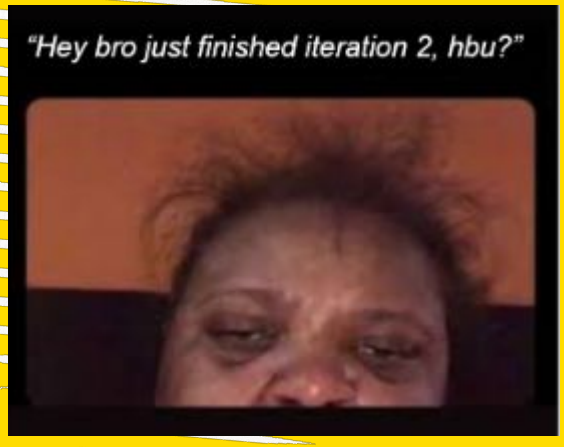
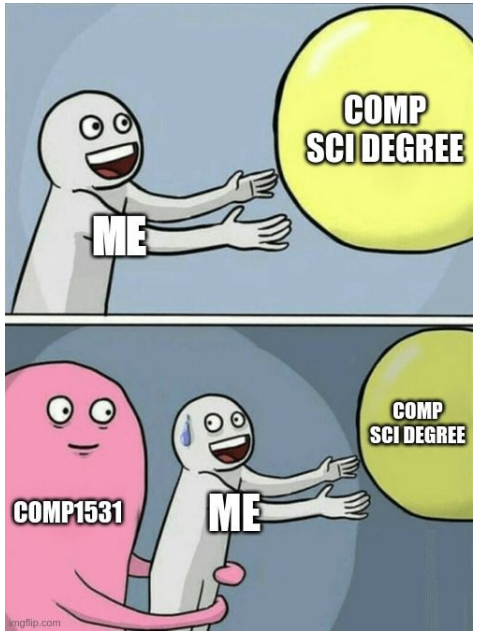


UNSW SYDNEY

COMP1531 | T09B / H17B

Week 8

william.huynh3@unsw.edu.au



Learning objectives

- **Iteration 3:** It's not that bad
- **Coverage:** Making our tests cover all our code
- **State Diagrams:** Understanding how we design software stages

Iteration 3 Tasks

48 Functions (29 are old, 19 are new)

Major Tasks (55%)

Hash your tokens & passwords

Send tokens in the header instead of body

Throw HTTP Errors instead of returning an error object

Write new HTTP Tests

Implement new functions

Minor Tasks (45% + 10% Bonus)

Code quality (**10%**)

Feature Demos (**10%**)
deployment
uploadPhoto & passwordReset
packend powers the frontend

Git & Project Management (**10%**)

Requirements Documentation (**15%**)

Fully TypeScripted Code (+10% bonus)

UNSW SYDNEY

Iteration 2 Tasks

29 Functions (11 are old, 18 are new)

Major Tasks (50%)

TypeScript your project

Setup Continuous Integration *(pipeline)*

Write new HTTP Tests

Implement new functions

Write the API (server.ts)

Setup persistence

Power the frontend

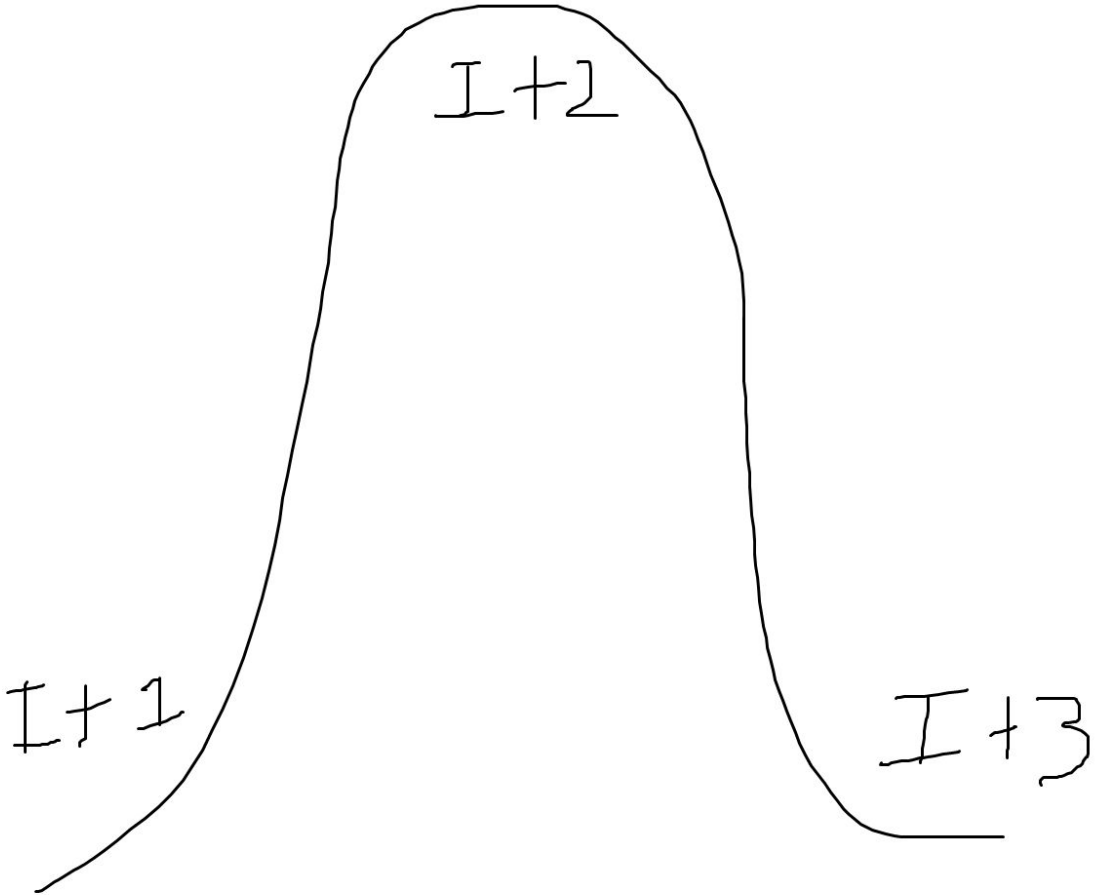
Linting your code

Minor Tasks (50%)

Code quality (30%)

Git Practices (20%)

UNSW SYDNEY


I+2
I+1
I+3

```
1
2   // A function that checks if the userId is valid
3   // Even userIds are valid, Odd userIds are not.
4   function checkIfUserValid (userId: number): string {
5     // checks if the userId is even
6     if (userId % 2 == 0) {
7       return "valid";
8     } else {
9       return "invalid";
10    }
11  }
```

```
25  test('even number is valid', () => {
26    const res = checkIfUserValid(2);
27    expect(res === "valid");
28  })
```

UNSW SYDNEY

Coverage: Making sure all our code is tested

Code coverage is a metric to quantify how much of written

code has been executed during testing.

For HTTP programs
(Iteration 3)

```
npm run ts-node-coverage
```

For regular JS programs
(Labs / Tute Activity)

```
npm run jest --coverage
```

Coverage contributes towards 10% of Iteration 3 (Code Quality criteria)

Coverage: The differences?

Not only will JEST tell you what tests you passed/failed but it will also tell you how many lines of code are not being run in your tests!

```
--- tut08/b.coverage <master> » npm run test

> tut08-coverage@1.0.0 test /home/william/Doc
> jest

PASS ./day-to-year.test.ts
  day to year tests
    ✓ dayToYear(1) => 1970 (2 ms)
    ✓ dayToYear(366) => 1971
    ✓ dayToYear(731) => 1972
    ✓ dayToYear(1097) => 1973 (1 ms)

Test Suites: 1 passed, 1 total
Tests:       4 passed, 4 total
Snapshots:   0 total
Time:        1.376 s, estimated 2 s
Ran all test suites.
--- tut08/b.coverage <master> »
```

```
PASS ./day-to-year.test.ts
  day to year tests
    ✓ dayToYear(1) => 1970 (1 ms)
    ✓ dayToYear(366) => 1971 (1 ms)
    ✓ dayToYear(731) => 1972
    ✓ dayToYear(1097) => 1973 (1 ms)

----------------|---------|----------|---------|---------|-------------------
File            | % Stmts | % Branch | % Funcs | % Lines | Uncovered Line #s
----------------|---------|----------|---------|---------|-------------------
All files       |   93.33 |       75 |     100 |    92.3 |
 day-to-year.ts |   93.33 |       75 |     100 |    92.3 | 13
----------------|---------|----------|---------|---------|-------------------
Test Suites: 1 passed, 1 total
Tests:       4 passed, 4 total
Snapshots:   0 total
Time:        1.423 s, estimated 2 s
Ran all test suites.
```

Coverage: The Metrics of Coverage

Statements

Branch

Functions

Statement Coverage

Statement coverage checks that all lines of code are executed at least once

$$\textbf{Statement Coverage} = \frac{Number\ of\ executed\ statements}{Total\ number\ of\ statements} \times 100$$

There are 6 total statements

```
1  function checkSum (a: number, b: number) {
2      const result = a + b;
3      if (result > 0) {
4          console.log("positive");
5      } else {
6          console.log("negative");
7      }
8  }
```

`a = 5, b = 10`

Line 1, 2, 3 and 4 are the executed statements

```
1  function checkSum (a: number, b: number) {
2    const result = a + b;
3    if (result > 0) {
4      console.log("positive");
5    } else {
6      console.log("negative");
7    }
8  }
```

`a = -1, b = -2`

```
1  function checkSum (a: number, b: number) {
2      const result = a + b;
3      if (result > 0) {
4          console.log("positive");
5      } else {
6          console.log("negative");
7      }
8  }
```

What are the executed statements?

Branch Coverage

Branch coverage checks how many of the branches of the control structures *(for example if statements)* have been executed.

```
1  function checkSum (a: number, b: number) {
2      const result = a + b;
3      if (result > 0) {
4          console.log("positive");
5      } else {
6          console.log("negative");
7      }
8  }
```

Function Coverage

Function coverage checks how many functions have been called in your test files

$$\text{Function coverage} = \frac{\text{Number of called functions}}{\text{Total number of functions}}$$

Function Coverage

Function coverage checks how many functions have been called in your test files

$$\text{Function coverage} = \frac{\text{Number of called functions}}{\text{Total number of functions}}$$

```ts
const ORIGIN_YEAR = 1970;       Khiet Tam Nguyen, 2 months ago • up

const isLeap = (y: number) => new Date(y, 1, 29).getDate() === 29;

export const dayToYear = (days) => {
  let year = ORIGIN_YEAR;
  while (days > 365) {
    if (isLeap(year)) {
      if (days > 366) {
        days -= 366;
        year += 1;
      } else {
        continue;
      }
    } else {
      days -= 365;
      year += 1;
    }
  }
  return year;
};

```

Activity

You have been provided a function in `b.coverage/day-to-year.ts`

In your project groups:

1. Figure out what the function does
2. Run the regular JEST tests and observe the results
3. Add coverage to JEST
4. Rerun the tests and observe the coverage report
5. Write tests to achieve 100% coverage!

Time Limit: 20 minutes

State Diagrams: Understanding how programs responds to actions

A State Diagram is a diagrammatic representation of a state.

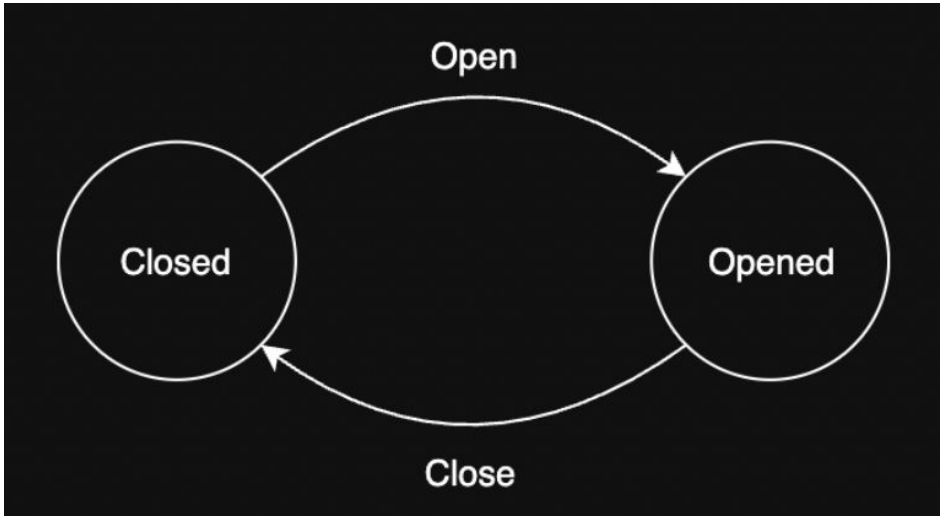
There are two symbols in a state diagram: Circles & Arrows

Circles represent states.

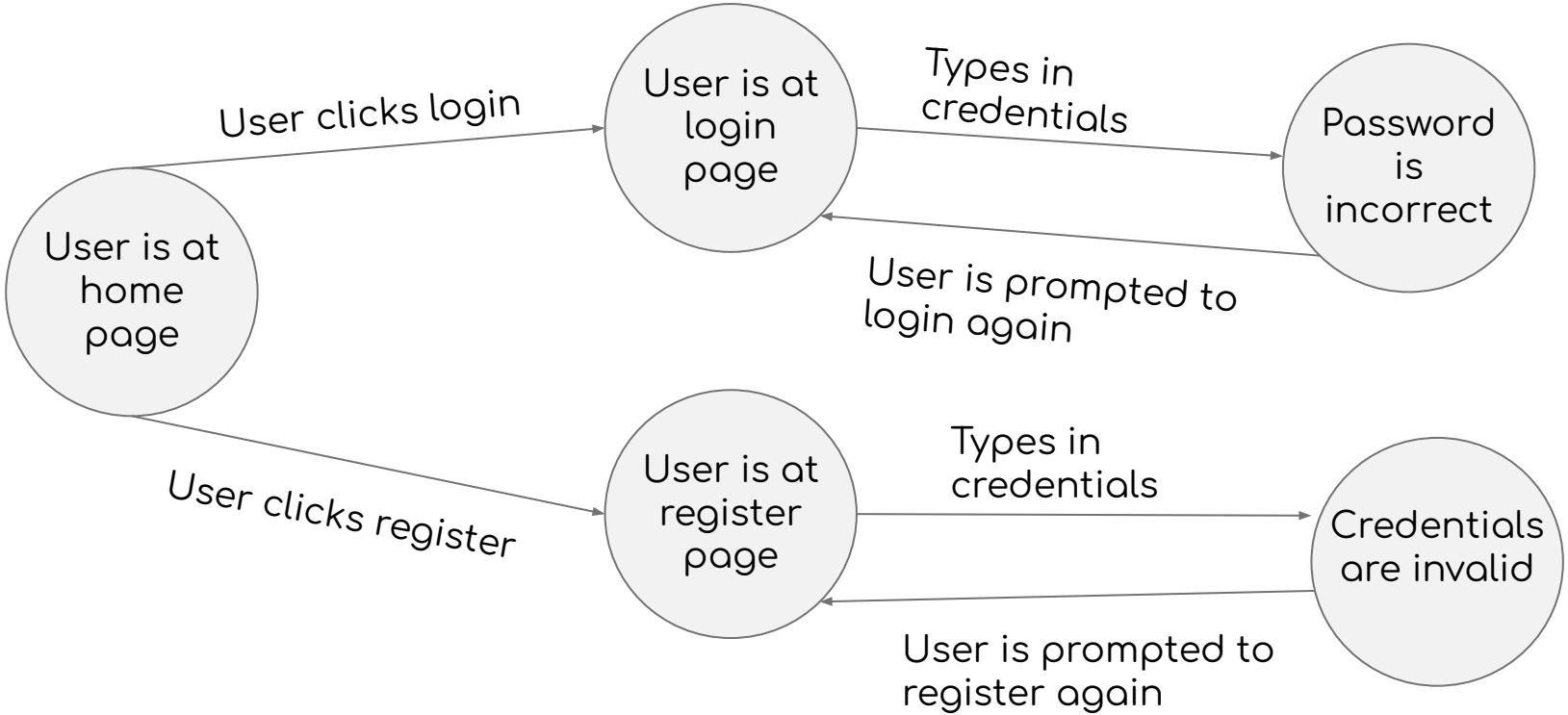
Arrows represents transitions

State Diagrams contribute to 15% of Iteration 3 (Requirements Documentation)

State Diagrams: A state diagram for a door



State Diagrams: An (incomplete) state diagram for AUTH



Activity

In your project groups, create a state diagram of a major feature in project-backend. This could be:

1. Auth *(recommended)*
2. Channel
3. Channels
4. DM
5. Users

Time Limit: 20 minutes