## Part One: Convert admin.js to admin.ts

Step 1; Ensure the file is treated as a module, preventing global variable/function conflicts across files.

```
export {};
```

Step 2: Define a reusable TypeScript type for quiz questions.

```
interface QuizQuestion {
  question: string;
  choices: string[];
  answer: string;
}
```

Step 3: Store the editable quiz data as an array of questions.

```
let quizData: QuizQuestion[] = [];

function getElement<T extends HTMLElement>(id: string): T {
  const el = document.getElementById(id);
  if (!el) throw new Error(`Missing element: ${id}`);
  return el as T;
}
```

Step 4: Utility function gets DOM elements with type safety and a helpful error if missing.

```
function getElement<T extends HTMLElement>(id: string): T {
  const el = document.getElementById(id);
  if (!el) throw new Error(`Missing element: ${id}`);
  return el as T;
}
```

Step 5: Make main UI render function. Clear and populate the admin list.

```
function renderQuizAdmin(): void {
  const listDiv = getElement<HTMLDivElement>('quiz-admin-list');
  listDiv.innerHTML = '';
```

Step 6: Create a Bootstrap card for each quiz item.

```
  quizData.forEach((q, index) => {
    const div = document.createElement('div');
    div.className = 'border p-3 mb-3';
```

Step 7: Editable input for the quiz question text.

```
  const questionInput = document.createElement('input');
    questionInput.value = q.question;
    questionInput.className = 'form-control mb-2';
    questionInput.oninput = () => (quizData[index].question =
questionInput.value);
    div.appendChild(questionInput);
```

Step 8: Editable input for choices (comma-separated), updates internal array.

```
   const choicesInput = document.createElement('input');
    choicesInput.value = q.choices.join(', ');
    choicesInput.className = 'form-control mb-2';
    choicesInput.oninput = () => (quizData[index].choices =
choicesInput.value.split(',').map(c => c.trim()));
    div.appendChild(choicesInput);
```

Step 9: Editable input for the correct answer.

```
    const answerInput = document.createElement('input');
    answerInput.value = q.answer;
    answerInput.className = 'form-control mb-2';
    answerInput.oninput = () => (quizData[index].answer =
answerInput.value.trim());
    div.appendChild(answerInput);
```

Step 10: Delete a quiz item and re-render the list.

```
    const delBtn = document.createElement('button');
    delBtn.className = 'btn btn-danger btn-sm';
    delBtn.textContent = 'Delete';
    delBtn.onclick = () => {
      quizData.splice(index, 1);
      renderQuizAdmin();
    };
    div.appendChild(delBtn);
```

Step 11: Adds the question block to the UI.

```
    listDiv.appendChild(div);
  });
```

Step 12: Add a save button that POSTs all updated quiz data to the backend.

```
  const saveBtn = document.createElement('button');
  saveBtn.textContent = 'Save All Changes';
  saveBtn.className = 'btn btn-primary mt-3';
  saveBtn.onclick = () => {
    fetch('/api/quiz', {
      method: 'POST',
      headers: { 'Content-Type': 'application/json' },
      body: JSON.stringify(quizData)
    })
      .then(res => res.json())
      .then(data => alert(data.message || 'Saved!'))
      .catch(err => alert('Failed to save changes.'));
  };
  listDiv.appendChild(saveBtn);
}
```

Adding a New Question

Step 13: Handle form submission for new questions.

```
document.getElementById('add-question-
form')?.addEventListener('submit', (e) => {
  e.preventDefault();
```

Step 14: Get new question data from form fields.

```
  const question = (document.getElementById('new-question') as
HTMLInputElement)?.value.trim();
  const choicesInput = (document.getElementById('new-choices') as
HTMLInputElement)?.value.trim();
  const answer = (document.getElementById('new-answer') as
HTMLInputElement)?.value.trim();
```

Step 15: Validate input and parses choices.

```
  if (!question || !choicesInput || !answer) return;
  const choices = choicesInput.split(',').map(c =>
c.trim()).filter(Boolean);
```

Step 16: Ensure valid number of answer choices.

```
  if (choices.length < 2) {
    alert('Please provide at least 2 choices');
    return;
  }
```

Step 17: Add question and refreshes admin view.

```
  quizData.push({ question, choices, answer });
  renderQuizAdmin();
```

Step 18: Clear form fields after adding a question.

```
  (document.getElementById('new-question') as HTMLInputElement).value
= '';
  (document.getElementById('new-choices') as HTMLInputElement).value =
'';
  (document.getElementById('new-answer') as HTMLInputElement).value =
'';
});
```

Step 19: Initial load - oads quiz data from the backend API and renders the admin panel on page load.

```
fetch('/api/quiz')
  .then(res => res.json())
  .then((data: QuizQuestion[]) => {
    quizData = data;
    renderQuizAdmin();
  });
```

**Part two: Convert filter.js to filter.ts**

Let’s assume you start with something like this:

```
const input = document.getElementById('search');
const rows = document.querySelectorAll('#infoTable tbody tr');

input.addEventListener('input', () => {
  const value = input.value.toLowerCase();

  rows.forEach(row => {
    const text = row.textContent.toLowerCase();
    row.style.display = text.includes(value) ? '' : 'none';
  });
});
```

We will convert it to TypeScript (filter.ts)

Step 1: Import libraries (replace code) and type the rows variable

```
import { fromEvent } from 'rxjs';
import { debounceTime, map, distinctUntilChanged } from 'rxjs/
operators';

function isInput(el: Element | null): el is HTMLInputElement {
  return el instanceof HTMLInputElement;
}

const rows: NodeListOf<HTMLTableRowElement> =
document.querySelectorAll('#infoTable tbody tr');
```

Step 2: Add a type guard for the input

```
const input = document.getElementById('search');
function isInput(el: Element | null): el is HTMLInputElement {
  return el instanceof HTMLInputElement;
}
```

Step 3: Use the type guard to protect .value

```
if (isInput(input)) {
  fromEvent(input, 'input').pipe(
    map((e: Event) => (e.target as
HTMLInputElement).value.toLowerCase()),
    debounceTime(300),
    distinctUntilChanged()
  ).subscribe((value: string) => {
    rows.forEach(row => {
      const text = row.textContent?.toLowerCase() || '';
      row.style.display = text.includes(value) ? '' : 'none';
    });
  });
} else {
  console.error('Search input not found or is not an input element.');
}
```

Rename the file

**Part three: Convert contact.js to contact.ts**

Step 1: Add types to imports. Add at the top (replace code):

```
import { fromEvent } from 'rxjs';
import { debounceTime, map, tap } from 'rxjs/operators';
```

Step 2. Add Type Guards (New in TS). Add these helper functions before DOMContentLoaded:

```
function isForm(el: Element | null): el is HTMLFormElement {
  return el instanceof HTMLFormElement;
}

function isInput(el: Element | null): el is HTMLInputElement {
  return el instanceof HTMLInputElement;
}
```

Step 3. Add a utility to get input values (with type safety). Replace any direct .value access with this helper:

```
function getInputValue(id: string): string {
  const el = document.getElementById(id);
  if (!isInput(el)) throw new Error(`Missing or invalid input element: 
${id}`);
  return el.value.trim();
}
```

Step 4. Add runtime type check for form and responseDiv. Change this:

```
const form = document.getElementById('myForm');
const responseDiv = document.getElementById('response');
```

To this:

```
const form = document.getElementById('myForm');
const responseDiv = document.getElementById('response');

if (!isForm(form) || !responseDiv) {
  console.error('Form or response div not found!');
  return;
}
```

Step 5. Replace .value.trim() direct access with getInputValue. Change this:

```
const phone = document.getElementById('phone').value.trim();
```

To this:

```
const phone = getInputValue('phone');
```

Do the same for email and zip.

Step 6. Type your errors array. Change this:

```
const errors = [];
```

To this:

```
const errors: string[] = [];
```

Step 7. Type the subscription result. Change this:

```
.subscribe(({ phone, email, zip, errors }) => {
```

To this:

```
.subscribe(async ({ phone, email, zip, errors }: { phone: string; 
email: string; zip: string; errors: string[] }) => {
```

**Part four: convert quizquestions.js to quizquestions.ts**

Step 1. Add an Interface for the Quiz Questions. At the top of your file:

```
interface QuizQuestion {
  question: string;
  choices: string[];
  answer: string;
}
```

This defines the shape of each quiz item, so TypeScript can give you autocomplete, catch typos, and prevent misuse.

Step 2. Type the quizData Array

Change this:

```
let quizData = [];
```

To this:

```
let quizData: QuizQuestion[] = [];
```

Step 3. Add Type to the Fetch Response

Change this:

```
fetch('/api/quiz')
  .then(res => res.json())
  .then(data => {
    quizData = data;
    renderQuiz(data);
  });
```

To this:

```
fetch('/api/quiz')
  .then(res => res.json())
  .then((data: QuizQuestion[]) => {
    quizData = data;
    renderQuiz(data);
  });
```

This ensures data is correctly typed as an array of QuizQuestion.

Step 4. Add Parameter Type to renderQuiz()

Change this:

```
function renderQuiz(data) {
```

To this:

```
function renderQuiz(data: QuizQuestion[]): void {
```

Explicit parameter and return types help with readability and prevent accidental misuse.

Step 5. Type All DOM Elements in renderQuiz

If you use:

```
const container = document.getElementById('quiz');
```

Change to:

```
const container = document.getElementById('quiz') as HTMLDivElement;
```

Use as HTMLInputElement, as HTMLDivElement, etc. to help TypeScript understand what each element is. This avoids errors when you access properties like .innerHTML, .textContent, or .appendChild().

Step 6. Use Optional Chaining Safely

Anywhere you see this:

```
const text = row.textContent.toLowerCase();
```

Change to:

```
const text = row.textContent?.toLowerCase() || '';
```

TypeScript will warn if textContent could be null, so optional chaining ?. avoids crashes.

Part five: Add tsconfig.json

Put ts files into a ts-src folder at root and also at root create a tsconfig.json file. To convert ts to js, use tsc (install TypeScript first):

```
{
  "compilerOptions": {
    "target": "ES6",
    "module": "ES6",
    "outDir": "./public/js",
    "rootDir": "./ts-src",
    "strict": true,
    "esModuleInterop": true,
    "moduleResolution": "node",
    "forceConsistentCasingInFileNames": true,
    "skipLibCheck": true,
    "lib": ["DOM", "ES6"]
  },
  "include": ["ts-src/**/*"]
}
```

**Explanation:**
`"compilerOptions": Core Settings for TypeScript Compiler`
`"target": "ES6"`

Tells TypeScript which JavaScript version to output

`ES6 = modern JavaScript (supports let, const, arrow functions, etc.)`

Needed for browser compatibility and modern features

`"module": "ES6"`

Sets the module system (how you use import / export). ”ES6" means import { x } from './x.js' is allowed. Use this for browser-compatible modules

`"outDir": "./public/js"`

Where compiled .js files go. After tsc, your .ts files from ts-src/ will become .js files here

This is the folder you link to in your HTML:

```
<script type="module" src="js/contact.js"></script>
```

`"rootDir": "./ts-src"`

Where your source .ts files live

This keeps your project organized: TypeScript only compiles what's in `ts-src/`

`"strict": true`

Turns on all strict type checks

Catches common bugs at compile time (e.g., using a variable before defining it)

Highly recommended for reliable code

`"esModuleInterop": true`

Allows you to use import x from 'package' even if the package uses CommonJS

Needed for importing many libraries without weird default import errors

`"moduleResolution": "node"`

Tells TypeScript how to look up modules (e.g., inside node_modules)

Follows the same rules as Node.js

Helps you use npm packages and TypeScript definitions easily

`"forceConsistentCasingInFileNames": true`

Ensures that MyFile.ts and myfile.ts are treated as different even on case-insensitive systems like macOS

Prevents cross-platform bugs when deploying to Linux servers

Recommended

`"skipLibCheck": true`

Skips type-checking for node_modules

Speeds up compilation and avoids unnecessary warnings from 3rd-party libs

`"lib": ["DOM", "ES6"]`

Tells TypeScript what built-in APIs to expect

"DOM" = TypeScript knows about document, window, HTMLElement, etc.

"ES6" = Includes built-in features like Map, Set, Promise, etc.

Required for browser-based apps

`"include": ["ts-src/**/*"]`

Tells TypeScript: only compile files in ts-src/ folder

**/* means include all subfolders and .ts files