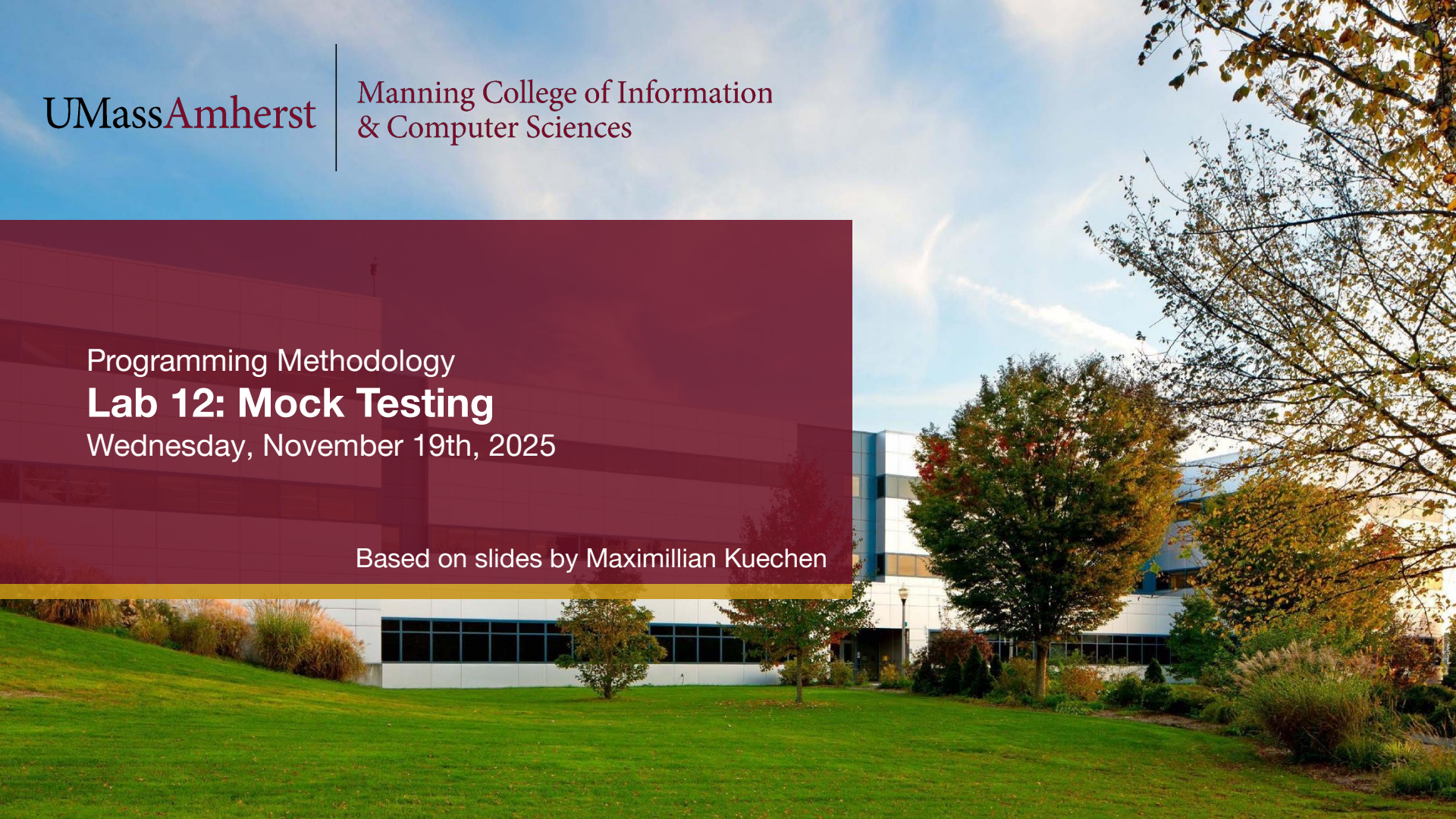


UMassAmherst | Manning College of Information & Computer Sciences

Programming Methodology

Lab 12: Mock Testing

Wednesday, November 19th, 2025

Based on slides by Maximillian Kuechen

Weekly Lab Agenda

- Go over reminders/goals
- Review past material
- Work in groups of 2-3 to solve a few exercises
- Discussion leaders will walk around and answer questions
- Solutions to exercises will be reviewed as a class
- Attendance taken at the end

Reminders

- HW 6 is due this Sunday (11/23) at 11:59pm
- Get in touch ASAP with your team for HW6 if you haven't already!
- Make sure you've joined the repo for you team
  - Around 30 students still have yet to accept their invites
  - Invites have now expired; post on Ed if you need a new invite!
- Reach out to us on Ed privately if there are any issues in your team.
  - Reach out early so we can get you back on track!

Today's Goals

- Practice testing asynchronous code and mocking fetch

Testing Asynchronous Code

We can use jest to help us test asynchronous code in various ways.

We will go over some brief examples for an asynchronous function MyFunction, which takes some sort of input and returns a Promise.

At the top of our testing file, we can use jest to define a time limit for running all our tests. The code to the right will set a 30 second time limit (jest.setTimeout takes milliseconds as input).

```
const SECOND = 1000;
jest.setTimeout(30 * SECOND);
```

Testing Asynchronous Code

Below, res is a Promise. We can then use expect(res).resolves or rejects to check if the Promise resolves/rejects, and examine the value that the Promise resolves/rejects with.

```
describe('Tests for MyFunction', () => {
  it('Testing resolve', () => {
    const res = MyFunction(someInput);
    return expect(res).resolves.toBe(someValue);
  });

  it('Testing reject', () => {
    const res = MyFunction(someInput);
    return expect(res).rejects.toBe(someValue);
  });
});
```

Note that you must include the return statement, as this allows jest to keep track of uncompleted tests.

Testing Asynchronous Code

Alternatively, we could also use .then() instead, or .catch() in a similar fashion. Again, we must include the return statement.

```
describe('Tests for MyFunction', () => {
  it('Some test', () => {
    const res = MyFunction(someInput);
    return res.then(val =>
      expect(val).toBe(someValue)
    );
  });
});
```

Testing Asynchronous Code

We can also use async/await. We need to specify async in front of the function passed to the it block though in order to use await in that test.

```
describe('Tests for MyFunction', () => {
  it('Some test', async () => {
    const res = await MyFunction(someInput);
    expect(res).toBe(someValue);
  });
});
```

When using async/await, we do not need the return statement.

Mock Testing

There are times when we implement a function that calls some other existing function, which may be complex, unpredictable, or expensive to call.

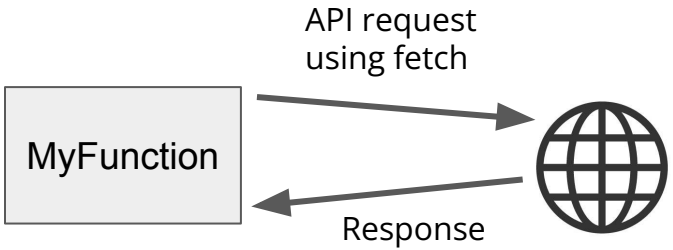
This can make testing difficult, as we may want to avoid calling this other function too much. However, we still want a way to test if our function will work for different outputs of this other function.

We can instead intercept all calls to this function, controlling what is returned. This is known as mock testing.

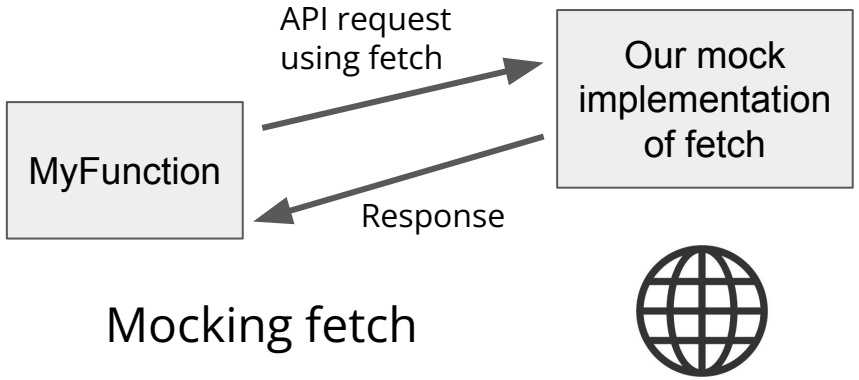
Mocking Fetch

Fetch is a good example of a function that we can use for mock testing. For instance, if our function relies on an API we can't control, or an API that is expensive to call repeatedly, it would be nice if we could "simulate" the API.

We can do so by mocking fetch, which allows us to intercept any API calls from our function and return whatever values we want during testing.



Normal API usage



Mocking fetch

Mocking Fetch

We will use the jest-fetch-mock library to help us mock fetch.

To mock fetch, we must first call fetchMock.enableMocks(). This tells jest that we intend to mock fetch.

We can then use fetchMock.mockResponse(val) to specify what our “intercepted” fetch should return.

We can use JSON.stringify() if we want fetch to return JSON.

```
import fetchMock from "jest-fetch-mock";

describe('Tests for MyFunction', () => {
  it('Some test', () => {
    fetchMock.enableMocks();

    const input = JSON.stringify([1, 2, 3]);
    fetchMock.mockResponse(input);
  });
});
```

Mocking Fetch

```
describe('Tests for MyFunction', () => {
  it('Some test', async () => {
    fetchMock.enableMocks();

    const input = JSON.stringify([1, 2, 3]);
    fetchMock.mockResponse(input);

    const res = await fetch("test_url");

    fetchMock.resetMocks();
    fetchMock.disableMocks();
  });
});
```

When we now call fetch, it will use our implementation. This means res will a Response object. If we call .json() on this Response object, we will obtain the array [1, 2, 3].

We should then call resetMocks() to clear our mocks. We can then disable mocks on fetch with disableMocks(), which allows fetch to revert to normal and not be intercepted anymore.

Mocking Fetch

```
describe('Tests that mock fetch', () => {
  beforeEach(() => {
    fetchMock.enableMocks();
  });

  afterEach(() => {
    fetchMock.resetMocks();
    fetchMock.disableMocks();
  });

  // insert tests here
});
```

enableMocks, resetMocks, and disableMocks need to be in every test that mocks fetch.

If we have multiple tests that need to mock fetch, we can reduce code duplication!

We will first define a describe block for just the tests that need to mock fetch.

We can then use beforeEach and afterEach in this describe block, which calls the desired functions before and after each test.

Mocking Fetch - Example

Suppose we want to write a function that takes a url as input. If everything works, we should be able to extract an array from calling fetch on this url. We want to then output “yes” or “no” depending on if this array has length less than or equal to three.

However, we should be careful! What if fetch returns a Promise that rejects? What if fetch resolves to a Response object where the ok property is false? What if everything else works but we don’t receive an array?

```
function arrayIsShort(url) {
  return fetch(url)
    .then(res => res.ok ? res.json() : Promise.reject(new Error("Response error")))
    .then(res => Array.isArray(res) ? res : Promise.reject(new Error("Not an array")))
    .then(output => output.length <= 3 ? "yes" : "no")
    .catch(error => error.message);
}
```

Mocking Fetch - Example

```
describe("arrayIsShort", () => {
  beforeEach(() => {
    fetchMock.enableMocks();
  });
  afterEach(() => {
    fetchMock.resetMocks();
    fetchMock.disableMocks();
  });

  it("returns yes if promise fulfills with short array", async () => {
    fetchMock.mockResponseOnce(JSON.stringify([1]));
    const res = await arrayIsShort("test_url");
    expect(res).toEqual("yes");
  });
  // ...
});
```

We call beforeEach and afterEach to set up our mocks for each test. In this first test, we demonstrate using mockResponseOnce to mock the Response from fetch for just one call.

Mocking Fetch - Example

```
describe("arrayIsShort", () => {
  it("returns no if promise fulfills with long array", async () => {
    fetchMock.mockImplementation(url => {
      const arr = JSON.stringify([1, 2, 3, 4, 5]);
      return Promise.resolve(new Response(arr));
    });
    const res = await arrayIsShort("test_url");
    expect(res).toEqual("no");
  });

  it("returns uh oh if promise from fetch rejects", async () => {
    fetchMock.mockReject(new Error("Test"));
    const res = await arrayIsShort("test_url");
    expect(res).toEqual("Test");
  });
  // ...
});
```

The second test demonstrates how we can use mockImplementation to have more control over how we mock fetch. The third test uses mockReject to make fetch reject.

Mocking Fetch - Example

```
describe("arrayIsShort", () => {
  it("returns uh oh if fetch produces not ok response", async () => {
    fetchMock.mockResponseOnce(JSON.stringify([]), { status: 404 });
    const res = await arrayIsShort("test_url");
    expect(res).toEqual("Response error");
  });

  it("returns uh oh if fetch doesnt produce an array", async () => {
    fetchMock.mockResponseOnce(JSON.stringify({ a: 1 }));
    const res = await arrayIsShort("test_url");
    expect(res).toEqual("Not an array");
  });
});
```

The fourth test demonstrates how we can have fetch produce a Response with the ok property being false. There are several statuses that result in the ok property being false, with 404 (not found) and 500 (internal server error) being two common ones.

The last test just examines the behavior when fetch doesnt produce an array.

Mock Testing

You can learn more about testing asynchronous code and mock testing in the slides linked in the HW 6 specs: Testing Asynchronous Code

To practice mocking fetch, we will introduce a function that we can test by mocking fetch. To help you understand the function, we will ask you to try and implement it for exercise 1.

We will only spend around 5-7 minutes on this exercise; do not worry if you do not solve it in time! Our main focus will be on writing tests for this function.

Exercise 1: Promises

Write a function getObjsWithName that takes in an array of URLs and returns a promise of an array of objects. The array of objects should contain all objects with the property “name” found at every URL that points to JSON data.

Assume (1) All urls point to valid JSON data.
(2) Assume the data is an array of objects.

Hint: You may use “in” to check properties

Exercise 1: Promises

Write a function getObjsWithName that takes in an array of URLs and returns a promise of an array of objects. The array of objects should contain all objects with the property “name” found at every URL that points to JSON data.

fetch() will return a promise, which may resolve or reject
=> we get an array of Promises

Use response.json() to get JSON representation of the GET response.
=> json() returns a Promise

Exercise 1: Promises

… returns a Promise of an array of objects. The array of objects should contain all objects with the property “name” found at every URL that points to JSON data.

Wrap result in Promise.allsettled.

=> Call to Promise.allsettled always fulfills with an array of objects.

```
{ status: "fulfilled" | "rejected", value?: someType, reason?: errType }
```

Use filter and in to check if object has property `name`.

For each url, we have an of array of objects with “name” property
Need to combine results from all urls into one, i.e, go from Object[ ][ ]⇒ Object[ ]
=> flatten the result into a single array using flat()

Exercise 2: Mock Testing

Now, write tests for exercise 1! You should mock fetch in all your tests. Hint: use fetchMock.mockImplementation() to control what fetch does depending on the url it is called on.

You can try running your tests on the flawed solutions in lab.js.

Exercise 1 for reference:

Write a function getObjsWithName that takes in an array of URLs and returns a promise of an array of objects. The array of objects should contain all objects with the property "name" found at every URL that points to JSON data.

Assume (1) All urls point to valid JSON data.
(2) Assume the data is an array of objects.

Exercise 3: Mock Testing

Now, suppose we did not have the following two assumptions from exercise 1:

Assume (1) All urls point to valid JSON data.
(2) Assume the data is an array of objects.

We will instead just assume that, if the ok property of a Response from fetch is true, then calling .json() on this response will always succeed. For simplicity, you can also assume that any array returned must be an array of objects.

We would then need to adjust our implementation from exercise 1 in order to handle more scenarios that fetch may present us with. For example, what if we don't receive an array? What if the Response is not ok?

Updating Exercise 1 Solution

We can add the two lines in bold in order to handle the weaker assumptions from the last slide.

```
function getObjsWithName(urls) {
     return Promise.allSettled(
          urls.map(url =>
               fetch(url)
               .then(res => res.ok ? res.json() : Promise.reject(new Error()))
               .then(res => Array.isArray(res) ? res : [])
          )
     ).then(resarr =>
          resarr
            .filter(res => res.status === "fulfilled")
            .map(res => res["value"])
            .map(objarr => objarr.filter(obj => "name" in obj))
            .flat()
  );
}
```

Exercise 3: Mock Testing

Now, add new tests to your test suite for getObjsWithName to test that this function behaves correctly for the updated problem statement.

Again, consider what you now need to test, given the weaker assumptions. What if we don't receive an array? What if the Response is not ok? How can you test these scenarios using mock testing?

Some of the earlier slides in this lab may be useful!