A DEEP DIVE INTO UNIT TESTING & MOCKING

FOR COMPLETE BEGINNERS

A Tech Talk By Sandra

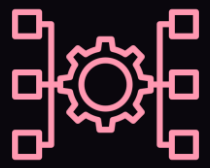
ABOUT ME

Created the ENCS Discord

Chemical Engineering @ McGill
4th year Software Engineering @ Concordia

Autodesk intern x 2
NodeJS, Infrastructure

AUTODESK



Developed scripts, metrics and server scaling tools

Infrastructure team was responsible for making the lives of developers easier

DevOps
Test Driven Development

TESTING

Testing code

1. Manually using the console
2. Automate it by writing code that tests your code

TESTING

Testing code automatically

Unit testing

UNIT TESTING

Testing code in small **units**

Units are typically individual functions

Must show that each unit behaves **correctly**

Advantages

- Facilitates safe refactoring
- Makes it easier to find bugs in complex code
- Large percent of defects detected early
- Improves code quality – TDD
- Makes other testing methods easier
- Simplifies debugging process
- Improves design of code
- Reduces costs

Disadvantages

- Increases amount of code written
- Great for business logic but not for UI
- Not testing the correct conditions compromises test results
- High coverage does not mean good tests
- Impossible to test every possible outcome

UNIT TESTING

Why I think unit testing is important

- Helped me understand my code
- Increased my analytics skills
- Favorite way of debugging

AN EXAMPLE

What are we actually going to test

- DynamoDB functions
- We’re testing completely offline

MOCKING

What is Mocking?

- Imitation of a resource
- Replace complex objects with fake objects
- Simulate behavior of imported modules
- Sinon.JS is a mocking library
- Proxy-based
- We don’t want to test external code

What is a Mocha/Chai?

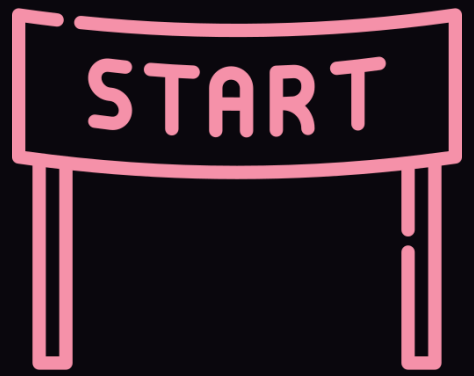
- Mocha is a unit testing library
- Chai is an assertion library
- Most libraries follow similar workflow

How are test written?

- Same as regular code but with some magic
- Each case will test 1 condition pathway

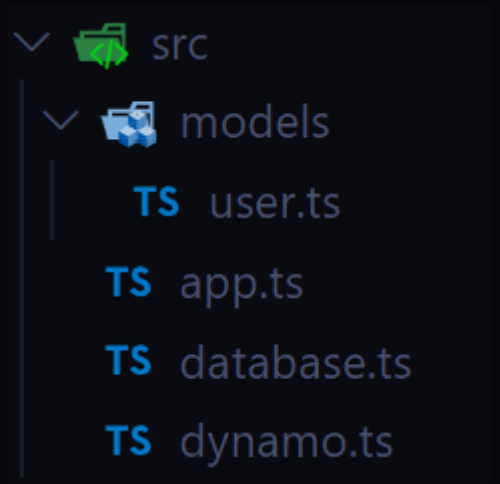
Each test case is described by an it function

```
describe('General Description', () => {
    it('Should behave like X', () => {
        assert(condition);
    });
});
```



Let's Get Started!

File structure of the app



User model

```ts
export type User = {
  name?: string;
  email?: string;
  phone?: string;
};
```

```ts
import AWS from 'aws-sdk';

/**
 * The singleton class that manages the dynamo db API
 */
class Dynamo {
  private static instance: AWS.DynamoDB;

  /**
   * The Singleton's constructor.
   */
  private constructor() {}

  /**
   * Function that returns the instance of the database to use.
   * @return {AWS.DynamoDB} An instance of the dynamo DB api.
   */
  public static getInstance(): AWS.DynamoDB {
    if (!Dynamo.instance) {
      AWS.config.update({ region: 'Local' });
      Dynamo.instance = new AWS.DynamoDB({
        apiVersion: '2012-08-10',
        endpoint: 'http://localhost:8000',
      });
    }
    return Dynamo.instance;
  }
}

export default Dynamo;
```

Singleton of
AWS DynamoDB

This is my own database object that wraps the DynamoDB functions I want to use. It uses the singleton from previous slide.

```typescript
import { AWSError } from 'aws-sdk';
import {
  PutItemInput,
  PutItemOutput,
  GetItemInput,
  GetItemOutput,
} from 'aws-sdk/clients/dynamodb';
import Dynamo from './dynamo';

/**
 * A wrapper for dynamo db.
 * @class Database
 */
export default class Database {      You, 2 days ago • added bulk code
  private db: AWS.DynamoDB;

  /**
   * @constructor
   */
  constructor() {
    this.db = Dynamo.getInstance();
  }

  /**
   * Add an item to a table in dynamo.
   * @param {PutItemInput} params
   * @return {Promise<PutItemOutput>} A promise.
   */
  public putItem = (params: PutItemInput): Promise<PutItemOutput> => {
    return this.db
      .putItem(params) // to be mocked
      .promise()
      .catch((err: AWSError) => {
        // console.error('Could not put item in', params.TableName);
        return Promise.reject(err);
      });
  };

  /**
   * Retreive an item from a table in dynamo.
   * @param {GetItemInput} params
   * @return {Promise<GetItemOutput>} A promise.
   */
  public getItem = (params: GetItemInput): Promise<GetItemOutput> => {
    return this.db
      .getItem(params) // to be mocked
      .promise()
      .catch((err: AWSError) => {
        // console.error('Could not get item from', params.TableName);
        return Promise.reject(err);
      });
  };
}
```

This is our main app entry point. It has only very basic functions to add and get a user object.

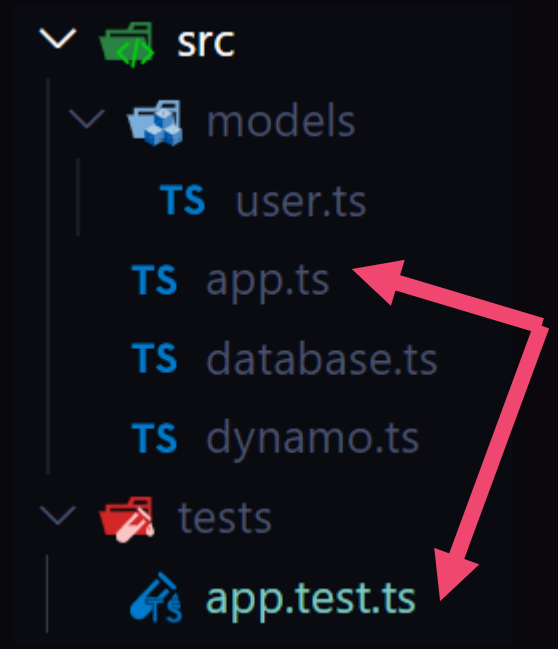
See function definitions on next slide

```
import Database from './database';
import { User } from './models/user';
import { GetItemInput, PutItemInput } from 'aws-sdk/clients/dynamodb';
import { AWSError } from 'aws-sdk';

/**
 * Main runner for dynamo db example.
 * @class Runner
 */
class Runner {
  private db: Database = new Database();

  /**
   * Adds a user to the database.
   * @param {User} user
   * @return {Promise<void>}
   */
  public addUser = (user: User): Promise<void> => {···
  };

  /**
   * Retreives a user from the database.
   * @param {string} name
   * @return {Promise<User | AWSError>}
   */
  public getUser = (name: string): Promise<User | AWSError> => {···
  };
}

export default new Runner();
```

```ts
  /**
   * Adds a user to the database.
   * @param {User} user
   * @return {Promise<void>}
   */
  public addUser = (user: User): Promise<PutItemOutput> => {
    // Pre-processing of user object to prepare it
    // for adding it to the databse

    // Fill in undefined data
    if (!user.name) {
      user.name = '';
    }

    if (!user.email) {
      user.email = '';
    }

    if (!user.phone) {
      user.phone = '';
    }

    // Create object that we are sending into the db
    const params: PutItemInput = {
      Item: {
        name: { S: user.name },
        email: { S: user.email },
        phone: { S: user.phone },
      },
      ReturnConsumedCapacity: 'TOTAL',
      TableName: 'User',
    };

    // Tell the database to add the user
    return this.db
      .putItem(params)
      .then((res) => {
        console.log('Succesfully added user to dabatase!');
        return Promise.resolve(res);
      })
      .catch((err) => {
        console.error('Could not add user to database.');
        return Promise.reject(err);
      });
  };
```

```ts
  /**
   * Retreives a user from the database.
   * @param {string} name
   * @return {Promise<User | AWSError>}
   */
  public getUser = (name: string): Promise<User | AWSError> => {
    // Create object that we are gettin from the db
    const params: GetItemInput = {
      Key: { name: { S: name } },
      TableName: 'User',
    };

    // Tell the database to add the user
    return this.db
      .getItem(params)
      .then((data) => {
        let user: User = {};

        // Check if we returned an item
        if (data.Item) {
          user = {
            email: data.Item.email.S,
            name: data.Item.name.S,
            phone: data.Item.phone.S,
          };
          console.log('Succesfully retreived user from dabatase!');
          return Promise.resolve(user);
        } else {
          console.log('Database returned empty item');
          const err = new Error('Database returned no data.');
          return Promise.reject(err);
        }
      })
      .catch((err: AWSError) => {
        console.error('Could not add user to database.');
        return Promise.reject(err);
      });
  };
}
```

The TDD test




```
import runner from '../src/app';
import { assert } from 'chai';
import 'mocha';

describe('User functions test', () => {
  it('should add a user to the db', () => {
    return runner.addUser({}).then((output) => {
      assert.exists(output);
    });
  });

  it('should NOT add a user to the db', () => {
    return runner
      .addUser({})
      .then(() => {
        assert.fail('This should have failed');
      })
      .catch((err) => {
        assert.exists(err);
      });
  });
});
```

We start with what we want to test. Even if running this test will obviously not pass, it’s a great starting point.

Why would it obviously fail?

We technically accounted for an empty user object by adding undefined parameters as empty strings. So why would case 1 fail?

Missing resources!

```
import runner from '../src/app';
import { assert } from 'chai';
import 'mocha';

describe('User functions test', () => {
  it('should add a user to the db', () => {
    return runner.addUser({}).then((output) => {
      assert.exists(output);
    });
  });

  it('should NOT add a user to the db', () => {
    return runner
      .addUser({})
      .then(() => {
        assert.fail('This should have failed');
      })
      .catch((err) => {
        assert.exists(err);
      });
  });
});
```

Test output

```
>> Sandra :: techTalk git(master) 12:20  npm test

> techtalk@1.0.0 test D:\src\techTalk
> mocha -r ts-node/register './tests/**/*.test.ts'



  User functions test
    1) should add a user to the db
    2) should NOT add a user to the db


  0 passing (4s)
  2 failing

  1) User functions test
       should add a user to the db:
     Error: Timeout of 2000ms exceeded. For async tests and hooks, ensure "done()" is called; if returning a Promise, ensure
it resolves. (D:\src\techTalk\tests\app.test.ts)
      at listOnTimeout (internal/timers.js:549:17)
      at processTimers (internal/timers.js:492:7)

  2) User functions test
       should NOT add a user to the db:
     Error: Timeout of 2000ms exceeded. For async tests and hooks, ensure "done()" is called; if returning a Promise, ensure
it resolves. (D:\src\techTalk\tests\app.test.ts)
      at listOnTimeout (internal/timers.js:549:17)
      at processTimers (internal/timers.js:492:7)



Could not add user to database.
Could not add user to database.
npm ERR! Test failed.  See above for more details.
```

Tests timing out means we are waiting for a promise that's taking too long to finish

30 seconds later, my promises actually reject, but my tests already happened

```
  /**
   * Adds a user to the database.
   * @param {User} user
   * @return {Promise<void>}
   */
  public addUser = (user: User): Promise<PutItemOutput> => {
    // Pre-processing of user object to prepare it
    // for adding it to the databse

    // Fill in undefined data
    if (!user.name) {
      user.name = '';
    }

    if (!user.email) {
      user.email = '';
    }

    if (!user.phone) {
      user.phone = '';
    }

    // Create object that we are sending into the db
    const params: PutItemInput = {
      Item: {
        name: { S: user.name },
        email: { S: user.email },
        phone: { S: user.phone },
      },
      ReturnConsumedCapacity: 'TOTAL',
      TableName: 'User',
    };

    // Tell the database to add the user
    return this.db
      .putItem(params)
      .then((res) => {
        console.log('Succesfully added user to dabatase!');
        return Promise.resolve(res);
      })
      .catch((err) => {
        console.error('Could not add user to database.');
        return Promise.reject(err);
      });
  };
```

```
import { AWSError } from 'aws-sdk';
import {
  PutItemInput,
  PutItemOutput,
  GetItemInput,
  GetItemOutput,
} from 'aws-sdk/clients/dynamodb';
import Dynamo from './dynamo';

/**
 * A wrapper for dynamo db.
 * @class Database
 */
export default class Database {      You, 2 days ago • added bulk code
  private db: AWS.DynamoDB;

  /**
   * @constructor
   */
  constructor() {
    this.db = Dynamo.getInstance();
  }

  /**
   * Add an item to a table in dynamo.
   * @param {PutItemInput} params
   * @return {Promise<PutItemOutput>} A promise.
   */
  public putItem = (params: PutItemInput): Promise<PutItemOutput> => {
    return this.db
      .putItem(params) // to be mocked
      .promise()
      .catch((err: AWSError) => {
        // console.error('Could not put item in', params.TableName);
        return Promise.reject(err);
      });
  };

  /**
   * Retreive an item from a table in dynamo.
   * @param {GetItemInput} params
   * @return {Promise<GetItemOutput>} A promise.
   */
  public getItem = (params: GetItemInput): Promise<GetItemOutput> => {
    return this.db
      .getItem(params) // to be mocked
      .promise()
      .catch((err: AWSError) => {
        // console.error('Could not get item from', params.TableName);
        return Promise.reject(err);
      });
  };
}
```

Let’s start adding mocks.

The beforeEach and afterEach functions will before and after each test case.

It is very important to **restore any mock** after a test.

```
import runner from '../src/app';
import { assert } from 'chai';
import 'mocha';
import Dynamo from '../src/dynamo';
import Sinon from 'sinon';

describe('User functions test', () => {
  let dynamo: AWS.DynamoDB;
  let sandbox: Sinon.SinonSandbox;

  beforeEach(() => {
    dynamo = Dynamo.getInstance();
    sandbox = Sinon.createSandbox();
  });

  afterEach(() => {
    // cleanup
    sandbox.restore();
  });

  it('should add a user to the db', () => {…
  });

  it('should NOT add a user to the db', () => {…
  });
});

```

I know it looks ugly, but all we are doing is satisfying the AWS object inputs and outputs

Because AWS defines promises on their functions by calling .promise() on them I have to imitate that behavior too.

```
it('should add a user to the db', () => {
  const putItemOutput: PutItemOutput = {
    ConsumedCapacity: { TableName: 'User', CapacityUnits: 1 },
  };

  const putItemOutputResolves = ({
    promise() {
      return Promise.resolve(putItemOutput);
    },
  } as unknown) as AWS.Request<PutItemOutput, AWSError>;

  sandbox.stub(dynamo, 'putItem').returns(putItemOutputResolves);

  return runner.addUser({}).then((output) => {
    assert.exists(output);
  });
});
```

I skipped my other test to focus on only this test case. We're still failing!

That's ok, it's actually because we haven't tested for our condition yet. Notice AssertionError means our condition is not being met.

But what condition should we test?

```
>> Sandra :: techTalk git(master) x 12:38  npm test

> techtalk@1.0.0 test D:\src\techTalk
> mocha -r ts-node/register './tests/**/*.test.ts'



  User functions test
Succesfully added user to dabatase!
    1) should add a user to the db
    - should NOT add a user to the db


  0 passing (19ms)
  1 pending
  1 failing

  1) User functions test
       should add a user to the db:
     AssertionError: expected undefined to exist
      at D:\src\techTalk\tests\app.test.ts:37:14



npm ERR! Test failed.  See above for more details.
```

```ts
it('should add a user to the db', () => {
  const user: User = {
    name: 'Matt',
    email: 'student@concordia.ca',
    phone: '5146663333',
  };

  const putItemOutput: PutItemOutput = {
    ConsumedCapacity: { TableName: 'User', CapacityUnits: 1 },
  };

  const putItemOutputResolves = ({
    promise() {
      return Promise.resolve(putItemOutput);
    },
  } as unknown) as AWS.Request<PutItemOutput, AWSError>;

  const params: PutItemInput = {
    Item: {
      name: { S: user.name },
      email: { S: user.email },
      phone: { S: user.phone },
    },
    ReturnConsumedCapacity: 'TOTAL',
    TableName: 'User',
  };

  sandbox
    .stub(dynamo, 'putItem')
    .withArgs(Sinon.match(params))
    .returns(putItemOutputResolves);

  return runner.addUser(user).then((res) => {
    assert.equal(res, putItemOutput);
  });
});
```

```
>> Sandra :: techTalk git(master) x 13:51  npm test

> techtalk@1.0.0 test D:\src\techTalk
> mocha -r ts-node/register './tests/**/*.test.ts'



  User functions test
Succesfully added user to dabatase!
    ✓ should add a user to the db
    - should NOT add a user to the db


  1 passing (14ms)
  1 pending
```

```ts
it('should NOT add a user to the db', () => {
  const awsError: AWSError = {
    code: 'badRequest',
    message: 'bad request',
    retryable: false,
    statusCode: 1,
    time: new Date(),
    name: '',
    hostname: '',
    region: '',
    retryDelay: 1,
    requestId: '',
    extendedRequestId: '',
    cfId: '',
  };

  const putItemOutputRejects = ({
    promise() {
      return Promise.reject(awsError);
    },
  } as unknown) as AWS.Request<PutItemOutput, AWSError>;

  const user: User = {
    name: 'Matt',
    email: 'student@concordia.ca',
    phone: '5146663333',
  };

  const params: PutItemInput = {
    Item: {
      name: { S: user.name },
      email: { S: user.email },
      phone: { S: user.phone },
    },
    ReturnConsumedCapacity: 'TOTAL',
    TableName: 'User',
  };

  sandbox
    .stub(dynamo, 'putItem')
    .withArgs(Sinon.match(params))
    .returns(putItemOutputRejects);

  return runner
    .addUser(user)
    .then(() => {
      assert.fail('add user should fail');
    })
    .catch((err) => {
      assert.equal(err, awsError);
    });
});
```

```
>> Sandra :: techTalk git(master) 13:54  npm test

> techtalk@1.0.0 test D:\src\techTalk
> mocha -r ts-node/register './tests/**/*.test.ts'



  User functions test
Succesfully added user to dabatase!
    ✓ should add a user to the db
Could not add user to database.
    ✓ should NOT add a user to the db


  2 passing (16ms)
```

```
119     it('should get a user', () => {
120       const user: User = {
121         name: 'Matt',
122         email: 'student@concordia.ca',
123         phone: '5146663333',
124       };
125
126       const params: GetItemInput = {
127         Key: { name: { S: user.name } },
128         TableName: 'User',
129       };
130
131       const getItemOutput: GetItemOutput = {
132         Item: {
133           name: { S: user.name },
134           email: { S: user.email },
135           phone: { S: user.phone },
136         },
137         ConsumedCapacity: { TableName: 'User', CapacityUnits: 1 },
138       };
139
140       const getItemResolves = ({
141         promise() {
142           return Promise.resolve(getItemOutput);
143         },
144       } as unknown) as AWS.Request<GetItemOutput, AWSError>;
145
146       sandbox
147         .stub(dynamo, 'getItem')
148         .withArgs(Sinon.match(params))
149         .returns(getItemResolves);
150
151       return runner.getUser(user.name || '').then((res: User) => {
152         assert(res.name, user.name);
153         assert(res.email, user.email);
154         assert(res.phone, user.phone);
155       });
156     });
157   });
```

```
>> Sandra :: techTalk git(master) x 14:03  npm test

> techtalk@1.0.0 test D:\src\techTalk
> mocha -r ts-node/register './tests/**/*.test.ts'



  User functions test
Succesfully added user to dabatase!
    ✓ should add a user to the db
Could not add user to database.
    ✓ should NOT add a user to the db
Succesfully retreived user from dabatase!
    ✓ should get a user


  3 passing (18ms)
```

```
158   it('should NOT get a user', () => {
159     const awsError: AWSError = {
160       code: 'badRequest',
161       message: 'bad request',
162       retryable: false,
163       statusCode: 1,
164       time: new Date(),
165       name: '',
166       hostname: '',
167       region: '',
168       retryDelay: 1,
169       requestId: '',
170       extendedRequestId: '',
171       cfId: '',
172     };
173
174     const user: User = {
175       name: 'Matt',
176       email: 'student@concordia.ca',
177       phone: '5146663333',
178     };
179
180     const params: GetItemInput = {
181       Key: { name: { S: user.name } },
182       TableName: 'User',
183     };
184
185     const getItemResolves = ({
186       promise() {
187         return Promise.reject(awsError);
188       },
189     } as unknown) as AWS.Request<GetItemOutput, AWSError>;
190
191     sandbox
192       .stub(dynamo, 'getItem')
193       .withArgs(Sinon.match(params))
194       .returns(getItemResolves);
195
196     return runner
197       .getUser(user.name || '')
198       .then((res: User) => {
199         assert.fail('should fail get the user');
200       })
201       .catch((err) => {
202         assert.equal(err, awsError);
203       });
204   });
```

```
>> Sandra :: techTalk git(master) x 14:04  npm test

> techtalk@1.0.0 test D:\src\techTalk
> mocha -r ts-node/register './tests/**/*.test.ts'



  User functions test
Succesfully added user to dabatase!
    √ should add a user to the db
Could not add user to database.
    √ should NOT add a user to the db
Succesfully retreived user from dabatase!
    √ should get a user
Could not add user to database.
    √ should NOT get a user


  4 passing (17ms)
```

```ts
it('should reject if no user found', () => {
  const user: User = {
    name: 'Matt',
    email: 'student@concordia.ca',
    phone: '5146663333',
  };

  const params: GetItemInput = {
    Key: { name: { S: user.name } },
    TableName: 'User',
  };

  const getItemOutput: GetItemOutput = {
    ConsumedCapacity: { TableName: 'User', CapacityUnits: 1 },
  };

  const getItemResolves = ({
    promise() {
      return Promise.resolve(getItemOutput);
    },
  } as unknown) as AWS.Request<GetItemOutput, AWSError>;

  sandbox
    .stub(dynamo, 'getItem')
    .withArgs(Sinon.match(params))
    .returns(getItemResolves);

  return runner
    .getUser(user.name || '')
    .then((res: User) => {
      assert.fail('Should fail getting user');
    })
    .catch((err) => {
      assert.equal(err.message, 'Database returned no data.');
    });
});
```

```
>> Sandra :: techTalk git(master) x 14:17  npm test

> techtalk@1.0.0 test D:\src\techTalk
> mocha -r ts-node/register './tests/**/*.test.ts'



  User functions test
Succesfully added user to dabatase!
    √ should add a user to the db
Could not add user to database.
    √ should NOT add a user to the db
Succesfully retreived user from dabatase!
    √ should get a user
Database returned empty item
Could not add user to database.
    √ should reject if no user found
Could not add user to database.
    √ should NOT get a user
```