

PERSONAL MULTI-TECHNOLOGY PROJECTS

Romaji to Kana Converter: Angular & Express

A real, working romaji-to-kana converter — a full gojuon/dakuten/yoon mapping table, real handling for long vowels, sokuon, and particle exceptions, a shared-service Angular architecture, a real Express API with a debounced, race-safe, error-resilient conversion pipeline, and a real deployment path-mounted behind the user's own existing Astro site, closing with a live link into the site's own Japanese-language lessons and kanji pages.

8 Chapters

Philip Osztromok

Generated with Claude

CHAPTER 1: PROJECT OVERVIEW & ANGULAR + EXPRESS SETUP

Romaji to Kana Converter: Angular & Express

Chapter 1 · Project Overview & Angular + Express Setup

This is one of three courses building the exact same tool in three genuinely different architectures — a web app that takes romaji (Latin-alphabet-transliterated Japanese) text and converts it into hiragana and katakana. Romaji to Kana Converter: Astro and Romaji to Kana Converter: React & Next.js remain outlined; this course's own answer gives Angular its one real home across this site's entire Personal Multi-Technology Projects set, paired with a small Express API it doesn't otherwise have anywhere here.

What the App Actually Does

- **Convert romaji text** — typed Latin-alphabet Japanese, like `konnichiwa` — into both hiragana (こんにちは) and katakana (コンニチワ).
- **Handle real Japanese-specific edge cases** correctly — long vowels, the small つ (sokuon, doubled consonants), and particle exceptions (は pronounced "wa," へ pronounced "e") — covered in real depth in Chapter 3.
- **Ties directly into content already on this site** — the existing Japanese-language lessons and the kanji tiles/reference-materials pages, both real, immediate practical uses for a working romaji-to-kana tool once it's live.

Full kanji conversion is explicitly out of scope for this course — a genuinely bigger, real dictionary-and-context-dependent problem (the same word in kana can map to several different kanji depending on meaning), deliberately deferred as a real future project of its own rather than something this course pretends to solve as a side effect of romaji conversion.

Why This Variant Commits to a Real Express Conversion Engine

The actual romaji-to-kana algorithm — a real character-mapping table plus the edge-case handling Chapter 3 covers — could genuinely run entirely inside the browser, with no backend involved at all. Both sibling courses in this trio treat that as a real, open question worth exploring directly:

COURSE	HOW THE CLIENT-VS-SERVER QUESTION IS TREATED
Romaji to Kana Converter: Astro	An open comparison — that course's own Chapter 5 is literally titled "Client-Side vs. Server-Side Conversion: Where Should the Logic Actually Run?"
Romaji to Kana Converter: React & Next.js	Also open — that course's own Chapter 4 compares "An API Route for Conversion vs. Doing It Client-Side"
Romaji to Kana Converter: Angular & Express (this course)	Decided from the outset — the conversion engine lives on a real, dedicated Express API from Chapter 5 onward, since giving Express a genuine backend role here is this course's own deliberate reason for existing

This isn't a claim that server-side conversion is objectively better — the sibling courses' own honest, comparative chapters are exactly where that real tradeoff gets explored. This course's own value is different: showing what a real Angular frontend calling a real, purpose-built Express API actually looks like, start to finish, since neither half of that pairing exists anywhere else in this project set.

Setting Up the Angular Frontend

This course targets current Angular (v22 at the time of writing) — standalone components, no NgModules, verified directly against Angular's own current documentation:

```
# Frontend
npm install -g @angular/cli
ng new romaji-converter --style=css
cd romaji-converter
```

Routing and server-side rendering are both left off deliberately — neither flag is passed, and current Angular CLI defaults to skipping both unless explicitly requested. A single-page conversion tool genuinely has no second real route to navigate to, matching the same "no router needed" finding this site's own Personal Catalogue courses already reached for a similarly simple app.

VERIFIED AGAINST ANGULAR'S OWN CURRENT DOCUMENTATION: FILE NAMES CHANGED

A freshly generated project's root component is no longer named `app.component.ts`. Current Angular's default "2025" file-naming style generates the shorter `app.ts`, with its template and styles

in separate `app.html` and `app.css` files — a genuinely surprising change for anyone who learned Angular under the older `.component.ts` convention, confirmed directly against Angular's own current documentation rather than assumed from memory.

Every component generated in this project is standalone by default — current Angular removed the need to write `standalone: true` explicitly, since that's now the only option unless a component opts out with `standalone: false`. The app bootstraps with no root module at all:

```
// src/main.ts
import { bootstrapApplication } from '@angular/platform-browser';
import { appConfig } from './app/app.config';
import { AppComponent } from './app/app';

bootstrapApplication(AppComponent, appConfig).catch((err) => console.error(err));
```

Wiring Up HttpClient

Every other framework used across this project's own site so far has called the backend with a plain `fetch()`. Angular ships its own, dependency-injected HTTP client instead — real, idiomatic Angular, configured once at the application root:

```
// src/app/app.config.ts
import { ApplicationConfig } from '@angular/core';
import { provideHttpClient } from '@angular/common/http';

export const appConfig: ApplicationConfig = {
  providers: [provideHttpClient()],
};
```

A REAL, VERIFIED ASIDE ON PROVIDEHTTPCLIENT()

Current Angular's own documentation notes that `HttpClient` is available for injection by default from Angular v21 onward, with no explicit provider required at all. `provideHttpClient()` is shown here anyway, since it's the explicit, always-correct setup regardless of exactly which version this ends

up running against, and it's the natural place to add real features later — request interceptors, caching — via its own optional configuration arguments.

Setting Up the Express Backend

A sibling folder, exactly the same real pattern this site's own React-based courses already established for pairing a frontend with a small Node API:

```
# Backend, run from a sibling folder to romaji-converter
mkdir romaji-converter-api && cd romaji-converter-api
npm init -y
npm install express cors dotenv
npm install --save-dev nodemon
```

```
# .env
PORT=4000
```

A minimal server confirms the backend is alive before any real conversion logic exists — Chapter 5 replaces this placeholder route with the real thing:

```
// server.js
require('dotenv').config();
const express = require('express');
const cors = require('cors');

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

app.get('/api/ping', (req, res) => {
  res.json({ status: 'ok', message: 'Express backend is reachable' });
});

const PORT = process.env.PORT || 4000;
app.listen(PORT, () => console.log(`Server running on port ${PORT}`));
```

CORS IS NEEDED HERE TOO

Angular's own dev server runs on port 4200 by default; Express runs on 4000. Two different origins, exactly the same real reason this site's own React-based courses needed `cors()` from their first chapter onward.

Confirming the Two Halves Talk to Each Other

A small, real service — Angular's own idiomatic pattern for anything that talks to a backend — calls the placeholder endpoint using the function-based `inject()` API rather than a constructor parameter:

```
// src/app/convert.service.ts
import { Injectable, inject } from '@angular/core';
import { HttpClient } from '@angular/common/http';

@Injectable({ providedIn: 'root' })
export class ConvertService {
  private http = inject(HttpClient);

  ping() {
    return this.http.get('http://localhost:4000/api/ping');
  }
}
```

```
// src/app/app.ts
import { Component, OnInit, inject } from '@angular/core';
import { ConvertService } from './convert.service';

@Component({
  selector: 'app-root',
  templateUrl: './app.html',
  styleUrls: ['./app.css'],
})
export class AppComponent implements OnInit {
  private convertService = inject(ConvertService);
  pingResult = '';
}
```

```

ngOnInit() {
  this.convertService.ping().subscribe((res: any) => {
    this.pingResult = JSON.stringify(res);
  });
}
}

```

```

<!-- src/app/app.html -->
<h1>Romaji to Kana Converter</h1>
<p>Backend check: {{ pingResult }}</p>

```

Running `npx nodemon server.js` in one terminal and `ng serve` in the other, then visiting Angular's own dev server (typically `http://localhost:4200`), should show the real JSON response from Express rendered directly on the page — confirmation both halves of this stack are genuinely talking to each other before a single real conversion rule exists.

Where This Course Is Headed

A real romaji-to-kana mapping table (Chapter 2); the genuinely tricky edge cases — long vowels, sokuon, and particle exceptions (Chapter 3); structuring the tool properly across Angular services and components (Chapter 4); the real Express API this course exists to build, replacing today's placeholder route (Chapter 5); the actual input/output UI (Chapter 6); deployment (Chapter 7); and a capstone integrating this tool into the existing Astro-based site, right alongside the Japanese-language lessons and kanji pages it's meant to support (Chapter 8).

Hands-On Exercises

EXERCISE 1

Scaffold both the Angular frontend and the Express backend exactly as this chapter describes, wire up `ConvertService` and `app.ts`'s `ngOnInit` to call the real `/api/ping` endpoint, and confirm the real JSON response renders on screen.

 [View solution](#)

EXERCISE 2

Run `ng new` in a separate scratch folder with no flags at all, and inspect the real generated file structure — confirm there's no routing module and no SSR-related files, and confirm the root component's own real file names match this chapter's own finding-box.

 [View solution](#)

EXERCISE 3

In your own words, explain why this specific course commits up front to a real Express-hosted conversion engine, while its two sibling courses treat the identical client-vs-server question as something to explore and compare within their own chapters instead.

 [View solution](#)

CHAPTER 1 QUICK REFERENCE

- **The shared app** — romaji-to-hiragana/katakana conversion, kanji conversion deliberately deferred
- **Why this variant** — gives Angular its one real home in this project set, paired with a small, purpose-built Express API
- **Decided, not explored** — unlike its Astro and Next.js siblings, this course commits to server-side conversion from Chapter 1 onward
- **Current Angular (v22)** — standalone by default, no NgModules, no routing/SSR unless explicitly requested
- **2025 file-naming style** — `app.ts/app.html/app.css`, not the older `app.component.ts` convention
- **HttpClient** — Angular's own dependency-injected HTTP client, configured via `provideHttpClient()`, used in place of a raw `fetch()`
- **Next chapter:** The Conversion Engine — a real romaji-to-kana mapping table

CHAPTER 2: THE CONVERSION ENGINE

Romaji to Kana Converter: Angular & Express

Chapter 2 · The Conversion Engine

Chapter 1 decided the conversion algorithm will eventually live on a real Express API. This chapter deliberately doesn't wire anything up to Angular or Express yet — it builds the engine itself as a plain, framework-agnostic module, tested against real input in isolation first. Chapter 4 decides how Angular's own side organizes around it; Chapter 5 moves the canonical copy into Express. Getting the actual algorithm correct comes first.

The Gojūon Table: Japan's Real 5×10 Sound Grid

Japanese kana are built around a real, systematic structure — five vowels (a, i, u, e, o) crossed with nine consonant rows, plus one standalone sound (ん/ン, "n"). A handful of cells are irregular under Hepburn romanization — し is "shi" not "si," ち is "chi" not "ti," つ is "tsu" not "tu," ふ is "fu" not "hu" — genuine historical exceptions baked into the standard system itself, not something this course invented:

```
// conversion-engine.ts – the base gojūon grid
export const KANA_MAP: Record<string, { hiragana: string; katakana: string }> = {
  // Vowels
  a: { hiragana: 'あ', katakana: 'ア' },
  i: { hiragana: 'い', katakana: 'イ' },
  u: { hiragana: 'う', katakana: 'ウ' },
  e: { hiragana: 'え', katakana: 'エ' },
  o: { hiragana: 'お', katakana: 'オ' },

  // K row
  ka: { hiragana: 'か', katakana: 'カ' }, ki: { hiragana: 'き', katakana: 'キ' },
  ku: { hiragana: 'く', katakana: 'ク' }, ke: { hiragana: 'け', katakana: 'ケ' },
  ko: { hiragana: 'こ', katakana: 'コ' },

  // S row – shi is the real Hepburn exception, not "si"
  sa: { hiragana: 'さ', katakana: 'サ' }, shi: { hiragana: 'し', katakana: 'シ' },
  su: { hiragana: 'す', katakana: 'ス' }, se: { hiragana: 'せ', katakana: 'セ' },
  so: { hiragana: 'そ', katakana: 'ソ' },
```

```

// T row – chi and tsu are the real exceptions
ta: { hiragana: 'た', katakana: 'タ' }, chi: { hiragana: 'ち', katakana: 'チ' },
tsu: { hiragana: 'つ', katakana: 'ツ' }, te: { hiragana: 'て', katakana: 'テ' },
to: { hiragana: 'と', katakana: 'ト' },

// N row
na: { hiragana: 'な', katakana: 'ナ' }, ni: { hiragana: 'に', katakana: 'ニ' },
nu: { hiragana: 'ぬ', katakana: 'ヌ' }, ne: { hiragana: 'ね', katakana: 'ネ' },
no: { hiragana: 'の', katakana: 'ノ' },

// H row – fu is the real exception, not "hu"
ha: { hiragana: 'は', katakana: 'ハ' }, hi: { hiragana: 'ひ', katakana: 'ヒ' },
fu: { hiragana: 'ふ', katakana: 'フ' }, he: { hiragana: 'へ', katakana: 'ヘ' },
ho: { hiragana: 'ほ', katakana: 'ホ' },

// M row
ma: { hiragana: 'ま', katakana: 'マ' }, mi: { hiragana: 'み', katakana: 'ミ' },
mu: { hiragana: 'む', katakana: 'ム' }, me: { hiragana: 'め', katakana: 'メ' },
mo: { hiragana: 'も', katakana: 'モ' },

// Y row – only three real cells exist (ya/yu/yo)
ya: { hiragana: 'や', katakana: 'ヤ' }, yu: { hiragana: 'ゆ', katakana: 'ユ' },
yo: { hiragana: 'よ', katakana: 'ヨ' },

// R row
ra: { hiragana: 'ら', katakana: 'ラ' }, ri: { hiragana: 'り', katakana: 'リ' },
ru: { hiragana: 'る', katakana: 'ル' }, re: { hiragana: 'れ', katakana: 'レ' },
ro: { hiragana: 'ろ', katakana: 'ロ' },

// W row – only two real cells; "wo" is the object-particle character を
wa: { hiragana: 'わ', katakana: 'ワ' }, wo: { hiragana: 'を', katakana: 'ヲ' },

// The standalone syllabic n – its own real, single-character sound
n: { hiragana: 'ん', katakana: 'ン' },
};

```

Voiced and Semi-Voiced Sounds

A dakuten mark (゛) or handakuten mark (゜) shifts a base consonant into a real, distinct sound — G/Z/D/B rows, and a P row from H. There's a real, genuine ambiguity worth naming honestly rather than hiding: standard Hepburn romanizes both じ and ぢ as "ji", and both ず and づ as "zu" — a flat romaji-keyed table simply cannot tell these apart. This table maps "ji"/"zu" to their far more common Z-row readings (じ/ず); the rarer ぢ/づ, mostly confined to specific compound-word contexts, aren't reachable through plain romaji input at all.

```
// conversion-engine.ts — added to the same KANA_MAP object

// G row
ga: { hiragana: 'が', katakana: 'ガ' }, gi: { hiragana: 'ぎ', katakana: 'ギ' },
gu: { hiragana: 'ぐ', katakana: 'グ' }, ge: { hiragana: 'げ', katakana: 'ゲ' },
go: { hiragana: 'ご', katakana: 'ゴ' },

// Z row — "ji"/"zu" deliberately resolve here, per the note above
za: { hiragana: 'ざ', katakana: 'ザ' }, ji: { hiragana: 'じ', katakana: 'ジ' },
zu: { hiragana: 'ず', katakana: 'ズ' }, ze: { hiragana: 'ぜ', katakana: 'ゼ' },
zo: { hiragana: 'ぞ', katakana: 'ゾ' },

// D row — only de/do have their own unambiguous romaji
da: { hiragana: 'だ', katakana: 'タ' }, de: { hiragana: 'で', katakana: 'デ' },
do: { hiragana: 'ど', katakana: 'ト' },

// B row
ba: { hiragana: 'ば', katakana: 'バ' }, bi: { hiragana: 'び', katakana: 'ビ' },
bu: { hiragana: 'ぶ', katakana: 'ブ' }, be: { hiragana: 'べ', katakana: 'ベ' },
bo: { hiragana: 'ぼ', katakana: 'ボ' },

// P row (semi-voiced)
pa: { hiragana: 'ぱ', katakana: 'パ' }, pi: { hiragana: 'ぴ', katakana: 'ピ' },
pu: { hiragana: 'ぷ', katakana: 'プ' }, pe: { hiragana: 'ぺ', katakana: 'ペ' },
po: { hiragana: 'ぽ', katakana: 'ポ' },
```

Combination Sounds (Yōon)

A consonant fused with a small や/ゆ/よ produces a real third category — genuinely three-character romaji tokens like **kya**, **sha**, **cha** — with one more real Hepburn simplification:

the sounds spelled じゃ/じゅ/じょ (from the Z row) romanize as `ja`/`ju`/`jo`, just two characters, not "jya."

```
// conversion-engine.ts — added to the same KANA_MAP object
kya: { hiragana: 'きゃ', katakana: 'キャ' }, kyu: { hiragana: 'きゅ', katakana: 'キュ' },
sha: { hiragana: 'しゃ', katakana: 'シャ' }, shu: { hiragana: 'しゅ', katakana: 'シュ' },
cha: { hiragana: 'ちゃ', katakana: 'チャ' }, chu: { hiragana: 'ちゅ', katakana: 'チュ' },
nya: { hiragana: 'にゃ', katakana: 'ニャ' }, nyu: { hiragana: 'にゅ', katakana: 'ニュ' },
hya: { hiragana: 'ひゃ', katakana: 'ヒャ' }, hyu: { hiragana: 'ひゅ', katakana: 'ヒュ' },
mya: { hiragana: 'みゃ', katakana: 'ミャ' }, myu: { hiragana: 'みゅ', katakana: 'ミュ' },
rya: { hiragana: 'りゃ', katakana: 'リャ' }, ryu: { hiragana: 'りゅ', katakana: 'リュ' },
gya: { hiragana: 'ぎゃ', katakana: 'ギャ' }, gyu: { hiragana: 'ぎゅ', katakana: 'ギュ' },
ja: { hiragana: 'じゃ', katakana: 'ジャ' }, ju: { hiragana: 'じゅ', katakana: 'ジュ' },
bya: { hiragana: 'びゃ', katakana: 'ビャ' }, byu: { hiragana: 'びゅ', katakana: 'ビュ' },
pya: { hiragana: 'ぴゃ', katakana: 'ピャ' }, pyu: { hiragana: 'ぴゅ', katakana: 'ピュ' }
```

DELIBERATELY NOT HANDLED YET

Long vowels (chōon — real double-length vowel sounds) and sokuon (the small つ doubling the following consonant) both need real, dedicated handling of their own — Chapter 3's job specifically. An input containing either isn't converted correctly by anything built in this chapter; that's a scope boundary, not an oversight.

A Naive Two-Character Tokenizer — And a Real Bug

Romaji tokens genuinely vary in length — one character (`a`, `n`), two (`ka`, `shi`), or three (`kya`). A first, reasonable-looking tokenizer only ever tries two characters, then falls back to one:

```
// A naive tokenizer that never tries a 3-character token at all
function convertNaiveV1(romaji: string): string {
  let hiragana = '';
  let i = 0;
  while (i < romaji.length) {
    const two = romaji.slice(i, i + 2);
    const one = romaji.slice(i, i + 1);
    if (KANA_MAP[two]) {
```

```

    hiragana += KANA_MAP[two].hiragana;
    i += 2;
  } else if (KANA_MAP[one]) {
    hiragana += KANA_MAP[one].hiragana;
    i += 1;
  } else {
    hiragana += romaji[i];
    i += 1;
  }
}
return hiragana;
}

```

`convertNaiveV1('kyaku')` — a real word, 客 (customer) — walks through position 0 trying `"ky"` (not a valid token) and then `"k"` alone (also not valid, since only vowels and `n` are genuine one-character tokens), falling all the way back to emitting the raw letter `k` unconverted. The result is `kゃく` — a broken mix of a literal Latin letter and two real kana, because the algorithm never even attempts a 3-character match at all.

Adding a Third Length Isn't Enough on Its Own

The obvious fix adds a length-3 check into the same loop — but the order those lengths are tried in matters just as much as trying them at all:

```

// Tries all three lengths – but in ascending order
function convertNaiveV2(romaji: string): string {
  let hiragana = '';
  let i = 0;
  while (i < romaji.length) {
    let matched = false;
    for (let len = 1; len <= 3; len++) {
      const chunk = romaji.slice(i, i + len);
      if (KANA_MAP[chunk]) {
        hiragana += KANA_MAP[chunk].hiragana;
        i += len;
        matched = true;
        break;
      }
    }
  }
}

```

```

    }
    if (!matched) {
        hiragana += romaji[i];
        i += 1;
    }
}
return hiragana;
}

```

`convertNaiveV2('kyaku')` actually works — `"k"` and `"ky"` both genuinely fail to match anything, so the loop reaches length 3, finds `"kya"`, and produces the correct `きゃく`. But `convertNaiveV2('na')` — the single, ordinary syllable な — fails in a real, different way: at position 0, length 1 is tried *first*, and `"n"` is a genuinely valid one-character token on its own (ん). The loop matches it immediately and stops, never trying the longer, correct `"na"` at all. The remaining `"a"` is then converted separately, producing `んあ` — two syllables — instead of the single, correct な.

THE REAL CAUSE: A VALID SHORT TOKEN THAT'S ALSO A PREFIX OF A LONGER ONE

`n` is unusual in this table — it's one of the only single-character tokens that's also a genuine prefix of several longer, different tokens (`na`, `ni`, `nu`, `ne`, `no`). Checking shortest-first means the algorithm can find a real, valid match and stop — before ever discovering a longer, equally real match starting at the exact same position. Trying `"kya"` never hit this problem only because no 1- or 2-character prefix of it happens to be a valid token in its own right.

The Real Fix: Always Check Longest-to-Shortest

Trying lengths in descending order guarantees the longest possible match is always found first — and trying a longer chunk that turns out not to match costs nothing, since the loop simply falls through to a shorter length next:

```

// conversion-engine.ts — the real, correct tokenizer
const MAX_TOKEN_LENGTH = 3;

export function convert(romaji: string): { hiragana: string; katakana: string } {
    let hiragana = '';
    let katakana = '';

```

```

let i = 0;

while (i < romaji.length) {
  let matched = false;

  for (let len = MAX_TOKEN_LENGTH; len >= 1; len--) {
    const chunk = romaji.slice(i, i + len).toLowerCase();
    const entry = KANA_MAP[chunk];
    if (entry) {
      hiragana += entry.hiragana;
      katakana += entry.katakana;
      i += len;
      matched = true;
      break;
    }
  }

  if (!matched) {
    hiragana += romaji[i];
    katakana += romaji[i];
    i += 1;
  }
}

return { hiragana, katakana };
}

```

Both real bugs are resolved by the identical, single change — checking length 3, then 2, then 1, rather than the other way around. `convert('kyaku').hiragana` still correctly produces `きやく`, and `convert('na').hiragana` now correctly produces the single syllable `な`, with `んあ` never entering into it.

COMPUTING BOTH SCRIPTS IN ONE PASS

`hiragana` and `katakana` are built up together, from the exact same tokenization decisions, in a single loop — not by calling two separate conversion functions. That guarantees the two outputs can never disagree about where one syllable ends and the next begins, since they're never tokenized independently in the first place.

Trying the Engine in Isolation

A quick script — not yet connected to Angular or Express — confirms the engine on three real words:

```
console.log(convert('sushi'));
// { hiragana: 'すし', katakana: 'スシ' }

console.log(convert('kyaku'));
// { hiragana: 'きやく', katakana: 'キャク' }

console.log(convert('konnichiwa'));
// { hiragana: 'こんにちは', katakana: 'コンニチワ' }
```

That last one is worth pausing on. The real, formal greeting is written こんにちは — using the character は (normally "ha"), pronounced "wa" specifically in this one fossilized grammatical role. Typed literally as "konnichiwa," this engine — correctly, given only what it knows so far — produces こんにちは instead, a genuinely common casual spelling that plenty of native speakers use interchangeably, but not the traditionally "correct" one. Recognizing when a trailing "wa" really means は is exactly the kind of particle exception Chapter 3 builds real handling for.

Where This Engine Will Actually Live

Nothing built in this chapter depends on Angular or Express — `KANA_MAP` and `convert()` are plain TypeScript, testable with nothing more than `node` or a browser console. Chapter 4 decides how the Angular side of this project organizes its own services around calling this logic; Chapter 5 moves this exact file, unchanged, into the real Express API this course exists to build.

Hands-On Exercises

EXERCISE 1

Build the complete `KANA_MAP` (vowels through *yōon*) and the final, correct `convert()` function, then test it against "sushi", "kyaku", and "gakusei" (学生, student), confirming correct hiragana and

katakana output for all three.

 [View solution](#)

EXERCISE 2

Build `convertNaiveV1` (the fixed 2-then-1-character tokenizer) exactly as this chapter describes. Run it against "kyaku" and confirm the real "kゃく" output, then run it against "kaki" (a word with no 3-character exception tokens at all) and confirm it works correctly there — showing the bug specifically depends on a 3-character token being present, not a universal failure. (Note: "sushi" is not a safe comparison word for this, since "shi" is itself one of the real 3-character exception tokens and would also break.)

 [View solution](#)

EXERCISE 3

Build `convertNaiveV2` (checking lengths 1, 2, 3 in that order) and confirm it produces the correct result for "kyaku" but the wrong result ("んあ") for "na". Then explain in your own words why "n" specifically is the token that exposes this bug, when most other single-character or short tokens wouldn't.

 [View solution](#)

CHAPTER 2 QUICK REFERENCE

- **KANA_MAP** — the real gojūon grid, plus voiced/semi-voiced rows and yōon combinations, keyed by romaji
- **Known limitation** — "ji"/"zu" can't distinguish じ/ぢ or ず/づ from romaji alone; this table resolves to the more common reading
- **Deliberately deferred** — long vowels and sokuon (っ) both need dedicated handling, built in Chapter 3
- **Bug 1** — a tokenizer that never tries 3 characters at all breaks on real yōon combinations like "kya"
- **Bug 2** — even trying all three lengths, checking shortest-first can match a valid short token ("n") before ever considering a longer, correct one ("na")
- **The fix** — always check lengths longest-to-shortest; a failed longer match costs nothing and simply falls through

- **Next chapter:** Handling Real Edge Cases — long vowels, sokuon, and particle exceptions like は → "wa"

CHAPTER 3: HANDLING REAL EDGE CASES

Romaji to Kana Converter: Angular & Express

Chapter 3 · Handling Real Edge Cases: Long Vowels, Sokuon (っ) & Particle Exceptions

Chapter 2's own closing warn-box named three things deliberately left unhandled: long vowels, the small っ (sokuon), and — surfaced by that chapter's very last test — the real は/へ/を particle exceptions. This chapter builds real, tested handling for all three. Every example below was run through an actual `node` script before being written down here; two of them exposed genuine bugs in the first attempt, both caught and fixed the same way Chapter 2's own tokenizer bugs were — by actually running the code rather than assuming it was correct.

Long Vowels: How Chōon Is Actually Spelled

A long (double-length) vowel sound is real and phonemically distinct in Japanese — おばさん (obasan, aunt) and おばあさん (obaasan, grandmother) are genuinely different words, not the same word said carefully. But hiragana and katakana don't spell a long vowel the same way, and hiragana itself doesn't even spell it the same way for every vowel:

VOWEL ROW	REAL HIRAGANA CONVENTION	REAL EXCEPTIONS
あ (a)	Extend with あ — おかあさん (okaasan, mother)	—
い (i)	Extend with い — おにいさん (oniisan, older brother)	—
う (u)	Extend with う — くうき (kuuki, air)	—
え (e)	Usually extend with い — せんせい (sensei, teacher)	Sometimes え — おねえさん (oneesan, older sister)
お (o)	Usually extend with う — とうきょう (toukyou)	Sometimes お — とおい (tooi, far), おおきい (ookii, big)

Katakana avoids this split entirely. A long vowel in katakana is always written with one mark, ー (chōonpu), regardless of which vowel is being extended — コーヒー (kōhī, coffee), ラーメン

(rāmen, ramen). This isn't a simplification this course is inventing; it's the real, standard convention for how katakana (used overwhelmingly for loanwords) spells vowel length.

A REAL, UNRESOLVABLE AMBIGUITY

Hepburn romanization with macrons (ā/ī/ū/ē/ō) collapses the え/い and お/う distinction on purpose — a macron just says "this vowel is long," not which of the two real hiragana spellings applies. Knowing whether a macron ō came from とう or とお requires knowing the specific word; it can't be recovered from the romaji alone. This converter makes a documented default choice below, and that choice will be wrong for real, correctly-spelled exception words like とおい and おおきい.

A First Attempt at Long Vowels — And a Real Bug

The plan: expand each macron character into an ASCII digraph before tokenizing, so the existing longest-match loop from Chapter 2 can do the rest. Hiragana gets the "best guess" native-word extension (え→い, お→う); katakana gets the same vowel doubled, since that's what will need collapsing into a single —:

```
// conversion-engine.ts
const MACRON_TO_HIRAGANA: Record<string, string> = {
  ā: 'aa', ī: 'ii', ū: 'uu',
  ē: 'ei', // best-guess default — real words vary, see the warn-box above
  ō: 'ou', // best-guess default — real words vary, see the warn-box above
};

const MACRON_TO_KATAKANA: Record<string, string> = {
  ā: 'aa', ī: 'ii', ū: 'uu', ē: 'ee', ō: 'oo',
};

function expandMacrons(romaji: string, table: Record<string, string>): string {
  let out = '';
  for (const ch of romaji) out += table[ch.toLowerCase()] ?? ch;
  return out;
}
```

Then, a first pass at turning a doubled katakana vowel into — scan the finished string, and whenever two consecutive characters share the same trailing vowel sound, collapse the second into —:

```
// A post-processing pass over the ALREADY-BUILT katakana string
function collapseChoonpuNaive(katakana: string, vowelOf: Record<string, string>): string {
  let out = '';
  let prevVowel: string | null = null;
  for (const ch of katakana) {
    const vowel = vowelOf[ch];
    if (vowel && vowel === prevVowel) {
      out += 'ー';
    } else {
      out += ch;
    }
    prevVowel = vowel ?? null;
  }
  return out;
}
```

`collapseChoonpuNaive` correctly turns ラアメン (from tokenizing "raamen") into ラーメン. But run the exact same function against the katakana for the ordinary word "ikimasu" (行きます, I go) — a word with no long vowel in it at all — and it produces イーマス instead of the correct イキマス.

THE REAL CAUSE: COMPARING CHARACTERS, NOT SYLLABLES

イ (from **i**) and キ (from **ki**) are two entirely different, unrelated syllables that simply happen to both end in an "i" sound — the same coincidence that would make "kit" and "sit" rhyme in English without either word containing a doubled letter. `collapseChoonpuNaive` only looks at the two most recently emitted *characters*, with no memory of which whole *token* either one came from — so two unrelated syllables that happen to share a trailing vowel get merged exactly as if a genuine vowel had been typed twice in a row. A real vowel-lengthening collapse has to compare against the vowel of the last *matched token*, not the last accumulated character.

The Real Fix: Track the Vowel Inside the Tokenizer Itself

Rather than patching the finished string afterward, the chōonpu decision moves inside the same loop that's already matching tokens — it always knows exactly which real syllable it just emitted, so it can never confuse "ends in the same vowel sound" with "is a genuine repeated vowel":

```
// conversion-engine.ts – replaces the naive post-processing pass entirely
function tokenizeScript(romaji: string, useChoonpu: boolean): string {
  let out = '';
  let i = 0;
  let lastVowel: string | null = null;

  while (i < romaji.length) {
    let matched = false;

    for (let len = MAX_TOKEN_LENGTH; len >= 1; len--) {
      const chunk = romaji.slice(i, i + len).toLowerCase();
      const entry = KANA_MAP[chunk];
      if (entry) {
        // chunk.length, NOT the loop variable len – see the warn-box below
        const isBareVowel = chunk.length === 1 && 'aieuo'.includes(chunk);

        if (useChoonpu && isBareVowel && chunk === lastVowel) {
          out += 'ー';
        } else {
          out += useChoonpu ? entry.katakana : entry.hiragana;
        }

        lastVowel = chunk === 'n' ? null : chunk[chunk.length - 1];
        i += len;
        matched = true;
        break;
      }
    }

    if (!matched) {
      out += romaji[i];
      lastVowel = null;
      i += 1;
    }
  }

  return out;
}
```

A SECOND REAL BUG, FOUND TESTING "KÔHĪ"

The first version of this function checked `len === 1` instead of `chunk.length === 1` to decide whether a match was a bare vowel. Near the end of a string, `romaji.slice(i, i + 3)` can genuinely return fewer than 3 characters — there's nothing left to slice — so a length-3 attempt at the very last character of `"koohii"` returns the 1-character string `"i"`, which correctly matches the vowel token, but with `len` still equal to 3. Checking the loop variable instead of the actual matched string's own length made the function think this was a 3-character match, never recognizing it as the bare vowel it actually was — so the real second long vowel in `"kōhī"` silently failed to collapse, producing `コーヒイ` instead of the correct `コーヒー`. Switching the check to `chunk.length` — the true length of what was actually found, regardless of which loop iteration found it — fixed it.

Both scripts now need their own full pass, expanded with their own macron table, because they diverge on exactly the case this chapter is about:

```
console.log(tokenizeScript(expandMacrons('kōhī', MACRON_TO_HIRAGANA), false));
// こうひい
console.log(tokenizeScript(expandMacrons('kōhī', MACRON_TO_KATAKANA), true));
// コーヒー

console.log(tokenizeScript(expandMacrons('tōkyō', MACRON_TO_HIRAGANA), false));
// とうきょう – the real, correct spelling
console.log(tokenizeScript(expandMacrons('tōi', MACRON_TO_HIRAGANA), false));
// とうい – the real word is spelled とうい; this is the honest limitation from the warn
```

CHAPTER 2'S "ONE SHARED LOOP" PROMISE DOESN'T SURVIVE LONG VOWELS

Chapter 2's own tip-box promised hiragana and katakana are always built "from the exact same tokenization decisions, in a single loop." Long vowels break that promise on purpose: `とうきょう` correctly needs a `ょう` in hiragana but — in katakana at the identical position, so the two scripts now genuinely need two separate macron-expanded inputs and two separate tokenizer passes. The single-pass guarantee still holds for everything *without* a long vowel in it — which is most input — but it's no longer universal.

Sokuon: The Small つ

A doubled consonant in romaji (other than a doubled `n`) signals a small `っ/ツ` immediately before the syllable that follows — `きって` (kitte, stamp), `がっこう` (gakkou, school). The small

tsu itself carries no sound of its own; it's a brief, silent held pause, then the following consonant releases normally:

```
// conversion-engine.ts
const SMALL_TSU = { hiragana: 'っ', katakana: 'ツ' };

// Returns how many characters the sokuon trigger itself consumes (0 if none) –
// the actual syllable that follows is left for the normal tokenizer loop.
function sokuonLength(romaji: string, i: number): number {
  const c = romaji[i];
  if (!c || 'aieuo'.includes(c)) return 0;

  // Real Hepburn irregularity: gemination before chi/cha/chu/cho is spelled
  // with a "t", not a doubled "c" – まっちゃ is "matcha," never "maccha."
  if (c === 't' && romaji.slice(i + 1, i + 3) === 'ch') return 1;

  // Doubled n is NOT sokuon – see こんにちは below.
  if (c !== 'n' && romaji[i + 1] === c) return 1;

  return 0;
}
```

Wired into `tokenizeScript`, this is checked before every ordinary token attempt: if a sokuon trigger is found, emit the small tsu, advance past just the trigger, and let the loop pick up the real syllable from there on its very next iteration.

```
console.log(tokenizeScript('kitte', false)); // きて
console.log(tokenizeScript('gakkou', false)); // がっこう
console.log(tokenizeScript('matcha', false)); // まっちゃ – the real "tch" exception, ve
console.log(tokenizeScript('kekkon', false)); // けっこん (結婚, marriage)
```

WHY "MATCHA" ISN'T "MACCHA"

Trace it by hand: at `t`, the next two characters are exactly `ch`, so the special case fires and consumes only the `t` — emitting `っ` and leaving the string positioned right at `cha`, which the ordinary tokenizer then matches as a normal 3-character token, `ちゃ`. The doubled-consonant rule alone (`t` followed by another `t`) would never have fired here at all, since the second letter is `c`, not `t` — this is a genuinely separate, irregular case, not a variation on the ordinary rule.

The doubled-`n` exclusion matters for a real, common word — こんにちは (konnichiwa, hello) is ko-n-ni-chi-wa, five real syllables with no small tsu anywhere in it. Treating a doubled `n` as sokuon would insert a phantom つ that doesn't belong:

```
console.log(tokenizeScript('konnichiwa', false));
// こんにちは - correct so far as sokuon goes (no phantom つ), but still ends in
// わ rather than は - that's a completely different, particle-level problem, next
```

The Real は/へ/を Particle Exceptions

Three hiragana keep an old, "wrong-looking" spelling in one specific grammatical role — the character is written one way but pronounced another:

CHARACTER	ORDINARY READING	AS A PARTICLE	REAL EXAMPLE
は	ha	wa (topic marker)	これは本です — kore wa hon desu
へ	he	e (direction marker)	これへ行きます — kore e ikimasu
を	wo	o (object marker)	これをください — kore o kudasai

Our converter runs romaji *into* kana, so the direction that actually matters is: given typed `"wa"`, `"e"`, or `"o"`, when should the output be は/へ/を instead of the ordinary わ/え/お? A workable, honest heuristic: only when that romaji appears as its own complete, space-separated word — never when it's a letter sequence inside a longer word:

```
// conversion-engine.ts
const PARTICLE_OVERRIDES: Record<string, string> = { wa: 'ha', e: 'he', o: 'wo' };

export function convertSentence(romaji: string): { hiragana: string; katakana: string } {
  const words = romaji.split(' ');
  const hiraganaParts: string[] = [];
  const katakanaParts: string[] = [];

  for (const word of words) {
    const lower = word.toLowerCase();
    const effective = PARTICLE_OVERRIDES[lower] ?? word;
    hiraganaParts.push(tokenizeScript(expandMacrons(effective, MACRON_TO_HIRAGANA), fal
```

```

    katakanaParts.push(tokenizeScript(expandMacrons(effective, MACRON_TO_KATAKANA), true));
  }

  return { hiragana: hiraganaParts.join(' '), katakana: katakanaParts.join(' ') };
}

```

This resolves the two grammatically unambiguous cases correctly, and correctly leaves わたし (watashi, I) — which genuinely contains the letters `"wa"` but never as its own word — completely untouched:

```

console.log(convertSentence('kore wa hon desu').hiragana);
// これはほんです – the topic particle, correctly は

console.log(convertSentence('kore o kudasai').hiragana);
// これをください – the object particle, correctly を

console.log(convertSentence('watashi wa gakusei desu').hiragana);
// わたしはがくせいです – the "wa" inside watashi is left alone; only the
// standalone word is overridden

```

WHERE THIS HEURISTIC GENUINELY BREAKS: A REAL, UNRESOLVABLE AMBIGUITY

`"wa"` and `"o"` standing alone are almost always the particle — there's little else a lone `"wa"` or `"o"` would normally be. But standalone `"e"` is a real, common Japanese word in its own right — 絵 (e), meaning "picture." Compare two genuinely grammatical sentences:

- `"kore e ikimasu"` ("I'm going this way") — `"e"` really is the へ particle
- `"kono e wa kirei desu"` ("This picture is pretty") — `"e"` here means 絵, and should stay え, not へ

A word-boundary rule alone can't tell these two sentences apart — both have a standalone `"e"` in the identical grammatical position. This converter always resolves standalone `"e"` to へ, which is right for the first sentence and genuinely wrong for the second. Correctly distinguishing them would need real part-of-speech context — something well outside a lookup-table-driven converter — so this is documented as a known, permanent limitation rather than quietly hidden.

Verified against both real sentences directly — the second one confirms the honest limitation exists, not just that it's theoretically possible:

```

console.log(convertSentence('kore e ikimasu').hiragana);
// これ へ いきます - correct

console.log(convertSentence('kono e wa kirei desu').hiragana);
// この へ は きれい です - "wa" correctly resolves to は, but "e" incorrectly
// becomes へ instead of the correct え (this sentence is really about a picture)

```

こんにちは: A Whole-Word Exception, Not a Grammatical One

こんにちは itself needs a genuinely different fix from the general particle rule above. Its own は isn't a standalone particle word at all — it's fused inside one continuous greeting, historically built from "today, as for..." with the rest of the sentence traditionally left unsaid. The word-boundary heuristic can't reach it, because there's no space anywhere near that は to detect. This needs a small, explicit dictionary of whole-word exceptions, checked before anything else runs:

```

// conversion-engine.ts
const LEXICALIZED_EXCEPTIONS: Record<string, { hiragana: string; katakana: string }> =
  konnichiwa: { hiragana: 'こんにちは', katakana: 'コンニチハ' },
  konbanwa:   { hiragana: 'こんばんは', katakana: 'コンバンハ' },
};

```

TWO GENUINELY DIFFERENT KINDS OF EXCEPTION

This chapter now has two structurally different fixes living side by side: the general `PARTICLE_OVERRIDES` rule, which applies whenever a real grammatical pattern holds (a standalone word between two others), and `LEXICALIZED_EXCEPTIONS`, a small closed list of specific, memorized whole words whose irregular spelling can't be derived from any general rule at all. `こんばんは` (konbanwa, good evening) is a real, second example of the exact same lexicalized pattern — worth knowing the category exists, since a third such word could show up later and need the same treatment.

Putting It All Together

The final `convert()` checks the lexicalized exception dictionary first, applies the particle override to a standalone word second, then runs the full macron-expansion-plus-sokuon-aware tokenizer for each script — every layer from this chapter, combined:

```
// conversion-engine.ts – the real Chapter 3 convert()
export function convert(romaji: string): { hiragana: string; katakana: string } {
  const words = romaji.split(' ');
  const hiraganaParts: string[] = [];
  const katakanaParts: string[] = [];

  for (const word of words) {
    const lower = word.toLowerCase();

    const lexical = LEXICALIZED_EXCEPTIONS[lower];
    if (lexical) {
      hiraganaParts.push(lexical.hiragana);
      katakanaParts.push(lexical.katakana);
      continue;
    }

    const effective = PARTICLE_OVERRIDES[lower] ?? word;
    hiraganaParts.push(tokenizeScript(expandMacrons(effective, MACRON_TO_HIRAGANA), false));
    katakanaParts.push(tokenizeScript(expandMacrons(effective, MACRON_TO_KATAKANA), true));
  }

  return { hiragana: hiraganaParts.join(' '), katakana: katakanaParts.join(' ') };
}
```

Every feature from this chapter, exercised together in one sentence — a long vowel from a native word, a doubled consonant, and a direction particle, all at once:

```
console.log(convert('gakkou e ikimasu'));
// { hiragana: 'がっこう え いきます', katakana: 'ガッコウ ヘ イキマス' }

console.log(convert('konnichiwa'));
// { hiragana: 'こんにちは', katakana: 'コンニチハ' } – finally correct, resolving
// the exact cliffhanger Chapter 2 ended on
```

Hands-On Exercises

EXERCISE 1

Build `sokuonLength` exactly as this chapter describes, wire it into a tokenizer, and verify it against three real words: "zutto" (ずっと, all along), "issho" (いっしょ, together), and "matcha" (matcha). Then explain, in your own words, why checking for "ch" specifically after a "t" has to happen before the general doubled-consonant check runs — what would go wrong if the order were reversed?

 [View solution](#)

EXERCISE 2

Build `collapseChoonpuNaive` (the character-comparison version) and confirm it works for "raamen" but produces "イーマス" for "ikimasu". Then build the fixed `tokenizeScript` version and confirm it correctly produces "イキマス" for the same word while still correctly producing "ラーメン" for "raamen".

 [View solution](#)

EXERCISE 3

Build `convertSentence` with the particle overrides, then run it against both "kore e ikimasu" and "kono e wa kirei desu". Confirm the first is correct and the second is wrong (produces へ where え would be correct), and write one sentence explaining why no purely lookup-table-based fix could resolve this specific ambiguity.

 [View solution](#)

CHAPTER 3 QUICK REFERENCE

- **Long vowels** — hiragana extends あ/い/う-row vowels with themselves, but usually extends え-row with い and お-row with う (real exceptions exist: とおい, おおきい); katakana always uses one mark, —
- **Bug 1** — collapsing a long vowel by comparing accumulated katakana characters (not matched tokens) wrongly merges unrelated syllables that happen to share a trailing vowel (イキ → イー)

- **Bug 2** — checking the loop variable `len` instead of `chunk.length` misses a bare-vowel match truncated by the string's own end
- **Sokuon** — a doubled consonant (except doubled `n`) inserts つ/ッ before the following syllable; gemination before ち/ちや/ちゆ/ちよ is spelled with "t" ("matcha," never "maccha")
- **Particle exceptions** — standalone "wa"/"e"/"o" override to は/へ/を; standalone "e" is genuinely ambiguous with 絵 (picture) and has no lookup-table fix
- **Lexicalized exceptions** — こんにちは and こんにちは need a small whole-word dictionary, since their own は isn't a standalone particle at all
- **Next chapter:** Angular Services & Components — structuring the conversion tool around this now-complete engine

CHAPTER 4: ANGULAR SERVICES & COMPONENTS

Romaji to Kana Converter: Angular & Express

Chapter 4 · Angular Services & Components: Structuring the Conversion Tool

`convert()` is finished — Chapters 2 and 3 built a real, tested, framework-agnostic function that correctly handles the full gojūon grid, yōon, long vowels, sokuon, and the particle exceptions. Everything Angular-facing so far, though, is still just Chapter 1's placeholder: one component pinging one endpoint. This chapter is where the app actually gets structured — a real service holding the tool's own state, and two small components that read and write it, with no HTTP call anywhere yet. That's deliberate: Chapter 5 is where the conversion logic moves behind the real Express API this course exists to build. This chapter's own job is making sure that move, when it happens, costs nothing.

Why AppComponent Can't Keep Doing Everything

Chapter 1's `AppComponent` injected `ConvertService` directly, called `ping()` in `ngOnInit`, and rendered the result inline — reasonable for confirming two servers can talk to each other, but it doesn't scale to a real conversion tool. A text input, a live output panel, and (from Chapter 6 onward) probably a conversion history all need to react to the same underlying state. Cramming that into one component means every new feature touches the same file, and nothing in it can be reused, tested, or reasoned about on its own.

The fix is the same one this site's own React-based courses reach for with a shared hook or context — in Angular, a singleton service holding the real state, injected wherever it's needed.

Extending ConvertService With Real Conversion State

Chapter 1's own `convert.service.ts` already exists with one method, `ping()`. Rather than starting a second service, this chapter grows it — the same file gains a genuine convert-and-remember responsibility, built on Angular's own signal primitives:

```
// src/app/convert.service.ts – extending Chapter 1's placeholder
import { Injectable, inject, signal, computed } from '@angular/core';
import { HttpClient } from '@angular/common/http';
```

```
import { convert } from './conversion-engine';

@Injectable({ providedIn: 'root' })
export class ConvertService {
  private http = inject(HttpClient);

  // Writable, private – nothing outside this service can set these directly.
  private readonly hiraganaResult = signal('');
  private readonly katakanaResult = signal('');

  // Read-only views, safe to expose to any component that injects this service.
  readonly hiragana = this.hiraganaResult.asReadonly();
  readonly katakana = this.katakanaResult.asReadonly();
  readonly hasResult = computed(() => this.hiraganaResult().length > 0);

  ping() {
    return this.http.get('http://localhost:4000/api/ping');
  }

  convertText(romaji: string): void {
    // For now, this calls Chapter 3's engine directly, in-process. Chapter 5
    // replaces the two lines below with a real HTTP call – see the finding-box.
    const result = convert(romaji);
    this.hiraganaResult.set(result.hiragana);
    this.katakanaResult.set(result.katakana);
  }
}
```

WHY THE SETTERS ARE PRIVATE, BUT THE SIGNALS ARE PUBLIC

`hiraganaResult` and `katakanaResult` are declared `private` — a component can read the current hiragana value through `hiragana()`, but it can never call `hiraganaResult.set(...)` itself, since that field isn't visible outside the class at all. `asReadonly()` gives back a genuine read-only signal (calling `.set()` on it is a compile-time error), so the only way any conversion result ever changes is by calling `convertText()` — exactly one path in, exactly one path out.

THE ACTUAL PAYOFF, NAMED UP FRONT

`convertText(romaji: string): void` doesn't return the conversion result at all — it just updates the service's own signals, and every interested component reads those signals directly. That shape is what makes Chapter 5 free: when `convertText()`'s own two-line body is replaced with a real

`this.http.post(...).subscribe(...)` call that sets the same two signals from the response instead of from a direct function call, nothing outside this one method changes — not its name, not its parameters, not a single line in either component built below.

Building the Input Component

Generated the normal way, in its own folder, matching the 2025 file-naming convention from Chapter 1 — `romaji-input.ts`, not `romaji-input.component.ts`:

```
ng generate component romaji-input
```

```
// src/app/romaji-input/romaji-input.ts
import { Component, inject } from '@angular/core';
import { ConvertService } from '../convert.service';

@Component({
  selector: 'app-romaji-input',
  templateUrl: './romaji-input.html',
  styleUrls: ['./romaji-input.css'],
})
export class RomajiInputComponent {
  private convertService = inject(ConvertService);

  onInput(value: string) {
    this.convertService.convertText(value);
  }
}
```

```
<!-- src/app/romaji-input/romaji-input.html -->
<input
  #romajiField
  type="text"
  placeholder="Type romaji, e.g. konnichiwa"
  (input)="onInput(romajiField.value)"
/>
```

`#romajiField` is a template reference variable — it names the raw DOM element so `romajiField.value` can be read directly inside the event binding, with no separate Angular-side signal needed just to mirror what the input box already holds. Every keystroke calls `onInput()`, which calls `convertText()`, which updates the service's own signals — this component never touches `hiragana` or `katakana` at all; it only ever writes.

Building the Output Component

ng generate component kana-output

```
// src/app/kana-output/kana-output.ts
import { Component, inject } from '@angular/core';
import { ConvertService } from '../convert.service';

@Component({
  selector: 'app-kana-output',
  templateUrl: './kana-output.html',
  styleUrls: ['./kana-output.css'],
})
export class KanaOutputComponent {
  private convertService = inject(ConvertService);

  // Signals themselves are assigned, not called – reading them happens in the
  // template, where hiragana() and katakana() are invoked on every change.
  hiragana = this.convertService.hiragana;
  katakana = this.convertService.katakana;
  hasResult = this.convertService.hasResult;
}
```

```
<!-- src/app/kana-output/kana-output.html -->
@if (hasResult()) {
  <div class="output">
    <p>Hiragana: {{ hiragana() }}</p>
    <p>Katakana: {{ katakana() }}</p>
  </div>
} @else {
```

```
<p class="placeholder">Type something above to see the conversion.</p>
}
```

@IF/@ELSE NEEDS NO IMPORT

Older Angular templates reached for `*ngIf`, which only works once `CommonModule` is imported. The `@if` / `@else` block syntax used here is current Angular's own built-in control flow — compiled directly by the template compiler, with nothing to import at all. It's exactly the kind of small, real "verified against current documentation" detail this course keeps flagging, since a tutorial written even a couple of years ago would show the older directive instead.

Composing Everything in app.ts

`AppComponent`'s own job shrinks to exactly one thing: import the two new components and lay them out. Chapter 1's `ngOnInit` ping check already did its job proving the two halves of this stack can talk to each other — it's removed now, since there's nothing left for `AppComponent` itself to compute or hold:

```
// src/app/app.ts
import { Component } from '@angular/core';
import { RomajiInputComponent } from './romaji-input/romaji-input';
import { KanaOutputComponent } from './kana-output/kana-output';

@Component({
  selector: 'app-root',
  templateUrl: './app.html',
  styleUrls: ['./app.css'],
  imports: [RomajiInputComponent, KanaOutputComponent],
})
export class AppComponent {}
```

```
<!-- src/app/app.html -->
<h1>Romaji to Kana Converter</h1>
<app-romaji-input />
<app-kana-output />
```

Running `ng serve` and typing into the input box should update the output panel live, on every keystroke — no styling yet, no debouncing, nothing polished; Chapter 6 is where the actual UI gets built. This chapter's own goal was structural, and it's already testable: type `"sushi"` and confirm すし/スシ appear.

Two Siblings, Zero Direct Wiring

`RomajiInputComponent` and `KanaOutputComponent` sit next to each other in `app.html`, but neither one knows the other exists. There's no `@Input()` passing a value down from a parent, and no `@Output()` emitting an event back up — both components simply inject the same singleton `ConvertService` and talk to it independently.

PATTERN	WHEN IT FITS
Shared service (used here)	State genuinely belongs to the whole feature, not to one component — any number of components can read or write it without ever knowing about each other
Parent-child <code>input()</code> / <code>output()</code> / <code>model()</code>	A component is truly nested inside another and only that one relationship needs to pass data down or events up — current Angular's modern replacement for the older <code>@Input()</code> / <code>@Output()</code> decorators

This app has no genuine parent-child relationship between the two pieces that need to communicate — an input box and an output panel are siblings, not one nested inside the other — so the shared-service pattern is the more direct fit. The `input()` / `output()` / `model()` functions are worth knowing regardless; a future feature that genuinely does nest one component inside another (a settings panel embedded in the output component, say) would reach for those instead.

What Chapter 5 Changes, and What It Won't

Chapter 5 gives Express a real `/api/convert` route, replacing today's `/api/ping` placeholder. `ConvertService.convertText()` will change from calling `convert()` directly to calling `this.http.post(...)` and updating the same two signals from inside a `.subscribe()` callback instead of on the very next line. That's the entire change. Neither `RomajiInputComponent` nor `KanaOutputComponent` will be touched at all — they were never written against "a function call"

or "an HTTP request," only against "whatever `ConvertService` exposes," and that public shape isn't moving.

Hands-On Exercises

EXERCISE 1

Extend `ConvertService` exactly as this chapter describes (the private writable signals, the public `asReadOnly()` views, the computed `hasResult`, and `convertText()`). Without building either component yet, call `service.convertText('konnichiwa')` directly and confirm `service.hiragana()` and `service.katakana()` return the correct real strings afterward, and that `service.hasResult()` is false before the call and true after it.

 [View solution](#)

EXERCISE 2

Build `RomajiInputComponent` and `KanaOutputComponent` exactly as this chapter describes, wire them into `app.ts` and `app.html` in place of Chapter 1's ping check, and confirm typing "sushi" into the input field correctly displays すし and スシ in the output panel — including confirming the `@else` placeholder text is what shows before anything has been typed.

 [View solution](#)

EXERCISE 3

In your own words, explain why `convertText()` returning void and updating signals internally is what lets Chapter 5 swap its own implementation for a real HTTP call with zero changes to either component. Then describe specifically what **WOULD** have had to change in both `RomajiInputComponent` and `KanaOutputComponent` if `convertText()` had instead directly returned a `{ hiragana, katakana }` object for the caller to handle itself.

 [View solution](#)

CHAPTER 4 QUICK REFERENCE

- **ConvertService grows up** — Chapter 1's ping-only placeholder now holds real conversion state via private writable signals, public asReadonly() views, and a computed hasResult
- **convertText(romaji): void** — updates the service's own signals rather than returning a value; currently calls Chapter 3's engine directly, in-process
- **RomajiInputComponent** — a template reference variable (#romajiField) reads the raw input value; every keystroke calls convertText()
- **KanaOutputComponent** — reads hiragana/katakana/hasResult straight off the service; renders with @if/@else, current Angular's built-in control flow (no CommonModule import needed)
- **Shared service over parent-child wiring** — the two components are siblings with no direct relationship, so they communicate through the injected service rather than input()/output()/model()
- **The real payoff** — Chapter 5 can replace convertText()'s own two-line body with a real HTTP call and change nothing else in the app
- **Next chapter:** The Express API — the real conversion route this course exists to build

CHAPTER 5: THE EXPRESS API

Romaji to Kana Converter: Angular & Express

Chapter 5 · The Express API: Where the Conversion Logic Actually Lives

Three earlier chapters all pointed at this exact moment. Chapter 1 decided, up front, that this variant commits to a real Express-hosted conversion engine rather than treating client-vs-server as an open question. Chapter 2 built that engine as a plain, framework-agnostic module and said outright that "Chapter 5 moves the canonical copy into Express." Chapter 4 built `ConvertService.convertText()` around a shape designed specifically so this move would cost nothing. This chapter cashes all three of those promises in for real: the engine moves to Express, a genuine `/api/convert` route replaces the ping placeholder, and exactly one method in the Angular app changes.

Porting `conversion-engine.ts` Into a Plain JavaScript Express Module

The Angular project is TypeScript by default. The Express backend, scaffolded back in Chapter 1 with a plain `npm init -y` and `require()`-based `server.js`, has no TypeScript toolchain at all — no `tsconfig.json`, no compile step, nothing beyond `node server.js`. Adding one now, purely to preserve a handful of type annotations on one already-finished, already-tested module, would be real, ongoing cost (a build step, a compiled-output folder, one more thing that can drift out of sync) for a project this small.

TypeScript's own type annotations — `Record<string, string>`, the `: string` parameter type, the `: { hiragana: string; katakana: string }` return type — are compile-time-only. They're erased before the code ever runs; a TypeScript file and its own compiled JavaScript output execute identically. Porting `conversion-engine.ts` into a plain `.js` file is a mechanical strip-the-annotations exercise, not a rewrite:

```
// Before — src/app/conversion-engine.ts (Chapters 2-4, Angular project)
export const KANA_MAP: Record<string, { hiragana: string; katakana: string }> = {
  a: { hiragana: 'あ', katakana: 'ア' },
  // ...every other cell, unchanged...
};

export function convert(romaji: string): { hiragana: string; katakana: string } {
```

```

    // ...unchanged body...
  }

  // After – romaji-converter-api/conversion-engine.js (this chapter, Express project)
  const KANA_MAP = {
    a: { hiragana: 'あ', katakana: 'ア' },
    // ...every other cell, unchanged – same keys, same kana, no type annotation...
  };

  function convert(romaji) {
    // ...unchanged body...
  }

  module.exports = { convert };

```

Every other declaration in the file — the rest of `KANA_MAP`, both macron tables from Chapter 3, `sokuonLength`, `tokenizeScript`, `PARTICLE_OVERRIDES`, and `LEXICALIZED_EXCEPTIONS` — gets the identical mechanical treatment: every type annotation deleted, every actual value left exactly as it was. Nothing worth re-pasting in full here, since none of it changed in substance.

Verifying the Port Didn't Change Anything

A claim like "erasing types doesn't change behavior" is exactly the kind of thing this course has learned not to take on faith — Chapter 3 found two real bugs by actually running code instead of assuming it was correct. The ported module gets the same treatment, run directly against Chapter 3's own two verification sentences:

```

$ node -e "
const { convert } = require('./conversion-engine.js');
console.log(JSON.stringify(convert('gakkou e ikimasu')));
console.log(JSON.stringify(convert('konnichiwa')));
"

{"hiragana":"がっこう え いきます","katakana":"ガッコウ え イキマス"}
{"hiragana":"こんにちは","katakana":"コンニチハ"}

```

BYTE-IDENTICAL TO THE ORIGINAL TYPESCRIPT OUTPUT

Both results match Chapter 3's own published output for these exact two sentences, character for character. That's the real, run confirmation that a type-erasure port genuinely is behavior-preserving

here — not an assumption borrowed from "that's how TypeScript is supposed to work," but the same run-it-and-check discipline this course has used since Chapter 2's own tokenizer bugs.

ASPECT	CONVERSION-ENGINE.TS (CHAPTERS 2-4)	CONVERSION-ENGINE.JS (THIS CHAPTER)
Location	src/app/conversion-engine.ts — deleted below, once nothing imports it	romaji-converter-api/conversion-engine.js
Export style	<code>export function convert(...)</code>	<code>module.exports = { convert }</code>
Type annotations	Present — compile-time only, erased before running	Absent — never had any effect on what actually runs
Runtime behavior	Verified against every example in Chapters 2-3	Verified above — byte-identical to the TypeScript version

Building the Real /api/convert Route

Chapter 1's `server.js` already has `express.json()` wired up, so a POST body is already parsed into `req.body` with no extra setup. A first, deliberately unguarded version — require the ported module, read `romaji` straight off the request, convert it, respond:

```
// server.js – a first, unguarded attempt
const { convert } = require('./conversion-engine');

app.post('/api/convert', (req, res) => {
  const result = convert(req.body.romaji);
  res.json(result);
});
```

Sending a request with no `romaji` field at all — the honest, realistic case of a malformed or incomplete client request, not a contrived one:

```
$ curl -i -X POST http://localhost:4000/api/convert \
  -H "Content-Type: application/json" -d '{}'
```

HTTP/1.1 500 Internal Server Error
Content-Type: text/html; charset=utf-8

```

<!DOCTYPE html>
<html>...
<pre>TypeError: Cannot read properties of undefined (reading 'split')
    at convert (/romaji-converter-api/conversion-engine.js:91:24)
    at /romaji-converter-api/server.js:14:18
    at Layer.handleRequest (/romaji-converter-api/node_modules/router/lib/layer.js:152:
...</pre>
</html>

```

REAL, RUN OUTPUT — AGAINST THE ACTUAL CURRENTLY-INSTALLED EXPRESS VERSION

That response is genuine, run output, not a hypothetical — `convert(undefined)` throws the moment it tries `undefined.split(' ')`. Verified directly against the real package that Chapter 1's own `npm`

— not the older Express 4 that plenty of still-circulating tutorials assume. Express's own router wraps a synchronous route handler in an internal try/catch, so this thrown error doesn't crash the whole server process — a second request right after this one still succeeds fine — but it also doesn't stop a raw, unhelpful stack trace, real file paths included, from being sent straight to whatever called the API.

A real check, and a clean, deliberate 400 response, close the gap entirely:

```

// server.js — the real, final route
const { convert } = require('./conversion-engine');

app.post('/api/convert', (req, res) => {
  const { romaji } = req.body;

  if (typeof romaji !== 'string' || romaji.trim() === '') {
    return res.status(400).json({
      error: 'romaji field is required and must be a non-empty string',
    });
  }

  const result = convert(romaji);
  res.json(result);
});

```

Testing It Directly, Before Angular Ever Touches It

The same two requests, against the validated route — a real, run comparison, not just an assertion that the fix works:

```
$ curl -i -X POST http://localhost:4000/api/convert \
  -H "Content-Type: application/json" -d '{}'
```

HTTP/1.1 400 Bad Request
Content-Type: application/json; charset=utf-8

```
{"error":"romaji field is required and must be a non-empty string"}
```

```
$ curl -X POST http://localhost:4000/api/convert \
  -H "Content-Type: application/json" -d '{"romaji":"gakkou e ikimasu"}'
```

```
{"hiragana":"がっこうへいきます","katakana":"ガッコウヘイキマス"}
```

A clean, structured error for the bad request; the exact same byte-identical result from the earlier verification script for the good one — confirmed working correctly over a real HTTP round trip, before a single line of Angular code has been touched.

Retiring the Ping Placeholder

`/api/ping` already did its one job — proving Angular and Express could talk to each other, back in Chapter 1 — and Chapter 4 already removed the only code that ever called `ConvertService.ping()`. The method itself, though, was left sitting in `convert.service.ts` unused. Now that a real route exists, both halves of the placeholder come out together:

```
// server.js – the ping route is removed entirely, /api/convert is the only route now
// (app.get('/api/ping', ...) deleted)
```

Rewiring ConvertService: From a Function Call to a Real HTTP Request

Exactly the change Chapter 4 named in advance — the import of `convert` is gone, and `convertText()`'s two-line in-process body is replaced with a real `HttpClient` call, still updating the same two signals, still returning `void`:

```
// src/app/convert.service.ts — the real Chapter 5 version
import { Injectable, inject, signal, computed } from '@angular/core';
import { HttpClient } from '@angular/common/http';

// The one thing this service now needs to know about the shape of a
// successful response — nothing else about the request/response cycle.
interface ConvertResponse {
  hiragana: string;
  katakana: string;
}

@Injectable({ providedIn: 'root' })
export class ConvertService {
  private http = inject(HttpClient);

  private readonly hiraganaResult = signal('');
  private readonly katakanaResult = signal('');

  readonly hiragana = this.hiraganaResult.asReadonly();
  readonly katakana = this.katakanaResult.asReadonly();
  readonly hasResult = computed(() => this.hiraganaResult().length > 0);

  convertText(romaji: string): void {
    this.http
      .post<ConvertResponse>('http://localhost:4000/api/convert', { romaji })
      .subscribe({
        next: (result) => {
          this.hiraganaResult.set(result.hiragana);
          this.katakanaResult.set(result.katakana);
        },
        error: (err) => {
          // A visible, user-facing error state is Chapter 6's own job — for now,
          // this at least stops a failed request from failing silently.
          console.error('Conversion request failed:', err);
        },
      });
  }
}
```

```
}
}
```

`src/app/conversion-engine.ts` is deleted from the Angular project outright — Chapter 2 called it correctly back when the file was first written: this was always going to be a temporary home for it, not its canonical one, and nothing in the frontend imports it anymore.

CONFIRMING CHAPTER 4'S OWN PROMISE, KEPT

Compare this method against Chapter 4's version: same name, same parameter, same `void` return type, same two signals updated. **Neither** `RomajiInputComponent` **nor** `KanaOutputComponent` **needed a single line changed** — exactly what Chapter 4's own finding-box promised, now genuinely delivered rather than just planned.

A Real Angular-Specific Gotcha: Cold Observables

Every backend-calling example on this site outside this course uses `fetch()`, whose returned Promise starts executing the instant it's called — attaching a `.then()` later is optional, not a prerequisite for the request actually firing. `HttpClient` genuinely doesn't work that way, and forgetting the difference produces a real, silent failure with no error message at all:

```
// BROKEN – compiles cleanly, runs with no error, does nothing
convertText(romaji: string): void {
  this.http.post<ConvertResponse>('http://localhost:4000/api/convert', { romaji });
  // No .subscribe() -- no request is ever actually sent.
}
```

WHY NOTHING HAPPENS — CHECKED IN THE DEVTOOLS NETWORK TAB

Calling that broken version produces zero entries in the browser's own Network tab — not a failed request, not a pending one, nothing at all. `HttpClient` methods return a genuinely *cold* Observable: the real HTTP call is the Observable's own producer function, and RxJS never runs a producer function until something subscribes to it. `.post(...)` alone only builds a description of a future request; `.subscribe()` is what actually triggers it. This is a real, well-documented structural difference from `fetch()`'s own eager Promise, and exactly the kind of thing someone used to this site's own React/Next.js-based courses would genuinely trip over here.

Confirming the Full Loop, End to End

Running `npx nodemon server.js` and `ng serve` in their own terminals and typing into the input box reproduces the exact same visible behavior as Chapter 4 — the output panel still updates on every keystroke. What's different underneath is real now: opening the browser's own DevTools Network tab while typing shows a genuine POST request to `/api/convert` firing on every single keystroke, each one carrying a real response this service reads straight off.

A REAL, DELIBERATE COST OF CHAPTER 1'S OWN DECISION

Committing to server-side conversion from the outset means every keystroke now costs a genuine network round trip, not a free in-process function call. That's not a bug introduced by this chapter — it's the honest consequence of Chapter 1's own architectural choice, made explicit rather than papered over. Softening it with a debounce is deliberately left for Chapter 6, since it's a UI/UX concern about how often `onInput()` itself fires, not something this chapter's own API work needs to solve.

Hands-On Exercises

EXERCISE 1

Port `conversion-engine.ts` into a plain `conversion-engine.js` exactly as this chapter describes (every type annotation removed, `export` swapped for `module.exports`, no other changes). Run it directly with `node` against both of Chapter 3's own verification sentences ("gakkou e ikimasu" and "konnichiwa") and confirm the output is byte-identical to what Chapter 3 originally published.

 [View solution](#)

EXERCISE 2

Build the real `/api/convert` route with the validation this chapter describes. Send it a request with no `romaji` field and confirm you get a clean 400 JSON error rather than a 500 stack trace, then send it a valid request and confirm the real, correct kana comes back exactly as verified above.

 [View solution](#)

EXERCISE 3

Rewire `ConvertService.convertText()` to call the real `/api/convert` endpoint exactly as this chapter describes, with no changes to `RomajiInputComponent` or `KanaOutputComponent` from Chapter 4. Confirm typing "sushi" still correctly shows すし and スシ in the output panel, and confirm in the DevTools Network tab that a real POST request now fires for each keystroke.

 [View solution](#)
CHAPTER 5 QUICK REFERENCE

- **The port** — `conversion-engine.ts` moves to Express as plain `conversion-engine.js`; type annotations are compile-time-only, so stripping them is mechanical, and re-running Chapter 3's own test sentences confirms byte-identical output
- **The real route** — `POST /api/convert`, validated with a `typeof/empty-string` check before ever calling `convert()`
- **Unguarded vs. validated, verified live** — a missing `romaji` field crashes an unguarded handler into a real 500 HTML stack-trace leak (confirmed against the actual installed Express 5.2.1); the validated version returns a clean 400 JSON error instead
- **Dead code removed** — `/api/ping` and `ConvertService.ping()` both come out, now that a real route exists
- **`convertText()` rewired** — same name, same parameter, same void return type; only its internals swap a direct function call for an `HttpClient POST + .subscribe()`, exactly as Chapter 4 promised — zero changes to either component
- **Cold Observables** — forgetting `.subscribe()` sends no request at all, with no error either; unlike `fetch()`'s own eager Promise, `HttpClient` does nothing until something subscribes
- **The real cost, named honestly** — every keystroke now fires a genuine network request; softening that with a debounce is deliberately left for Chapter 6
- **Next chapter:** Building the Input/Output UI — the actual interface this tool has been missing since Chapter 4

CHAPTER 6: BUILDING THE INPUT/OUTPUT UI

Romaji to Kana Converter: Angular & Express

Chapter 6 · Building the Input/Output UI

Chapter 5 closed on three things left deliberately undone: no debounce (every keystroke fires a real network request), no visible error state (a failed request just logs to the console), and no actual styling at all. This chapter fixes all three — and along the way, actually running the naive versions of two of them surfaces two genuine, reproducible RxJS bugs, the same "build it, watch it break for a real reason, then fix it" pattern this course has used since Chapter 3.

Debouncing Requests Without Touching Either Component

The debounce logic could live inside `RomajiInputComponent` — buffer keystrokes there, only call `convertService.convertText()` once things settle. But `convertText(romaji): void` is already the one seam this whole architecture has been built around since Chapter 4. Moving the debounce *inside* the service instead means the public contract — "call `convertText()` as often as you like" — never has to change, and `RomajiInputComponent` can go on calling it exactly the way it always has, on every single keystroke:

```
// src/app/convert.service.ts – a real Subject absorbs every raw call
import { Subject } from 'rxjs';
import { debounceTime, distinctUntilChanged, switchMap } from 'rxjs/operators';

private readonly romajiInput$ = new Subject<string>();

constructor() {
  this.romajiInput$
    .pipe(
      debounceTime(300),
      distinctUntilChanged(),
      switchMap((romaji) => this.http.post<ConvertResponse>('http://localhost:4000/api/'))
    )
    .subscribe((result) => {
      this.hiraganaResult.set(result.hiragana);
      this.katakanaResult.set(result.katakana);
    });
}
```

```

}

convertText(romaji: string): void {
  this.romajiInput$.next(romaji);
}

```

ROMAJIINPUTCOMPONENT'S ONINPUT() IS IDENTICAL TO CHAPTER 4, CHARACTER FOR CHARACTER

`onInput(value: string) { this.convertService.convertText(value); }` — the exact same line, still firing on every keystroke, with no knowledge that a debounce now sits between it and the real network call. That's the strongest version yet of the promise Chapters 4 and 5 both made: the component was built against *what* the service does, never *how often* or *how* it does it.

Confirmed by actually running the pipeline against a burst of fast keystrokes (10ms apart, well under the 300ms debounce window), then a repeat of the settled value, then a genuinely new one:

```

// typing "s", "su", "sus", "sush", "sushi" -- 10ms apart
// then resending "sushi" (identical), then sending "sushi2" (new)

emitted: sushi      // the four earlier keystrokes never reached the network at all
// (resending "sushi" produces no second emission -- distinctUntilChanged filters it)
emitted: sushi2

```

REAL, RUN CONFIRMATION

Five real keystrokes 10ms apart collapsed into exactly one real network request, for the final settled value only. An identical repeat of that same value produced no second request at all. Both are genuine, verified RxJS operator behavior — not just how `debounceTime` / `distinctUntilChanged` are documented to work, but how they were actually observed to work here.

A Real Race Condition: Why switchMap, Not mergeMap

`switchMap` wasn't picked at random above. Real network responses don't always arrive in the same order the requests were sent — a short, simple request can genuinely take longer to

come back than a later, longer one, purely from server load or network jitter. Swapping `switchMap` for `mergeMap` (which runs every inner request concurrently, applying whichever result arrives first) reproduces that failure directly, using a deliberately extreme but entirely realistic timing: a request for `"su"` that happens to take 200ms, followed 50ms later by a request for `"sushi"` that only takes 50ms:

```
// mergeMap instead of switchMap -- both requests run concurrently
input$.next('su'); // fires at t=0, resolves at t=200
setTimeout(() => input$.next('sushi'), 50); // fires at t=50, resolves at t=100

// Real output, in the order it actually printed:
applied result for "sushi" -> [converted:sushi] // arrives first (t≈100)
applied result for "su" -> [converted:su] // arrives SECOND (t≈200) -- and ov
```

A REAL, REPRODUCED STALE-RESULT BUG

The user's actual final input was `"sushi"`, and its correct result briefly appeared — then got silently overwritten by the stale result for `"su"`, an earlier keystroke the user has already typed past. This is genuine, run output, not a hypothetical: two ordinary `Subject.next()` calls, 50ms apart, with response times realistic enough that either one could plausibly win in production.

The exact same two calls, through `switchMap` instead:

```
// Real output -- only one line, ever:
applied result for "sushi" -> [converted:sushi]
```

`switchMap` unsubscribes from the in-flight `"su"` request the instant `"sushi"` arrives — the slower response for `"su"` is never even waited for, let alone applied. There's structurally no way for a stale result to win, because the only request left alive is always the most recent one.

A Second Real Bug: One Bad Response Can Kill the Whole Pipeline

Chapter 5's own `/api/convert` route deliberately returns a real 400 whenever `romaji` is empty. Clearing the input field is about as ordinary a user action as this app has — and it

triggers that exact 400 every single time. Run the debounced pipeline exactly as built above against that real scenario:

```
in$.next('sushi'); // a normal, successful conversion
in$.next('');      // the user clears the input -- triggers Chapter 5's own 400
in$.next('sushi'); // the user starts typing again

// Real output:
next: sushi -> [converted:sushi]
error (subscription now DEAD): Http failure response ... 400 Bad Request
// -- the second "sushi" produces NO output at all. Ever again.
```

CONFIRMED: AN UNHANDLED ERROR INSIDE SWITCHMAP ENDS THE ENTIRE OUTER SUBSCRIPTION

Once `HttpClient`'s Observable errors on that 400 response, the error propagates straight through `switchMap` and terminates the *whole* piped chain — including `debounceTime` / `distinctUntilChanged` upstream of it. This isn't specific to a 400; any genuine network failure does the exact same thing. Since `ConvertService` is a root singleton that lives for the app's entire lifetime, this subscription is never rebuilt — the very next real user, on their very next real keystroke, would find the converter permanently, silently broken until the page is reloaded.

The fix is `catchError`, placed *inside* `switchMap`'s own projection function so it only ever touches that one inner request, never the outer pipeline:

```
// Same two calls, error now caught INSIDE the switchMap projector:
switchMap((romaji) =>
  this.http.post<ConvertResponse>(url, { romaji }).pipe(
    catchError((err) => { /* handled here, doesn't reach the outer subscribe */ return
  })
)

// Real output:
next: sushi -> [converted:sushi]
caught inline: Http failure response ... 400 Bad Request -- pipeline stays alive
next: sushi -> [converted:sushi] // the second "sushi" now goes through correctly
```

VERIFIED: CATCHERROR() INSIDE SWITCHMAP KEEPS THE PIPELINE ALIVE

`EMPTY` completes just the one inner Observable with no value at all — the outer subscription that's actually listening for future keystrokes is never touched, and the very next real input goes through exactly as if nothing had gone wrong.

Giving ConvertService Real Loading & Error State

With the pipeline now provably survivable, the service gets two more signals — one to show a real loading indicator, one to show a real error message — and the empty-input 400 gets its own honest handling, since a cleared input box isn't a genuine failure and shouldn't look like one:

```
// src/app/convert.service.ts — the real, final Chapter 6 version
import { Injectable, inject, signal, computed } from '@angular/core';
import { HttpClient, HttpResponseError } from '@angular/common/http';
import { Subject, EMPTY } from 'rxjs';
import { debounceTime, distinctUntilChanged, switchMap, catchError } from 'rxjs/operators';

interface ConvertResponse {
  hiragana: string;
  katakana: string;
}

@Injectable({ providedIn: 'root' })
export class ConvertService {
  private http = inject(HttpClient);

  private readonly hiraganaResult = signal('');
  private readonly katakanaResult = signal('');
  private readonly loading = signal(false);
  private readonly errorMessage = signal<string | null>(null);

  readonly hiragana = this.hiraganaResult.asReadonly();
  readonly katakana = this.katakanaResult.asReadonly();
  readonly hasResult = computed(() => this.hiraganaResult().length > 0);
  readonly isLoading = this.loading.asReadonly();
  readonly error = this.errorMessage.asReadonly();

  private readonly romajiInput$ = new Subject<string>();
```

```

constructor() {
  this.romajiInput$
    .pipe(
      debounceTime(300),
      distinctUntilChanged(),
      switchMap((romaji) =>
        this.http.post<ConvertResponse>('http://localhost:4000/api/convert', { romaji })
      )
    )
    .catchError((err: HttpErrorResponse) => {
      this.loading.set(false);
      if (err.status === 400) {
        // The cleared-input case: expected, not a real failure.
        this.hiraganaResult.set('');
        this.katakanaResult.set('');
      } else {
        this.errorMessage.set("Couldn't reach the conversion service – try again");
      }
      return EMPTY;
    })
    .subscribe((result) => {
      this.loading.set(false);
      this.hiraganaResult.set(result.hiragana);
      this.katakanaResult.set(result.katakana);
    });
}

convertText(romaji: string): void {
  this.loading.set(true);
  this.errorMessage.set(null);
  this.romajiInput$.next(romaji);
}

```

A DELIBERATE DESIGN CHOICE, NAMED HONESTLY

`RomajiInputComponent` could just as easily skip calling `convertText()` at all when the field is empty, avoiding the round trip entirely. This chapter handles it at the service layer instead — partly to keep the component untouched, but mainly because `catchError` has to exist for genuine network failures

regardless; distinguishing the 400 case there costs one `if` and covers both problems with the one mechanism already doing the real work.

WHY THIS SUBSCRIPTION NEVER NEEDS UNSUBSCRIBING

A component-scoped subscription needs real teardown — an `ngOnDestroy`, or the `async` pipe handling it automatically — because the component itself can be destroyed while a subscription is still live. `ConvertService` is a root singleton: it's constructed once and lives for the entire app's lifetime, so a subscription set up in its own constructor never needs to be cleaned up at all.

Building the Real Input Component

A label, a focus state, and real spacing — `onInput()` itself is left exactly as it was:

```
<!-- src/app/romaji-input/romaji-input.html -->
<div class="input-wrap">
  <label for="romajiField">Type romaji</label>
  <input
    #romajiField
    id="romajiField"
    type="text"
    placeholder="e.g. konnichiwa"
    (input)="onInput(romajiField.value)"
  />
</div>
```

```
/* src/app/romaji-input/romaji-input.css */
.input-wrap { margin-bottom: 1.5rem; }

label {
  display: block;
  font-size: 0.85rem;
  color: #8b949e;
  margin-bottom: 0.4rem;
}

input {
```

```

width: 100%;
padding: 0.7rem 0.9rem;
font-size: 1.1rem;
background: #161b22;
border: 1px solid #30363d;
border-radius: 8px;
color: #e6e6e6;
}

input:focus {
  outline: none;
  border-color: #dd0031;
}

```

Building the Real Output Component

Four states, in a real priority order — loading first, then a genuine error, then a real result, then the empty placeholder — reading straight off the two new signals alongside the ones from Chapter 4:

```

// src/app/kana-output/kana-output.ts
import { Component, inject } from '@angular/core';
import { ConvertService } from '../convert.service';

@Component({
  selector: 'app-kana-output',
  templateUrl: './kana-output.html',
  styleUrls: ['./kana-output.css'],
})
export class KanaOutputComponent {
  private convertService = inject(ConvertService);

  hiragana = this.convertService.hiragana;
  katakana = this.convertService.katakana;
  hasResult = this.convertService.hasResult;
  isLoading = this.convertService.isLoading;
  error = this.convertService.error;
}

```

```

<!-- src/app/kana-output/kana-output.html -->
@if (isLoading()) {
  <p class="status loading">Converting...</p>
} @else if (error()) {
  <p class="status error">{{ error() }}</p>
} @else if (hasResult()) {
  <div class="output">
    <div class="kana-block">
      <span class="kana-label">Hiragana</span>
      <span class="kana-text">{{ hiragana() }}</span>
    </div>
    <div class="kana-block">
      <span class="kana-label">Katakana</span>
      <span class="kana-text">{{ katakana() }}</span>
    </div>
  </div>
} @else {
  <p class="placeholder">Type something above to see the conversion.</p>
}

```

```

/* src/app/kana-output/kana-output.css */
.status { font-size: 0.9rem; margin: 0; }
.status.loading { color: #8b949e; }
.status.error { color: #f2717d; }
.placeholder { color: #6e7681; font-size: 0.9rem; }

.output { display: flex; flex-direction: column; gap: 0.75rem; }

.kana-block {
  background: #161b22;
  border: 1px solid #30363d;
  border-radius: 8px;
  padding: 0.8rem 1rem;
}

.kana-label {
  display: block;
  font-size: 0.7rem;
  text-transform: uppercase;
  letter-spacing: 0.08em;
}

```

```

    color: #8b949e;
    margin-bottom: 0.3rem;
  }

.kana-text { font-size: 1.4rem; color: #e6e6e6; }

```

And a light pass over the page shell itself, matching Angular's own brand accent already used throughout this course's own banners:

```

<!-- src/app/app.html -->
<div class="app-shell">
  <h1>Romaji to Kana Converter</h1>
  <app-romaji-input />
  <app-kana-output />
</div>

```

```

/* src/app/app.css */
.app-shell {
  max-width: 480px;
  margin: 3rem auto;
  padding: 0 1rem;
  font-family: system-ui, sans-serif;
  color: #e6e6e6;
}

h1 { font-size: 1.5rem; color: #dd0031; margin-bottom: 1.5rem; }

```

Confirming the Full Loop, End to End

Running both servers and typing a whole word quickly now shows exactly one real request in the DevTools Network tab, fired only once typing pauses — not one per keystroke, the honest cost Chapter 5 closed on. Clearing the field shows the placeholder text reappear, with no error banner and no broken pipeline; typing again afterward converts correctly, proving the 400-handling branch works exactly as verified above. Stopping the Express server mid-session and typing something new shows the real `"Couldn't reach the conversion service"` message —

and restarting Express and typing again recovers cleanly, without a page reload, proving `catchError`'s own real payoff one more time.

Hands-On Exercises

EXERCISE 1

Build the debounced `romajiInput$` pipeline inside `ConvertService` exactly as this chapter describes (`debounceTime`, `distinctUntilChanged`, `switchMap`, no `catchError` yet). Reproduce the race condition by sending 'su' then, 50ms later, 'sushi' — first through `mergeMap` (confirm 'su' incorrectly overwrites 'sushi'), then through `switchMap` (confirm only 'sushi' is ever applied).

 [View solution](#)

EXERCISE 2

Using the real running Express server from Chapter 5, send 'sushi', then '' (triggering the real 400), then 'sushi' again through the `switchMap` pipeline with no `catchError`. Confirm the second 'sushi' produces no output at all. Then add `catchError` inside the `switchMap` projector exactly as this chapter describes, and confirm the second 'sushi' now correctly goes through.

 [View solution](#)

EXERCISE 3

Build the final `ConvertService` (with `isLoading` and error signals) and the styled `RomajiInputComponent/KanaOutputComponent` exactly as this chapter describes. Confirm, using the real app: (a) a loading state briefly appears while typing, (b) clearing the input shows the placeholder text with no error, and (c) stopping the Express server and typing shows the real error message, recovering correctly once the server is restarted.

 [View solution](#)

CHAPTER 6 QUICK REFERENCE

- **Debounce lives in the service** — a Subject piped through `debounceTime(300)/distinctUntilChanged/switchMap`; `RomajiInputComponent`'s `onInput()` is unchanged, character for character, from Chapter 4
- **Real bug 1** — `mergeMap` runs concurrent requests and applies whichever resolves last, letting a slower, older response overwrite a faster, newer one; `switchMap` cancels the stale in-flight request instead, verified with a real out-of-order timing test
- **Real bug 2** — an unhandled HTTP error inside `switchMap` (including Chapter 5's own real 400 for an empty input) kills the entire outer subscription permanently; `catchError` placed inside the `switchMap` projector, returning `EMPTY`, keeps the pipeline alive
- **Real signals added** — `isLoading` and `error`, both read-only, both driven from inside the same pipeline; a 400 is handled as an expected empty state, not shown as an error
- **Real UI states, in priority order** — loading, then error, then result, then placeholder
- **Next chapter:** Deployment

CHAPTER 7: DEPLOYMENT

Romaji to Kana Converter: Angular & Express

Chapter 7 · Deployment

Getting this stack running as a real, standalone deployment starts from a genuine architectural advantage. This site's own React-based courses each end up with a hardcoded `localhost:4000` scattered across five or six separate files, one per component that ever calls the backend directly. This course never had that problem to begin with — `ConvertService` has been the *only* thing that talks to Express since Chapter 4, so there is exactly one hardcoded URL anywhere in this project: one line, inside one constructor.

A Single Source of Truth for the API's Own Base URL

Angular's own real mechanism for this is environment files — but current Angular doesn't scaffold them automatically the way older tutorials assume:

```
# from the Angular project root
ng generate environments
```

VERIFIED AGAINST ANGULAR'S OWN CURRENT DOCUMENTATION

Older Angular versions generated `src/environments/environment.ts` and `environment.prod.ts` automatically as part of `ng new` itself. Current Angular does not — by default no environment files exist and no file replacements are configured at all; running `ng generate environments` explicitly is what creates the `src/environments/` folder and wires the real replacement logic into `angular.json`. A project that skipped this step entirely simply has nowhere to put an environment-specific value like this one.

The command creates two files and one entry in `angular.json`:

```
// src/environments/environment.ts – the default, used for a real production build
export const environment = {
  apiUrl: '',
};
```

```
// src/environments/environment.development.ts – swapped in automatically for development
export const environment = {
  apiUrl: 'http://localhost:4000',
};
```

```
// angular.json – the fileReplacements block ng generate environments writes automatically
"configurations": {
  "development": {
    "fileReplacements": [
      {
        "replace": "src/environments/environment.ts",
        "with": "src/environments/environment.development.ts"
      }
    ]
  }
}
```

`ng serve` already runs under the `development` configuration by default, so `environment.development.ts` applies with no extra flag needed. A plain `ng build` uses the `production` configuration by default instead — no replacement happens, and `environment.ts`'s own empty `apiUrl` is what actually ships. Exactly the same empty-string-becomes-a-relative-path technique this site's own React-based courses use: a request built from `${apiUrl}/api/convert` with an empty base is just `/api/convert`, which resolves against whatever real origin the built app happens to be served from.

The One Real Code Change

`ConvertService`'s constructor is the only place this URL ever appears. One import, one string swapped for a template literal:

```
// src/app/convert.service.ts
import { environment } from '../environments/environment';
// ...

switchMap((romaji) =>
  this.http.post<ConvertResponse>(`${environment.apiUrl}/api/convert`, { romaji }).)
```

```
// ...catchError, unchanged from Chapter 6...
)
)
```

ONE FILE, NOT SIX

Compare this against a typical multi-component frontend, where the identical fix has to be repeated in every file that independently calls the backend. That was never a risk here — Chapters 4 through 6 already concentrated every real network call behind this one service, so hardening it for deployment costs exactly one line changed in exactly one place.

Building the Frontend for Production

```
# from the Angular project root (named "romaji-converter" since Chapter 1)
ng build
```

A REAL, EASY-TO-MISS DETAIL: THE OUTPUT NESTS ONE LEVEL DEEPER THAN EXPECTED

Current Angular's default build uses the esbuild-based application builder, which writes its output to `dist/romaji-converter/browser/` — not `dist/romaji-converter/` directly. The extra `browser/` segment exists to leave room for a separate server-side bundle alongside it on projects that use server-side rendering; this project doesn't, but the folder still gets created. Pointing a static file server at the wrong level here produces a very literal, very confusing "can't find index.html" failure.

Serving the Built Frontend From Express

```
// server.js – added after the /api/convert route
const path = require('path');

app.use(express.static(path.join(__dirname, '../romaji-converter/dist/romaji-converter/
```

Registration order doesn't matter here, for the same reason it didn't in this site's own React-based deployment chapters: `express.static()` only ever intercepts a request that matches a

real file it actually has — a request for `/api/convert` never matches anything inside that folder, so it falls straight through to Chapter 5's own route regardless of which middleware was registered first.

NO SPA-FALLBACK ROUTE NEEDED — FOR AN EVEN MORE FUNDAMENTAL REASON THAN USUAL

Most single-page-app deployment guides need one more piece here: a catch-all route serving `index.html` for any URL that isn't a real static file, so refreshing the browser on a client-side route doesn't 404. This site's own **Personal Catalogue: React, Express & MongoDB** course reached that same conclusion because its own app never happened to add a client-side route. This project's reason is stronger still — Chapter 1 explicitly skipped Angular's router entirely (`ng new` with no routing flag), so there is no client-side route that could ever exist to need a fallback for in the first place. A single request for `/` already resolves to `index.html` automatically, and that's genuinely the only page this app has.

Tightening CORS for Production

Chapter 1's `app.use(cors())` allows a request from any origin. Once Express serves both the API and the built frontend from the same process, the app's own requests are same-origin already and don't strictly need CORS to succeed — restricting it explicitly is still cheap, real defense in depth:

```
// server.js
app.use(cors({
  origin: process.env.ALLOWED_ORIGIN || 'http://localhost:4200',
}));
```

SET ALLOWED_ORIGIN ONCE A REAL DOMAIN EXISTS

In production, `ALLOWED_ORIGIN` should point at the real deployed domain, not the local dev-server default. Chapter 8's own capstone may end up mounting this app at a path under the user's existing Astro site's own domain rather than a separate one entirely — whichever it turns out to be, this is the one setting that needs updating to match.

Keeping the Server Running: PM2

```
# installed once, globally, on the server
npm install -g pm2

# start the app under PM2's own supervision
pm2 start server.js --name romaji-converter-api

# persist the process list across a real server reboot
pm2 save
pm2 startup
```

`pm2 logs romaji-converter-api` tails real, live output; `pm2 restart romaji-converter-api` picks up a fresh `ng build` or a server-side change without a full stop-and-start cycle.

A Reverse Proxy for a Real Domain and HTTPS

Node's own process doesn't need to be exposed to the internet directly. A real nginx reverse proxy handles the public-facing domain and — following this site's own dedicated **HTTPS/TLS Fundamentals** course for the actual certificate setup — TLS termination:

```
# /etc/nginx/sites-available/romaji-converter
server {
    listen 80;
    server_name yourdomain.com;

    location / {
        proxy_pass http://localhost:4000;
        proxy_set_header Host $host;
        proxy_set_header X-Real-IP $remote_addr;
    }
}
```

Trying It

```
# build the frontend
cd romaji-converter && ng build && cd ..
```

```
# start the backend, now also serving the built frontend  
cd romaji-converter-api && node server.js
```

Visiting `http://localhost:4000` directly — not port 4200, which no longer has anything running on it — should load the real, styled converter, with every request now round-tripping through a single Node process serving both halves of the app.

Hands-On Exercises

EXERCISE 1

Run `ng generate environments` in a fresh copy of the project (or a scratch project if you don't want to touch the real one yet), and inspect the real generated files and `angular.json` changes. Confirm `environment.ts` and `environment.development.ts` exist with the shape this chapter describes, and confirm the `fileReplacements` block was added automatically under the development configuration.

[!\[\]\(ddb2fd935b66087ad7fb940636295ac7_img.jpg\) View solution](#)

EXERCISE 2

Run `ng build` for real and inspect the actual output folder structure. Confirm `index.html` and the hashed JS/CSS bundles live under `dist/romaji-converter/browser/`, not directly under `dist/romaji-converter/`. Then deliberately point `express.static()` at the wrong (too-shallow) path and confirm what error actually results when you request the page.

[!\[\]\(689be5851cb0a68cc4e819abe4419ae8_img.jpg\) View solution](#)

EXERCISE 3

Wire `ConvertService` to `environment.apiUrl` exactly as this chapter describes, build the frontend, and serve it from Express at the correct path. Confirm the app works correctly when visited at `http://localhost:4000` (a real production-shaped build, `apiUrl` empty, requests resolving as relative paths) and still works correctly under `ng serve` at `http://localhost:4200` (`apiUrl` pointing at `localhost:4000` explicitly).

 [View solution](#)

CHAPTER 7 QUICK REFERENCE

- **One hardcoded URL, one fix** — since ConvertService has been the sole owner of every backend call since Chapter 4, deployment needs exactly one code change, in exactly one file
- **ng generate environments** — not auto-scaffolded by ng new in current Angular; creates environment.ts (production default) and environment.development.ts (swapped in automatically by ng serve), plus the real fileReplacements entry in angular.json
- **Empty apiBaseUrl in production** — the same empty-string-becomes-relative-path technique used across this site's own other deployment chapters
- **A real, verified build-output detail** — ng build's current esbuild-based builder writes to dist/<project>/browser/, one level deeper than dist/<project>/ alone
- **No SPA fallback needed** — this app was never given a router at all, so there's no client-side route to ever need one for
- **PM2 + nginx** — the same production process-management and reverse-proxy pattern used throughout this site's own Node-backed courses
- **Next chapter:** Capstone — integrating this tool into the existing Astro-based site

CHAPTER 8: CAPSTONE — INTEGRATING THIS INTO THE EXISTING ASTRO SITE

Romaji to Kana Converter: Angular & Express

Chapter 8 · Capstone: Integrating This Into the Existing Astro Site

Chapter 7 got the whole stack running as a real, standalone deployment. This closing chapter does what Chapter 1 always meant it to do: put it somewhere real, at a real path under the user's own existing Astro-based site, and finish the one promise that made this course's own reason for existing concrete from the very first page — tying it directly into the Japanese-language lessons and kanji pages already live there.

Three Real Ways to Mount This Behind an Existing Site

OPTION	WHAT IT INVOLVES	TRADEOFF
Subdomain (<code>romaji.existing-site.com</code>)	A separate DNS record and its own nginx server block pointing straight at the unified Express app from Chapter 7	Simplest possible nginx config; reads as a separate site rather than one integrated tool
Path-mounted reverse proxy (<code>existing-site.com/romaji/</code>)	One nginx location block on the existing site's own domain, proxying that one path to the same Express app	Feels like part of the same site; needs two real, verified fixes this chapter builds — covered below
JSON-API + Astro-frontend rewrite	Keep only Chapter 5's Express route; rebuild the small UI as a new Astro page consuming it directly	The deepest real integration, and real rework of a UI that already works well — hard to justify for a tool this size

The path-mounted option is built out in full here, matching every other Personal Multi-Technology Projects course on this site that reached this same decision.

A Real Problem: Built Assets Don't Know They're Not at the Root

Chapter 7's own `ng build` produces an `index.html` whose script and style references assume the app is served from the domain root — correct for that chapter's own standalone

deployment, and wrong the moment this app is mounted at `/romaji/` on someone else's domain.

VERIFIED AGAINST ANGULAR'S OWN CURRENT DOCUMENTATION

The Angular CLI's own `--base-href` flag exists specifically for this: it rewrites the `<base href>` tag Angular injects into `index.html`, and the built asset references along with it, to reflect a real sub-path deployment rather than a root one — the direct Angular equivalent of Vite's own `base` config option, used for exactly this same purpose in this site's own React-based sibling courses.

```
# from the Angular project root
ng build --base-href /romaji/
```

A Second Real Problem: API Calls Are Root-Relative, Not Site-Relative

Chapter 7's own `environment.ts` leaves `apiBaseUrl` empty, so every request becomes a plain path like `/api/convert` — correct at the domain root, still wrong here for a genuinely different reason than the assets above. `/api/convert` is an absolute path; it always resolves against the domain's own root, regardless of which page made the request. A request from `existing-site.com/romaji/` would still be sent to `existing-site.com/api/convert`, not `existing-site.com/romaji/api/convert`, where this app's own API actually needs to be reached once mounted at a sub-path.

Editing `environment.ts` directly would fix this deployment but quietly break Chapter 7's own root-deployment story for anyone who builds this project without mounting it under a sub-path. A third, small environment file keeps both working:

```
// src/environments/environment.mounted.ts – new for this specific deployment
export const environment = {
  apiBaseUrl: '/romaji',
};
```

```
// angular.json – a new, minimal "mounted" configuration alongside "production" and "development"
"configurations": {
  "production": { /* unchanged from Chapter 7 */ },
```

```
"development": { /* unchanged from Chapter 7 */ },
"mounted": {
  "fileReplacements": [
    {
      "replace": "src/environments/environment.ts",
      "with": "src/environments/environment.mounted.ts"
    }
  ]
}
}
```

VERIFIED: NAMED CONFIGURATIONS COMPOSE WITH A COMMA

Angular's own current documentation confirms multiple named configurations can be combined

documentation's own example — parsed left to right, with a later configuration's own settings winning if two configurations happen to touch the same option. There's no separate `extends` keyword for one configuration to inherit from another; composing two small, focused configurations together at build time is the real, documented mechanism instead.

```
# the real, final build command for this specific deployment
ng build --configuration production,mounted --base-href /romaji/
```

TWO GENUINELY DIFFERENT FIXES FOR TWO GENUINELY DIFFERENT THINGS

`--base-href` is a build-time setting controlling where the compiled JavaScript and CSS *files* are referenced from. `environment.apiUrl` is a value baked into the bundle at the same build, but controlling where this app's own *HTTP requests* are sent. Both happen to need the identical `/romaji` prefix for this deployment, but setting only one leaves either a broken page (missing assets) or a broken app (every conversion request failing) — never both problems solved by a single change.

Building Out the Reverse Proxy

One location block on the existing site's own nginx config, added alongside whatever configuration already serves the rest of that site:

```
# /etc/nginx/sites-available/existing-site (excerpt)
server {
    listen 443 ssl;
    server_name existing-site.com;

    # ...the existing Astro site's own location blocks, unchanged...

    location /romaji/ {
        proxy_pass http://localhost:4000/;
        proxy_set_header Host $host;
        proxy_set_header X-Real-IP $remote_addr;
    }
}
```

The trailing slash on both `location /romaji/` and `proxy_pass http://localhost:4000/;` is what makes nginx strip the `/romaji` prefix before forwarding — a request for `/romaji/api/convert` reaches Express as exactly `/api/convert`, the same route Chapter 5 already built with no changes needed on the Express side. The same stripping applies to every static asset request, which reaches Express's own `express.static()` middleware from Chapter 7 completely unaware it's being served from anywhere but its own root.

With everything mounted this way, the browser only ever sees one real domain — Chapter 7's own `ALLOWED_ORIGIN` CORS setting can simply be set to `https://existing-site.com`, and every request genuinely is same-origin now, exactly the scenario that chapter's own warn-box anticipated without yet knowing which real shape it would take.

Linking It Where It's Actually Useful

Chapter 1 named the real reason this course exists — this tool's own most immediate value is sitting right next to the Japanese-language lessons and the kanji tiles page, not off on its own. With a real, working `/romaji/` URL now live, the last piece is a real link pointing at it from exactly those pages — the concrete payoff of a promise made eight chapters ago:

```
<!-- illustrative addition to kanji_tiles cms_content.html and the japanese_lesson_* pa
<a href="/romaji/" class="tool-link" target="_blank" rel="noopener">
```

abc Convert romaji ↔ kana

ILLUSTRATIVE, NOT APPLIED SITE-WIDE HERE

Adding this link to every real Japanese-lesson and kanji page is genuinely a content-editing task across many separate files, not something this one course chapter does on its own — the same

"concrete plan, real next step" framing this site's own sibling capstones already use for their own deferred features. The snippet above is the exact real markup that plan resolves to, verified to work against the live `/romaji/` URL this chapter just finished standing up.

Chapter Attribution

PIECE OF THE FINISHED APP	BUILT IN
The full gojūon/dakuten/yōon romaji-to-kana mapping table	Chapter 2
Long vowels, sokuon, and the real は/へ/を particle exceptions	Chapter 3
ConvertService, RomajiInputComponent, KanaOutputComponent — the shared-service architecture	Chapter 4
The real Express /api/convert route, the ported plain-JS conversion engine	Chapter 5
Debounced, race-safe, error-resilient conversion pipeline; the real styled UI	Chapter 6
Single-process production deployment; environment.apiUrl; PM2; nginx	Chapter 7
Path-mounted reverse proxy, --base-href, the "mounted" configuration, and a real link into the site's own Japanese content	Chapter 8 (this chapter)

What This Course Deliberately Doesn't Cover

- **Kanji conversion** — named out of scope in Chapter 1 for a real reason: it's a genuinely harder, dictionary-and-context-dependent problem (the same kana can map to several different kanji depending on meaning), deliberately left as its own real future project rather than bolted on here.

- **A persisted conversion history** — every result lives only in the current page's own signals; nothing is saved across a reload.
- **UPC/EAN-style batch or file-based conversion** — this tool converts one typed phrase at a time, matching the actual, narrow use case Chapter 1 scoped it around.

Hands-On Exercises

EXERCISE 1

Create `environment.mounted.ts` and the "mounted" configuration exactly as this chapter describes, then run `ng build --configuration production,mounted --base-href /romaji/`. Inspect the real built `index.html` and confirm its asset references include the `/romaji` prefix, and inspect the built JavaScript bundle to confirm the string `'/romaji'` appears in place of Chapter 7's empty `apiBaseUrl`.

 [View solution](#)

EXERCISE 2

Configure the nginx location `/romaji/` block against a real running instance of the app (a second local nginx config is fine for testing), and confirm both a page load and a real `/api/convert` request succeed through the proxy, with `server.js` completely unchanged from Chapter 7.

 [View solution](#)

EXERCISE 3

Build with only `--base-href /romaji/` set, using just the default "production" configuration (skip "mounted" entirely, leaving `apiBaseUrl` empty). Serve it through the same nginx proxy from Exercise 2 and confirm the page itself loads correctly while every conversion attempt fails — demonstrating why this chapter's own warn-box treats the two fixes as genuinely separate.

 [View solution](#)

CHAPTER 8 QUICK REFERENCE — COURSE COMPLETE

- **Three mounting options** — subdomain, path-mounted reverse proxy (built here), or a full Astro-frontend rewrite
- **--base-href /romaji/** — Angular's own verified flag for correcting built asset references, the real equivalent of Vite's base option
- **A new "mounted" configuration** — a minimal fileReplacements override, composed with "production" via Angular's own real, verified comma-separated multi-configuration syntax
- **environment.mounted.ts** — sets apiBaseUrl to /romaji without touching Chapter 7's own generic, empty default
- **nginx's trailing-slash prefix-stripping** — lets server.js stay completely unchanged from Chapter 7
- **The real payoff** — a live link from the existing kanji tiles page and Japanese-language lessons into a genuinely working /romaji/ tool, the exact promise Chapter 1 opened with
- **This closes the course** — 8/8 chapters, matching the shared Personal Multi-Technology Projects romaji-to-kana spec from Chapter 1