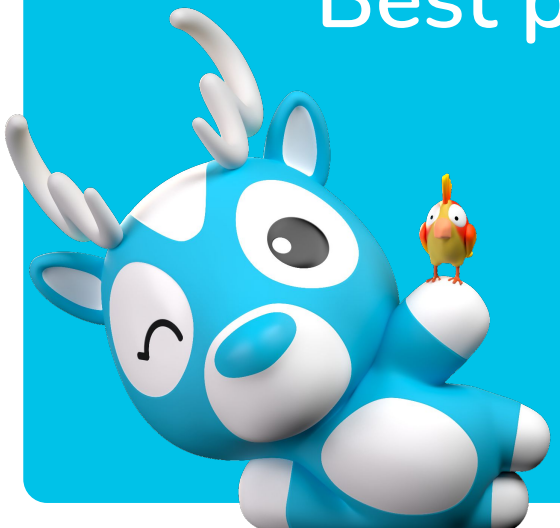
Wolt Frontend
Best practices & How to Nail the Technical Interview



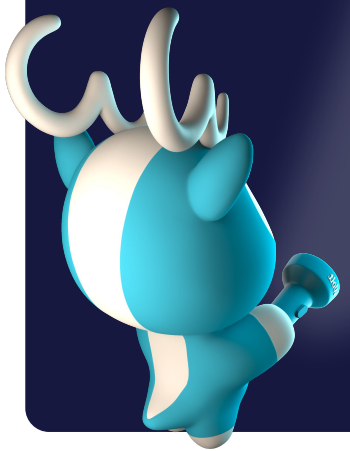
01.2026

Wolt

Amedeo Majer

- 👨‍🍳 Worked at Fafa’s 2014-2021
- 🐝 Did my studies at Hive 2021-2023 – Hiver 3
- 🏫 Work as dev bocal at Hive 2023-2025
- 💻 Junior web developer at Wolt 2025

What makes frontend code good?



Headline 1. Preferably max. two lines of text.

- **Clarity:** easy to read, easy to explain.
- **Accessible:** works for everyone, not just mouse + screen users.
- **Predictable:** behaves the same way every time.
- **Testable:** important logic can be verified in isolation.
- **Maintainable:** small changes do not cause surprises.

Accessibility & Semantics

If it only works with a mouse and perfect vision, it is not finished yet.

Accessibility & Semantics (a11y)

Good Practices:

- Use **semantic HTML** (`button`, `label`, `form`, `nav, table`)
- Prefer `input[type="text"]` + `inputmode` for numeric input instead of `input[type="number"]`
- Use and learn aria attributes
- Provide **clear instructions** for the expected input format

Why:

- Screen readers, keyboards, and assistive tech rely on semantics
- Better accessibility usually means better UX for everyone
- Interviewers notice when you care about this

UX, UI & Polish

Good UX is about removing friction, not adding decoration.

UX, UI & Polish

Good Practices:

- Implement **loading states** (spinners, skeletons, disabled actions)
- Format **numbers, currency, and dates** using the user's locale
- Support **light and dark mode**
- Add light visual polish (spacing, hierarchy, subtle animations)
- Use brand colors **tastefully**, not excessively

Why:

- Users trust interfaces that feel responsive and consistent
- Clear feedback prevents repeated actions and errors
- Small polish details signal care and professionalism

Separation of Concerns

Structure your code so each responsibility is clear and isolated.

Separation of Concerns

Separate these concerns:

- Input validation
- Submit / action logic
- Communication with APIs
- Parsing and validating API responses
- Business logic
- Output formatting for display

Why:

- Users trust interfaces that feel responsive and consistent
- Clear feedback prevents repeated actions and errors
- Small polish details signal care and professionalism

Simple Frontend Structure

Component

Handles input and rendering

- Reads user input
- Manages UI state
- Renders UI based on data
- No business rules

Pure functions

Calculations & decisions

- No side effects
- Same input → same output
- Easy to test
- Contains business logic

Examples:

- `isEmailValid(email)`
- `calculateTotal(items)`

Service

External communication

- API calls
- Data fetching
- Error handling
- No UI logic

Examples:

- `fetchUsers()`
- `submitForm(data)`
- `loadProfile(id)`

Testing

Testing is easier when your code is structured well.

Testing

Good practices:

- Prefer testing **small, pure functions**
- Focus tests on **business logic** and **data parsing**
- Test edge cases and invalid inputs
- Keep tests fast and deterministic

Avoid:

- Large tests that require rendering everything
- Heavy mocking of UI and network layers
- Testing implementation details instead of behavior

Why:

- Tests give confidence to refactor
- Failures are easier to understand and fix
- Interviewers care more about *what* you test than the testing library

```ts
async function processOrder(orderId: string) {
  const response = await fetch(`/api/orders/${orderId}`)
  const order = await response.json()

  let total = 0
  for (const item of order.items) {
    total += item.price * item.quantity
  }

  if (order.discountCode === "SAVE10") {
    total = total * 0.9
  }

  if (order.country === "IT") {
    total = total * 1.22
  }

  return Math.round(total * 100) / 100
}
```

```ts
global.fetch = vi.fn(() =>
  Promise.resolve({
    json: () =>
      Promise.resolve({
        items: [{ price: 10, quantity: 2 }],
        discountCode: "SAVE10",
        country: "IT",
      }),
  } as any)
)

it("calculates total price", async () => {
  const total = await processOrder("123")
  expect(total).toBe(21.96)
})
```

Big mocking function

If the test fails we do not know where it happened.

```ts
type Order = {
  items: { price: number; quantity: number }[]
  discountCode?: string
  country: string
}

function calculateSubtotal(items: Order["items"]) {
  return items.reduce(
    (sum, item) => sum + item.price * item.quantity,
    0
  )
}

function applyDiscount(total: number, code?: string) {
  return code === "SAVE10" ? total * 0.9 : total
}

function applyTax(total: number, country: string) {
  if (country === "IT") return total * 1.22
  return total
}

function roundMoney(amount: number) {
  return Math.round(amount * 100) / 100
}
```

```ts
it("applies Italian tax", () => {
  expect(applyTax(10, "IT")).toBe(12.2)
})

it("applies discount", () => {
  expect(applyDiscount(100, "SAVE10")).toBe(90)
})

it("calculates subtotal", () => {
  expect(
    calculateSubtotal([{ price: 5, quantity: 2 }])
  ).toBe(10)
})
```

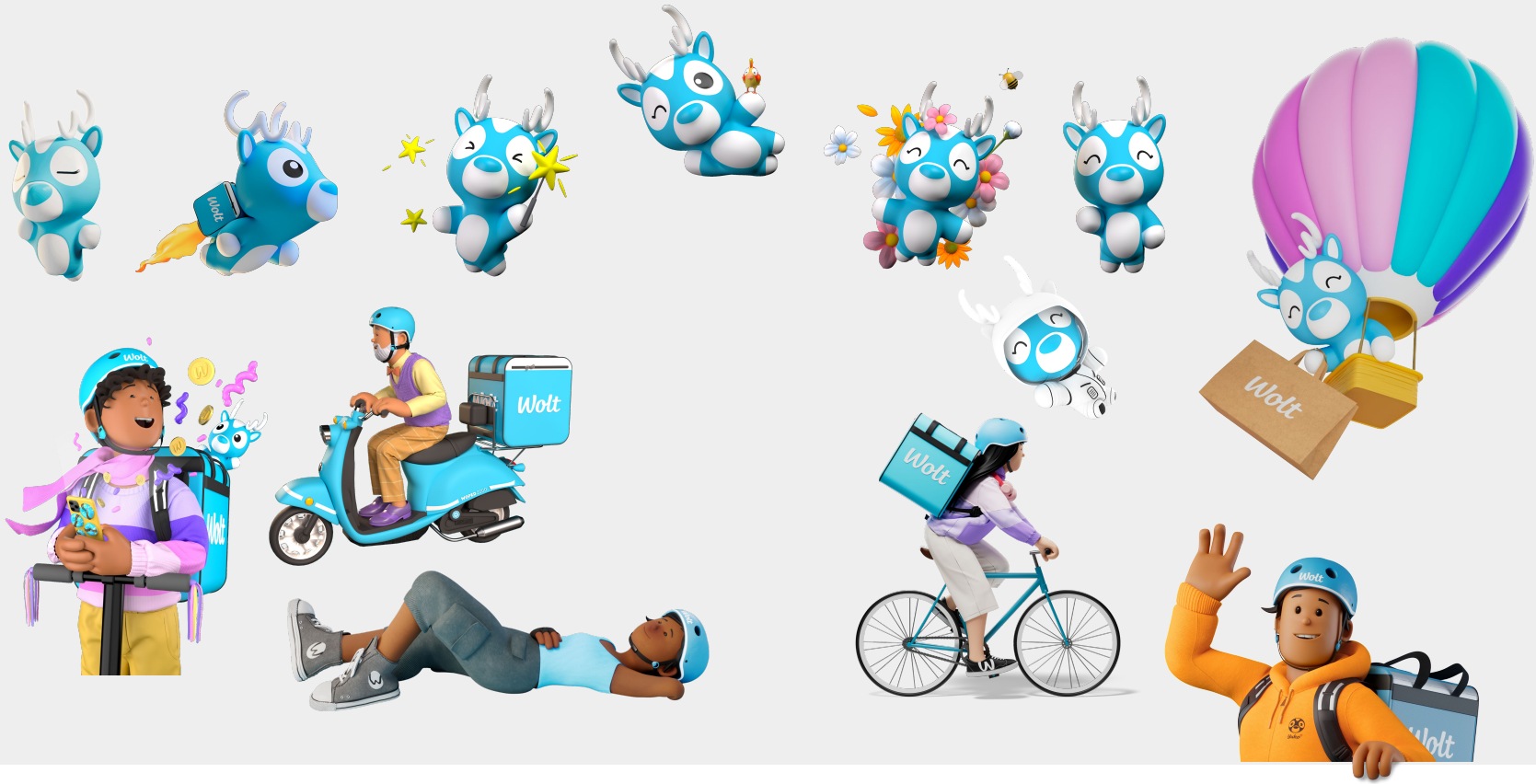
Small single responsibility functions can be easily tested, without the need for mocking.

If something goes wrong, we know exactly where.

That’s it!

Wolt

Copy/paste & scale Wolt City characters to your slides



Copy/paste & scale Wolt City characters to your slides

Copy/paste & scale Wolt City characters to your slides

Copy/paste & scale icons to your slides

More found at wolt.bynder.com