# CS 4530: Fundamentals of Software Engineering
# Module 3.1: Trusting TypeScript (or not!)

Adeel Bhutta, Rob Simmons, and Mitch Wand

Khoury College of Computer Sciences

## When Have I Written Enough Tests?

## When Have I Written Enough Tests?

- **When I’ve tested the valid inputs**
- When I’ve tested all the code
- When the tests will catch bugs

## Learning Goals for this Lesson

At the end of this lesson, you should be able to

- Explain how TypeScript types and documented preconditions influence what tests you need to write
- Explain the difference between the any vs unknown types in TypeScript
- Understand the structure of a simple Express server incorporating Zod validation

## What Inputs Should We Test?

What input values do I need to test this function on?

- Edge cases (definitely 0)
- Probably 1 and some larger number? But most numbers > 1 are kind of interchangeable.
- What about -3? 1.4? NaN? `null`? `{ lol: 'owned' }` ?

```
/** Returns an array that repeats "hello"
 * @param numHellos - number of “hello”s to return, must be an integer >= 0
 */
function helloNTimes(numHellos: number): string[] {
  const arr: string[] = [];
  for (let i = numHellos; i !== 0; i--) { arr.push("hello"); }
  return arr;
}
```

# For Unit Testing, Tests Inputs Should Respect a Function’s Contracts

- **Unit Tests:** testing a single function in isolation
  - Unit testing only needs to give a function tests that respect the functions preconditions: no need to test -3 or 1.4
- **Integration Tests** test how parts of a program work together: that’s where we ensure *other* functions respect our function’s contracts.

```
/** Returns an array that repeats "hello"
 * @param numHellos - number of “hello”s to return, must be an integer >= 0
 */
function helloNTimes(numHellos: number): string[] {
  const arr: string[] = [];
  for (let i = numHellos; i !== 0; i--) { arr.push("hello"); }
  return arr;
}
```

## What Trusting Contracts Looks Like

```
/**
 * Adds a message to a chat, updating the chat
 *
 * @param chatId - Ostensible chat id
 * @param user - Authenticated user
 * @param messageId - Valid message id
 * @returns the updated chat info object
 * @throws if the chat id is not valid
 */
export function addMessageToChat(
  chatId: string,
  user: UserWithId,
  messageId: string
): ChatInfo {
```

# TypeScript Types Help With (Some) Contracts

- TypeScript is helpful, but it's obviously only providing help in making sure that the function gets `3` and not `“three”`. It's not going to help with `3` versus `3.1` or `-8` or `NaN`.

```
/** Returns an array that repeats "hello"
 * @param numHellos - number of “hello”s to return, must be an integer >= 0
 */
function helloNTimes(numHellos: number): string[] {
  const arr: string[] = [];
  for (let i = numHellos; i !== 0; i--) { arr.push("hello"); }
  return arr;
}
```

## TypeScript Types Are Easily Circumvented (1)

- In a language like Java, we’d need to worry that another function could call `helloNTimes` with -3: calling the function with a string or `null` is a compiler error.
- That’s not true in TypeScript, and that can be surprising.

```
/** Returns an array that repeats "hello"
 * @param numHellos - number of “hello”s to return, must be an integer >= 0
 */
function helloNTimes(numHellos: number): string[] {
  const arr: string[] = [];
  for (let i = numHellos; i !== 0; i--) { arr.push("hello"); }
  return arr;
}
```

## TypeScript Types Are Easily Circumvented (2)

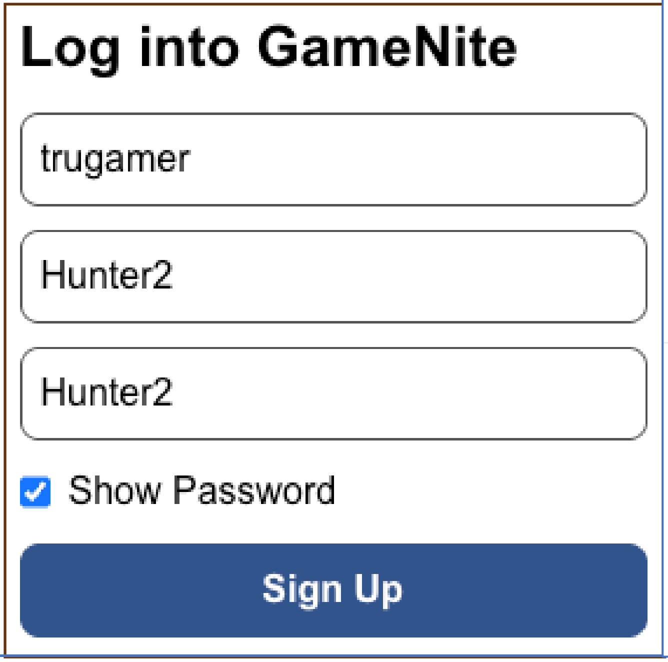
- In a language like Java, we'd need to worry that another function could call `helloNTimes` with -3: calling the function with a string or `null` is a compiler error.
- That's not true in TypeScript, and that can be surprising.
- TypeScript will happily accept the following as a well-typed expression:

```
helloNTimes({ lol: 'owned ' } as unknown as number)
```

- They do seem to make it less likely you'll screw up *accidentally*…, and ESLint + TypeScript work together to do even more

## Where Might An Untrusted Input Come From?

Any input given to a web app can also be given by other means…



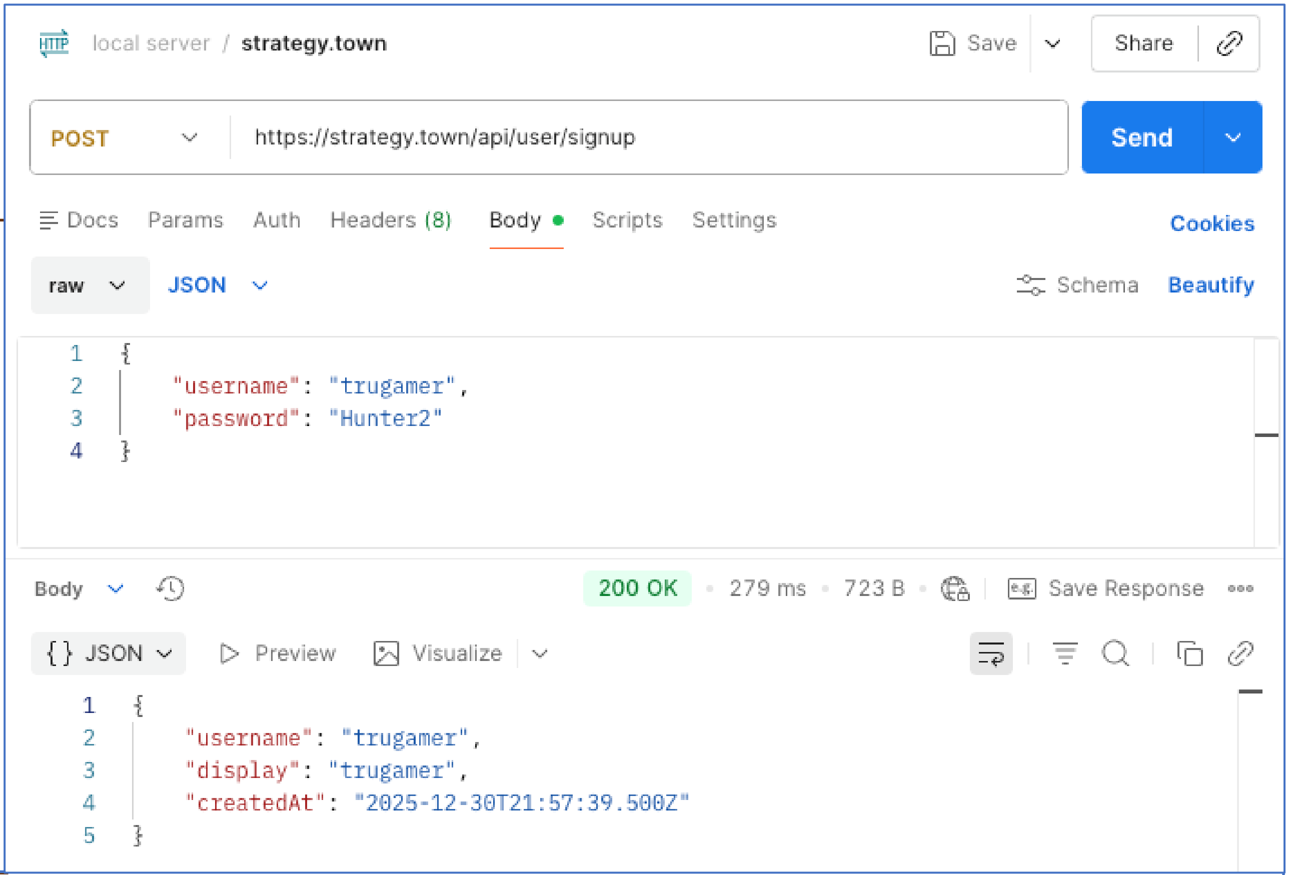
Log into GameNite

trugamer

Hunter2

Hunter2

☑ Show Password



POST https://strategy.town/api/user/signup

```
{
    "username": "trugamer",
    "password": "Hunter2"
}
```

Body  200 OK • 279 ms • 723 B

```
{
    "username": "trugamer",
    "display": "trugamer",
    "createdAt": "2025-12-30T21:57:39.500Z"
}
```

```
curl https://strategy.town/api/user/signup -H 'Content-Type: application/json' \
  --data '{ "username": "trugamer", "password": "Hunter2" }'
```

## Untrusted Inputs Should Have Type unknown

- The appropriate TypeScript type for an unknown value is unknown

```
function lookAtMe(input: unknown) {
  console.log(input.toUpperCase());
  if (typeof input === "string") {
    console.log(input.toUpperCase());
  }
}
```

TypeScript error here!

it's ok here!

- If you use the any type instead, TypeScript will just say "ok, I guess you know what you're doing"

## How Can unknown Inputs Be Used Safely?

This can get complicated fast…

```
export type Auth = { username: string, password: string }

function useAuth(x: unknown) {

  if (
    (typeof x === 'object' && x !== null) &&
    ('username' in x && typeof x.username === 'string’) &&
    ('password' in x && typeof x.password === 'string’)
  ) {
    const auth: Auth = { username: x.username, password: x.password };
    // write the code you care about here!
  }
}
```

# Libraries Make Checking Types Easier

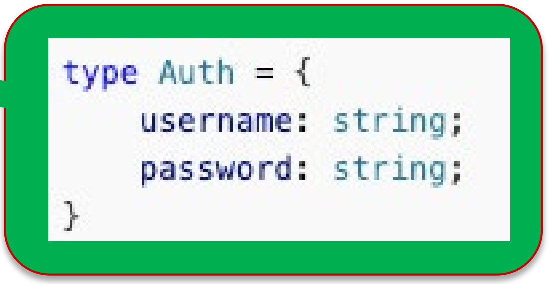
*Zod* is a library that makes checking structure less tedious & error-prone.

```
import { z } from 'zod’;
const zAuth = z.object({ username: z.string(), password: z.string() });
export type Auth = z.infer<typeof zAuth>;

function useAuth(x: unknown) {
  const parseResult = zAuth.safeParse(x);
  if (parseResult.success) {



    const auth: Auth = parseResult.data;
    // write the code you care about here!
  }
}
```



```
type Auth = {
    username: string;
    password: string;
}
```

# Some Libraries Use any: Common But Dangerous

```
import express from 'express';
const app = express();
app.use(express.json());

type Auth = { username: string; password: string };
app.post('/', (req, res) => {
  const auth: Auth = req.body;


  if (auth.password !== 'secret') {
    res.status(403).send({ error: 'Wrong password' });
  } else {
    res.send({ message: `WELCOME,${auth.username.toUpperCase()}` });
  }
});
app.listen(8000, () => console.log(`Listening on port 8000`));
```

This has type “any” 😭

# Improving This Web Server With Zod

```
import { z } from 'zod';
import express from 'express';
const app = express();
app.use(express.json());

const zAuth = z.object({ username: z.string(); password: z.string() });
app.post('/', (req, res) => {
  const auth = zAuth.safeParse(req.body);
  if (auth.error) {
    res.status(400).send({ error: 'Unexpected message' });
  } else if (auth.data.password !== 'secret') {
    res.status(403).send({ error: 'Wrong password' });
  } else {
    res.send({ message: `WELCOME,${auth.data.username.toUpperCase()}` });
  }
});
app.listen(8000, () => console.log(`Listening on port 8000`));
```

# Zod Can Check Conditions Even Finer than TypeScript Types

```
const zHelloInput = z.int().gte(0);

/** Returns an array that repeats "hello"
 * @param numHellos - number of times to say “hello”
 * @throws if the input is not an integer >= 0
 */
function helloNTimes(numHellos: unknown): string[] {
  const parseResult = zHelloInput.safeParse(numHellos);
  if (!parseResult.success) throw new Error("Invalid input");
  const arr: string[] = [];
  for (let i = parseResult.data; i !== 0; i--) { arr.push("hello"); }
  return arr;
}
```

## Review

- One view of TypeScript is that it’s a handy way of documenting, and *imperfectly* checking, the contracts (preconditions and postconditions) of your code
- Do you need to test inputs that violate your contracts? It depends!
- You can never trust that the input to a web server will obey any sort of contract — important to test!