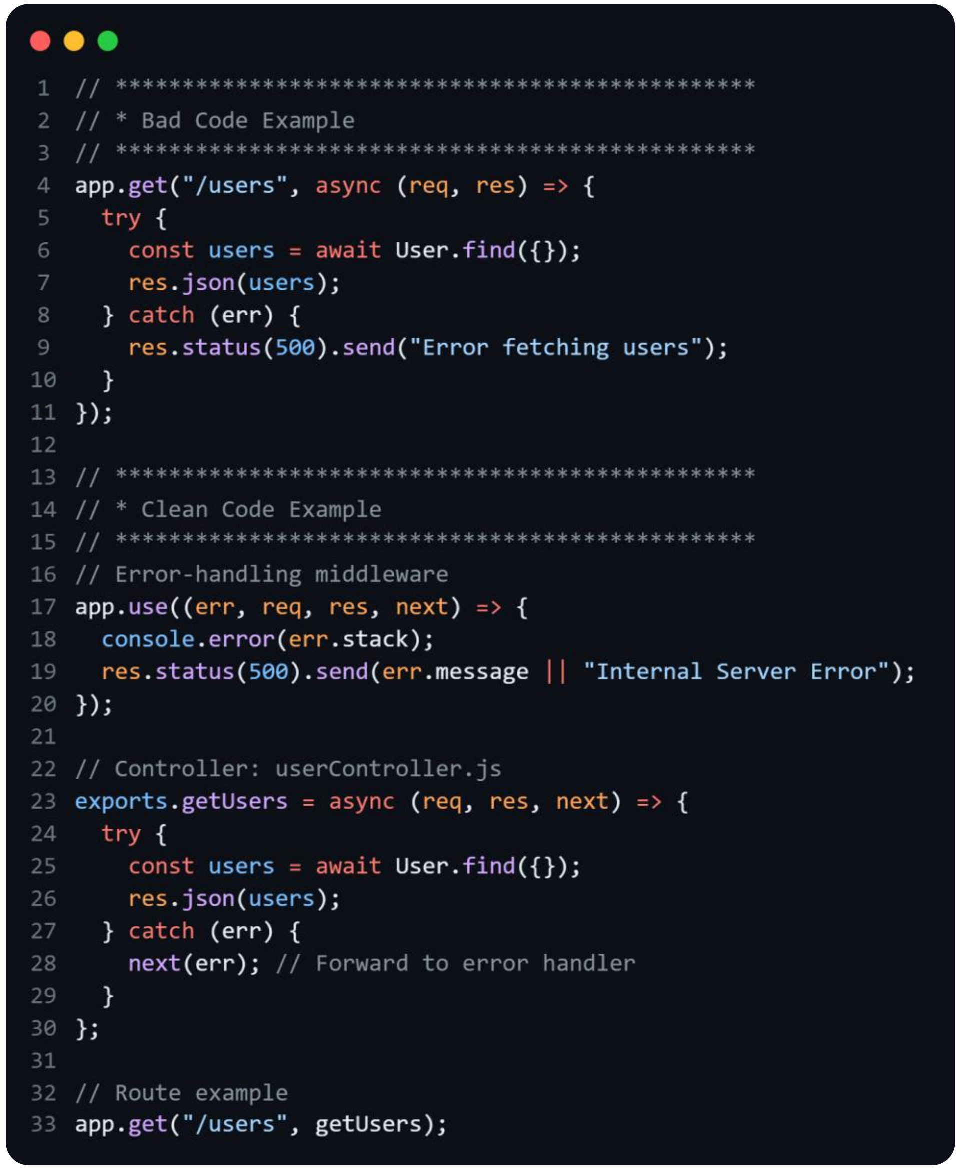
- Description
  - Handle Mongoose queries using async/await and proper error handling with try-catch.
- Explanation
  - Using async/await avoids callback-style code and makes it easier to read and manage error handling.



```javascript
// ***************************************************
// * Bad Code Example
// ***************************************************
app.get("/users", (req, res) => {
  User.find({}, (err, users) => {
    if (err) {
      res.status(500).send("Error fetching users");
    } else {
      res.json(users);
    }
  });
});

// ***************************************************
// * Clean Code Example
// ***************************************************
app.get("/users", async (req, res) => {
  try {
    const users = await User.find({});
    res.json(users);
  } catch (err) {
    res.status(500).send("Error fetching users");
  }
});
```

Guideline 3 – Centralize Error Handling

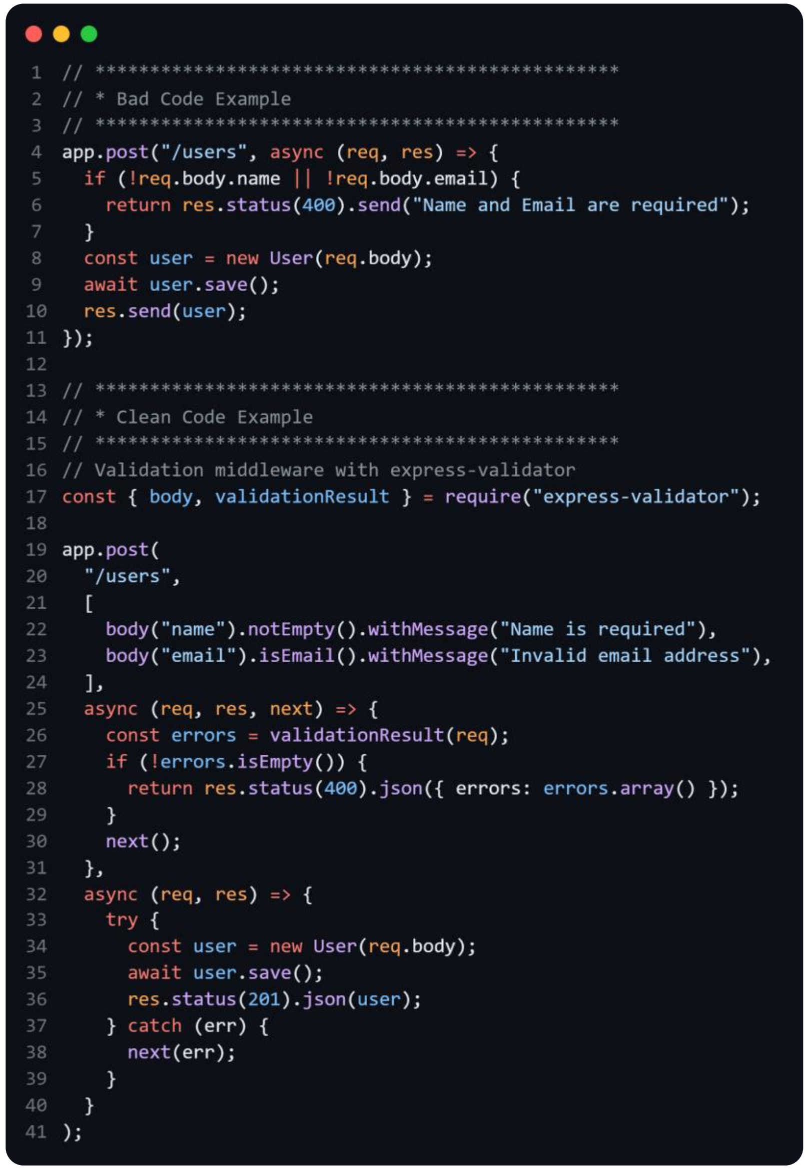
- Description
  - Handle Mongoose errors in a centralized middleware to avoid repeating error-handling logic.
- Explanation
  - Centralizing error handling ensures consistency in responses and reduces boilerplate error code.



```
// ***************************************************
// * Bad Code Example
// ***************************************************
app.get("/users", async (req, res) => {
  try {
    const users = await User.find({});
    res.json(users);
  } catch (err) {
    res.status(500).send("Error fetching users");
  }
});

// ***************************************************
// * Clean Code Example
// ***************************************************
// Error-handling middleware
app.use((err, req, res, next) => {
  console.error(err.stack);
  res.status(500).send(err.message || "Internal Server Error");
});

// Controller: userController.js
exports.getUsers = async (req, res, next) => {
  try {
    const users = await User.find({});
    res.json(users);
  } catch (err) {
    next(err); // Forward to error handler
  }
};

// Route example
app.get("/users", getUsers);
```

Guideline 4 – Validate Request Data with Middleware

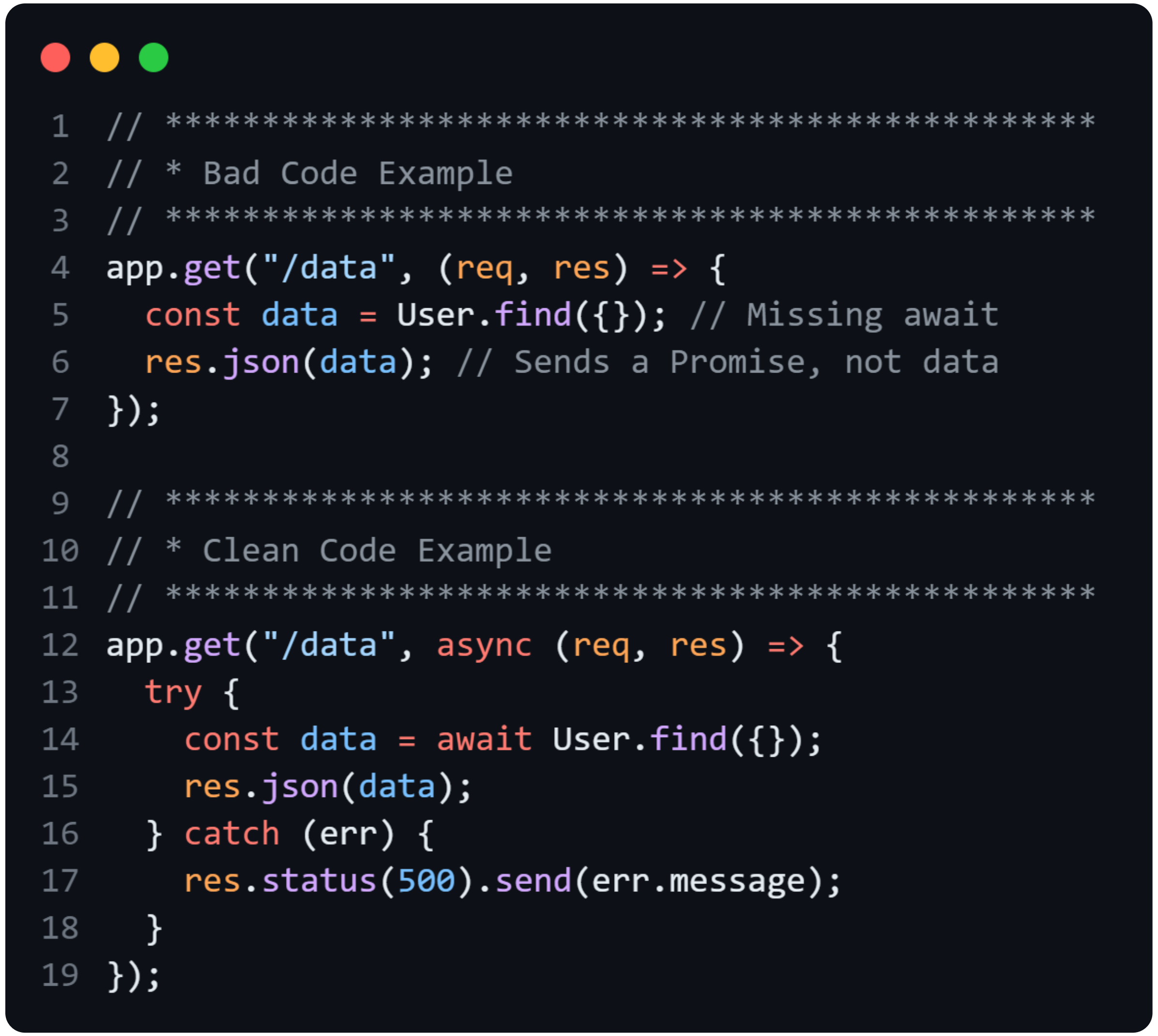
- Description
  - Validate incoming data for Mongoose models using middleware like express-validator or Mongoose validation directly.
- Explanation
  - Validating request data before it reaches the Mongoose model prevents unnecessary database operations and ensures data integrity.



```
// ************************************************
// * Bad Code Example
// ************************************************
app.post("/users", async (req, res) => {
  if (!req.body.name || !req.body.email) {
    return res.status(400).send("Name and Email are required");
  }
  const user = new User(req.body);
  await user.save();
  res.send(user);
});

// ************************************************
// * Clean Code Example
// ************************************************
// Validation middleware with express-validator
const { body, validationResult } = require("express-validator");

app.post(
  "/users",
  [
    body("name").notEmpty().withMessage("Name is required"),
    body("email").isEmail().withMessage("Invalid email address"),
  ],
  async (req, res, next) => {
    const errors = validationResult(req);
    if (!errors.isEmpty()) {
      return res.status(400).json({ errors: errors.array() });
    }
    next();
  },
  async (req, res) => {
    try {
      const user = new User(req.body);
      await user.save();
      res.status(201).json(user);
    } catch (err) {
      next(err);
    }
  }
);
```

Guideline 5 – Avoid Blocking Code in the Event Loop

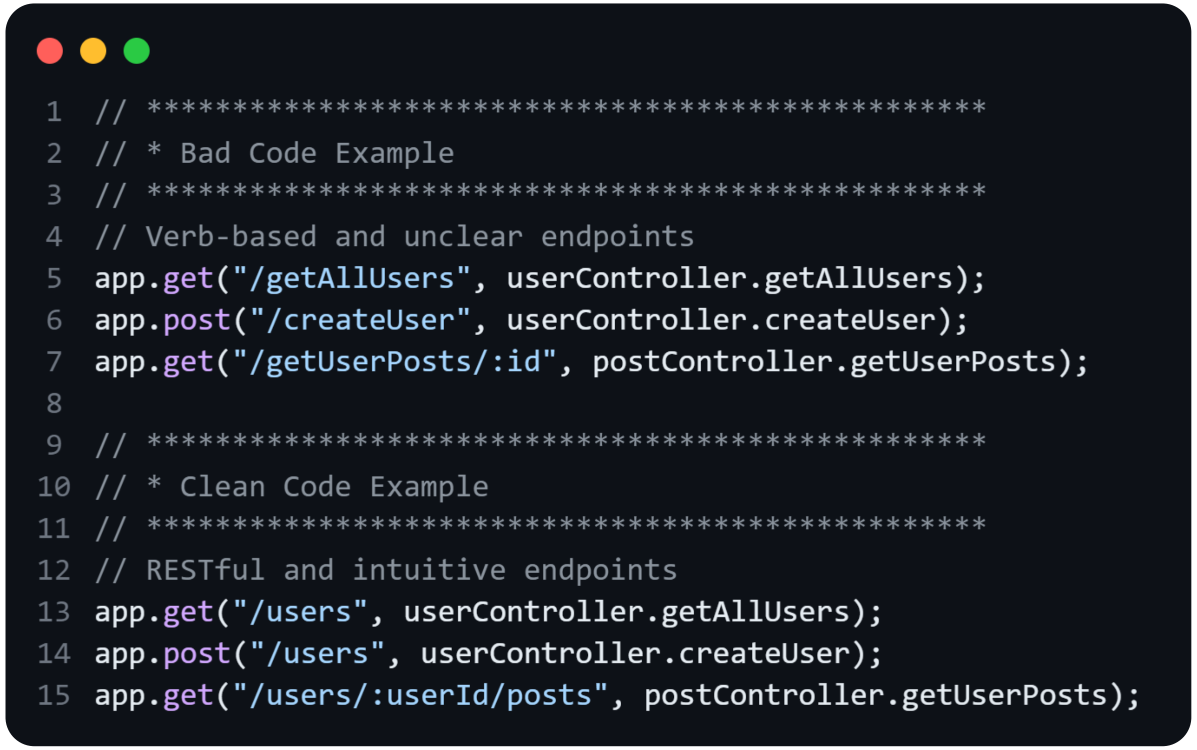
- Description
  - Handle Mongoose operations asynchronously to avoid blocking the event loop.
- Explanation
  - Always await asynchronous Mongoose operations to avoid sending unresolved Promises to the client.



```javascript
// ************************************************
// * Bad Code Example
// ************************************************
app.get("/data", (req, res) => {
  const data = User.find({}); // Missing await
  res.json(data); // Sends a Promise, not data
});

// ************************************************
// * Clean Code Example
// ************************************************
app.get("/data", async (req, res) => {
  try {
    const data = await User.find({});
    res.json(data);
  } catch (err) {
    res.status(500).send(err.message);
  }
});
```

Guideline 6 – Use Proper URL Naming

- Description
  - Follow RESTful conventions for URL structure and naming to ensure clarity and standardization. Use nouns instead of verbs, plural for resource names, and hierarchical paths for related resources.
  - Key Practices:
    - Use nouns, not verbs
      - Avoid action words in URLs like /getUser or /createUser. Instead, use resources like /users.
    - Use plural nouns for resources
      - Represent collections with plural nouns, e.g., /users instead of /user.
    - Use hierarchical structure for nested resources
      - Reflect relationships using paths, e.g., /users/:userId/posts.
    - Use query parameters for filtering, sorting, or searching
      - Don't encode these operations into the URL itself. Use /users?sort=age&limit=10 instead of /users/sortByAge.



```
// ************************************************
// * Bad Code Example
// ************************************************
// Verb-based and unclear endpoints
app.get("/getAllUsers", userController.getAllUsers);
app.post("/createUser", userController.createUser);
app.get("/getUserPosts/:id", postController.getUserPosts);

// ************************************************
// * Clean Code Example
// ************************************************
// RESTful and intuitive endpoints
app.get("/users", userController.getAllUsers);
app.post("/users", userController.createUser);
app.get("/users/:userId/posts", postController.getUserPosts);
```

Resources

- Robert C Martin Clean Code – A Handbook of Agile Software Craftsmanship [Link, last accessed November 30, 2024]
- Robet C Martin, The Clean Coder – A Code of Conduct for Professional Programmers [Link, last accessed November 30, 2024]
- React Best Practices – Tips for Writing Better React Code [Link, last accessed November 30, 2024]
- Best Practices For A Clean and Performant Angular Application [Link, last accessed November 30, 2024]
- Node.js Best Practices [Link, last accessed November 30, 2024]