# 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

# 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

# What Inputs Should We Test?

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

```
/**
 * Prints "hello" repeatedly
 *
 * @param numHellos - number of times to print “hello”,
 *   must be an integer >= 0
 */
function helloNTimes(numHellos: number) {
  for (let i = numHellos; i !== 0; i--) {
    console.log('hello');
  }
}
```

# 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.
  - If we want to sound fancy, we can call these “equivalence classes of inputs.”
- What about -3? 1.4? NaN? null? { lol: 'owned' } ?

```
/**
 * Prints "hello" repeatedly
 *
 * @param numHellos - number of times to print “hello”,
 *   must be an integer >= 0
 */
function helloNTimes(numHellos: number)
```

# 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 different parts of a program work together, and that's where we're concerned with ensuring that *other* parts of the program respect our function's contracts.

```
/**
 * Prints "hello" repeatedly
 *
 * @param numHellos - number of times to print “hello”,
 *   must be an integer >= 0
 */
function helloNTimes(numHellos: number)
```

# 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 different parts of a program work together, and that's where we're concerned with ensuring that *other* parts of the program respect our function's contracts.

```
/**
 * Prints "hello" repeatedly
 *
 * @param numHellos - number of times to print “hello”,
 *   must be an integer >= 0
 */
function helloNTimes(numHellos: number)
```

# TypeScript Types Are Easily Circumvented (1)

- In a language like Java, we'd need to worry that another function could call helloNTimes with -3, but not with a string or null.
- That's not true in TypeScript!

```
/**
 * Prints "hello" repeatedly
 *
 * @param numHellos - number of times to print “hello”,
 *   must be an integer >= 0
 */
function helloNTimes(numHellos: number)
```

# TypeScript Types Are Easily Circumvented (2)

- In a language like Java, we'd need to worry that another function could call helloNTimes with -3, but not with a string.
- That's not true in TypeScript!
- TypeScript types are, at the end of the day, no better than preconditions mentioned in comments.

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

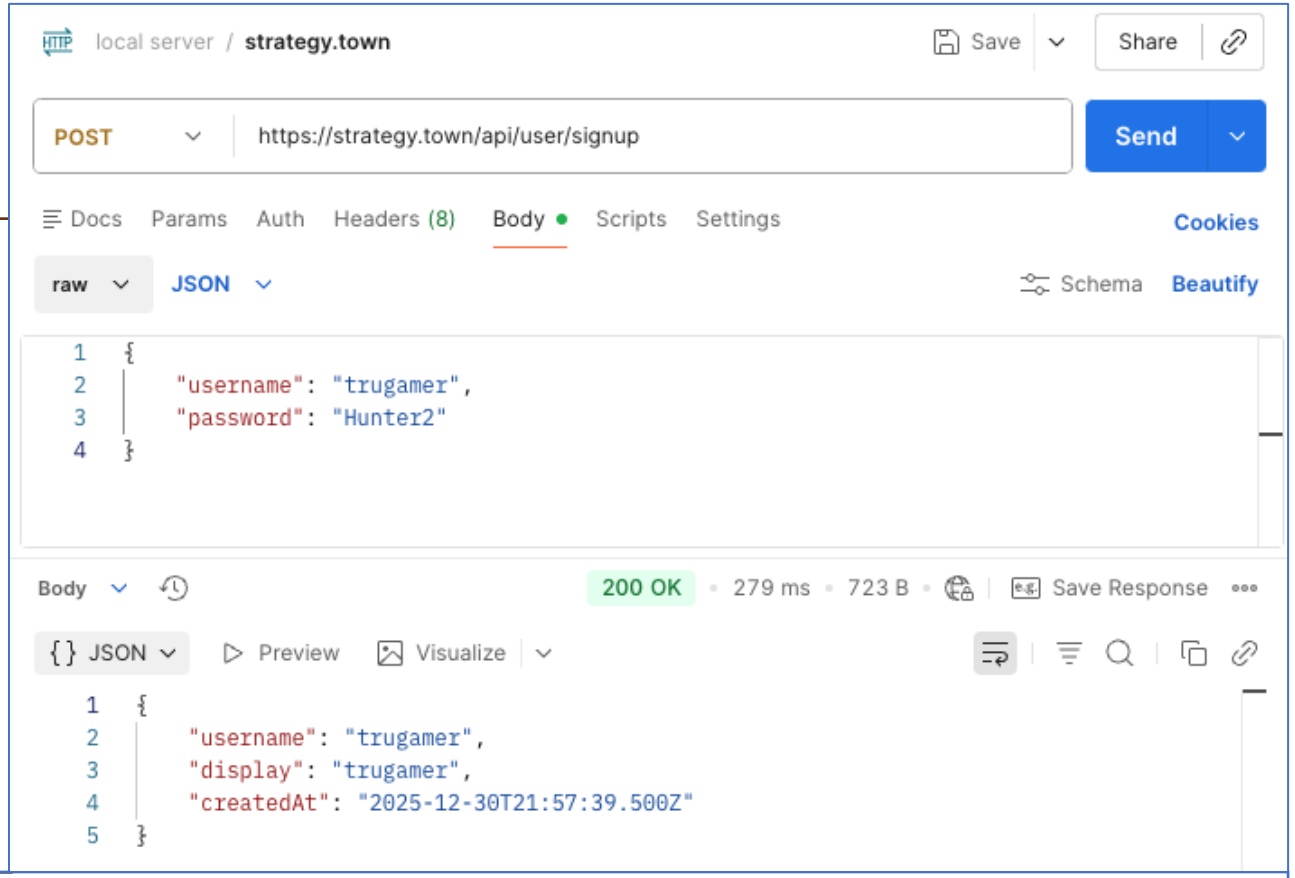
- They do seem to make it less likely you'll screw up *accidentally…*

# 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 {
```

# Untrusted Inputs

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

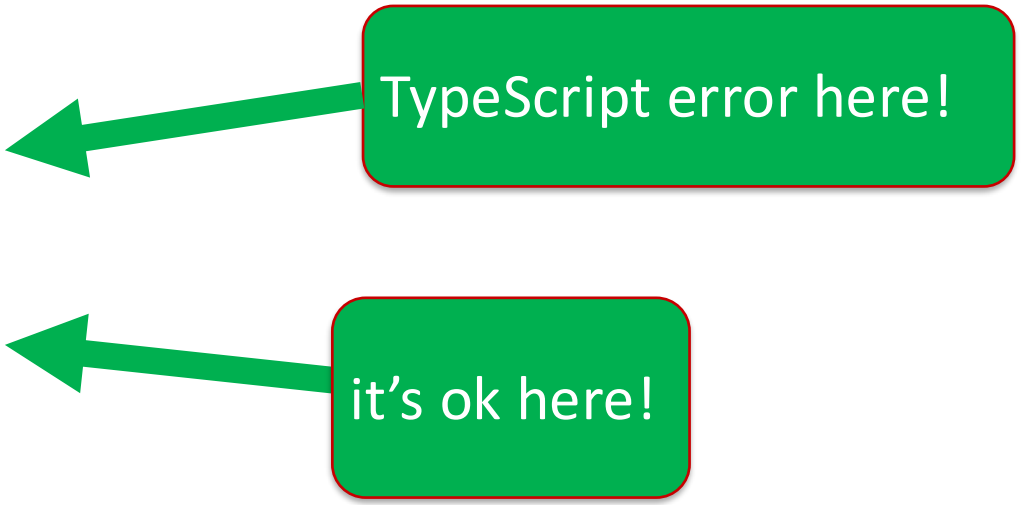




```
curl https://strategy.town/api/user/signup -H 'Content-Type: application/json' \
  --data '{ "username": "trugamer", "password": "Hunter2" }'
```

# Untrusted Inputs should be 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());
  }
}
```




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

# Untrusted Inputs Should be “unknown”

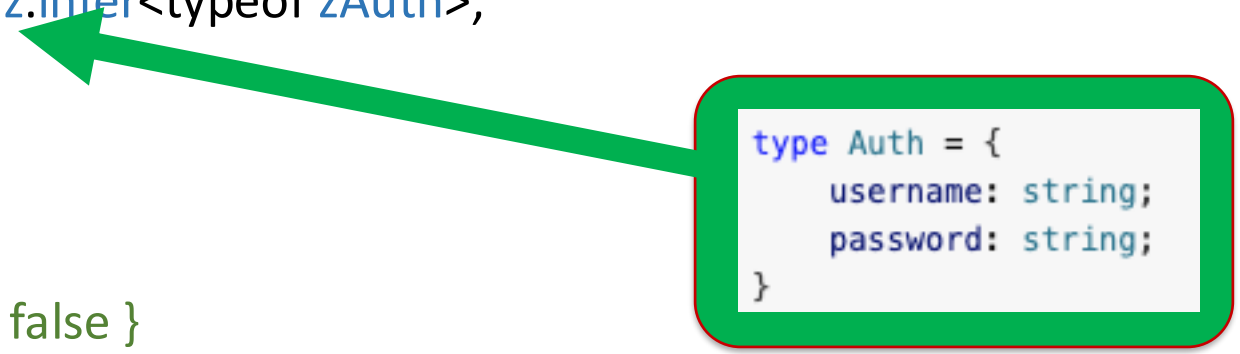
This can get complicated fast…

```
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 testing the structure of inputs less tedious and error-prone.

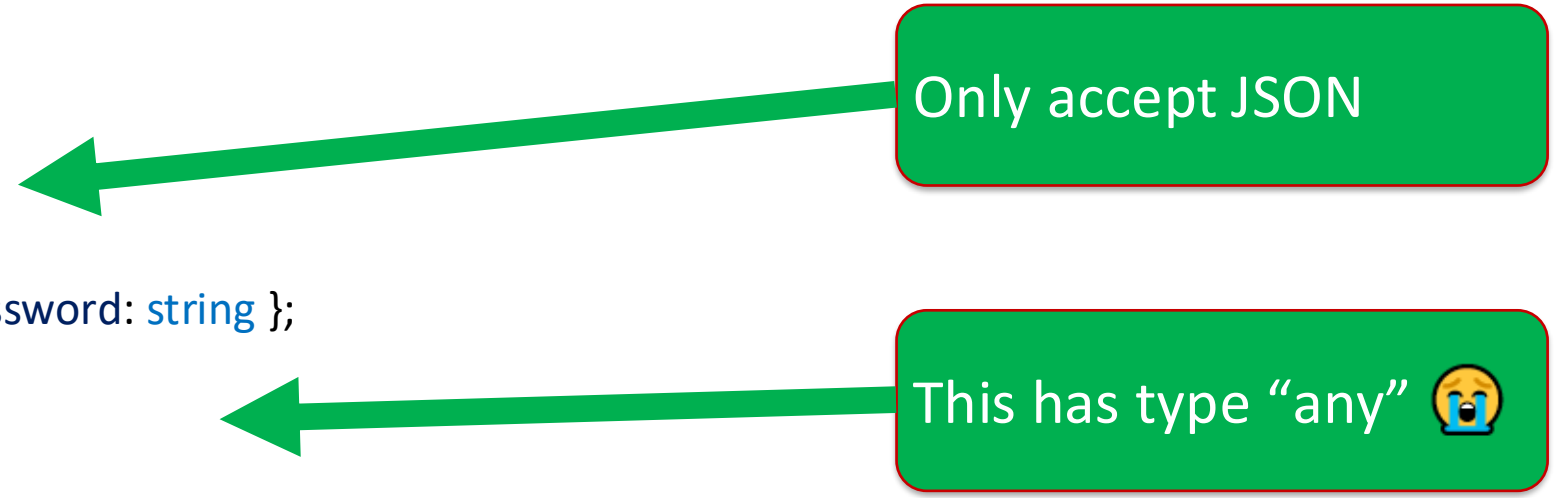
```
import { z } from 'zod';

const zAuth = z.object({ username: z.string(), password: z.string() });
type Auth = z.infer<typeof zAuth>;
```




```
// { success: false }
console.log(zAuth.safeParse({ username: 4, password: null }))
// { success: true, data: { username: "", password: "" } }
console.log(zAuth.safeParse({ username: "", password: "" }))
```

# Using “any”: Common, Not Great

```
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`));
```



# 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`));
```

# 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!