

Nice, this is a really important Cypress skill. Let's go step by step and keep it beginner-friendly. We'll cover:

1. What "intercepting network requests" means
  2. Mocking **GraphQL** with `cy.intercept()`
  3. API testing with `cy.request()` (REST)
  4. Verifying **request & response payloads** (both UI + API style)
- 

## 0. Basic Setup (once per project)

If you already have a Cypress project, you can skip.

```
mkdir cypress-network-demo
cd cypress-network-demo
npm init -y
npm install cypress --save-dev
npx cypress open
```

Cypress will create a `cypress/` folder. We'll use `cypress/e2e/` for tests.

---

## 1. What is `cy.intercept()`?

In a web app, when you click a button, the frontend often calls an API:

- REST: `GET /api/users`
- GraphQL: `POST /graphql`

`cy.intercept()` lets you:

- **Watch** those calls (spy): see what was sent & received

- **Control** them (stub/mock): instead of calling the real backend, return fake data

This is super useful when:

- Backend is not ready
  - Backend is slow or flaky
  - You want predictable test data
- 

## 2. Mocking GraphQL with `cy.intercept()`

### 2.1 Typical GraphQL request

Usually, a GraphQL client sends:

POST /graphql

Content-Type: application/json

```
{
  "operationName": "GetUsers",
  "query": "query GetUsers { users { id name } }",
  "variables": {}
}
```

We want to:

- Intercept that `POST /graphql`
- Check the `operationName`
- If it's `"GetUsers"`, return our fake users

### 2.2 Cypress test file

Create file: `cypress/e2e/graphql_intercept.cy.js`

```
/// <reference types="cypress" />
```

```

describe('GraphQL intercept demo', () => {
  it('mocks GetUsers GraphQL query', () => {
    // 1. Setup intercept BEFORE visiting the page
    cy.intercept('POST', '/graphql', (req) => {
      // req.body is the GraphQL payload
      const { operationName } = req.body

      if (operationName === 'GetUsers') {
        // 2. Respond with fake data
        req.reply({
          statusCode: 200,
          body: {
            data: {
              users: [
                { id: '1', name: 'Alice' },
                { id: '2', name: 'Bob' },
              ],
            },
          },
        })
      } else {
        // For other operations, let them continue to real backend
        req.continue()
      }
    }).as('getUsers')

    // 3. Visit your app (assume it calls POST /graphql GetUsers on load)
    cy.visit('http://localhost:3000')

    // 4. Wait for GraphQL call to happen
    cy.wait('@getUsers').then((interception) => {
      // 5. Verify request payload
      expect(interception.request.body.operationName).to.eq('GetUsers')

      // 6. Verify response payload (our mock)
      expect(interception.response.statusCode).to.eq(200)
      expect(interception.response.body.data.users).to.have.length(2)
    })

    // 7. Verify UI uses our fake data
    cy.contains('Alice').should('be.visible')
    cy.contains('Bob').should('be.visible')
  })
})

```

```
}}
```

Even if your real backend returns 1000 users, Cypress will **force** the response to be just Alice and Bob. So UI + test become predictable.

---

### 3. API Testing with `cy.request()` (REST)

`cy.request()` is used for **direct API testing**, without using the browser UI.

Think of it like Postman, but inside Cypress tests.

#### 3.1 Simple GET example (using a fake `/api/users`)

Create file: `cypress/e2e/rest_api.cy.js`

```
/// <reference types="cypress" />
```

```
describe('REST API testing with cy.request', () => {
  const baseUrl = 'http://localhost:3000' // change to your API URL

  it('GET /api/users - should return users list', () => {
    cy.request({
      method: 'GET',
      url: `${baseUrl}/api/users`,
    }).then((response) => {
      // 1. Status code
      expect(response.status).to.eq(200)

      // 2. Response body shape
      expect(response.body).to.be.an('array')

      // 3. Verify one item
      if (response.body.length > 0) {
        const user = response.body[0]
        expect(user).to.have.property('id')
        expect(user).to.have.property('name')
      }
    })
  })
})
```

### 3.2 POST example (creating a user)

```
it('POST /api/users - should create a user', () => {
  cy.request({
    method: 'POST',
    url: `${baseUrl}/api/users`,
    body: {
      name: 'New User',
      email: 'new@example.com',
    },
  }).then((response) => {
    expect(response.status).to.eq(201) // Created
    expect(response.body).to.have.property('id')
    expect(response.body.name).to.eq('New User')
  })
})
```

### 3.3 PUT / PATCH example (updating a user)

```
it('PUT /api/users/:id - should update a user', () => {
  const updatedName = 'Updated User'

  cy.request({
    method: 'PUT',
    url: `${baseUrl}/api/users/1`,
    body: { name: updatedName },
  }).then((response) => {
    expect(response.status).to.eq(200)
    expect(response.body.name).to.eq(updatedName)
  })
})
```

### 3.4 DELETE example

```
it('DELETE /api/users/:id - should delete a user', () => {
  cy.request({
    method: 'DELETE',
    url: `${baseUrl}/api/users/1`,
  }).then((response) => {
    // Some APIs return 200, some 204
    expect([200, 204]).to.include(response.status)
  })
})
```

This is pure API testing, no UI involved.

---

## 4. Verifying Request & Response Payloads

You'll do this in **two places**:

1. UI tests using `cy.intercept()` + `cy.wait()`
2. API tests using `cy.request()`

### 4.1 Verify payloads in UI tests (with intercept)

Example: UI calls `POST /api/login` when you submit a login form.

```
describe('Login flow', () => {
  it('sends correct request and handles response', () => {
    // 1. Set up intercept for login API
    cy.intercept('POST', '/api/login').as('login')

    // 2. Visit page and perform login
    cy.visit('http://localhost:3000/login')
    cy.get('input[name="email"]').type('test@example.com')
    cy.get('input[name="password"]').type('password123')
    cy.get('button[type="submit"]').click()

    // 3. Wait for API call and inspect it
    cy.wait('@login').then((interception) => {
      const { request, response } = interception

      // ---- Verify Request Payload ----
      expect(request.body).to.deep.equal({
        email: 'test@example.com',
        password: 'password123',
      })

      // ---- Verify Response Payload ----
      // assuming backend returns { token: "abc", user: { id, name } }
      expect(response.statusCode).to.eq(200)
      expect(response.body).to.have.property('token')
      expect(response.body.user).to.have.property('id')
```

```

    expect(response.body.user).to.have.property('name')
  })

  // 4. Verify UI changed
  cy.contains('Welcome').should('be.visible')
})
})

```

Key things:

- `cy.intercept()` before `cy.visit()`
- `cy.wait('@alias')` to grab that call
- `interception.request` & `interception.response` for assertions

## 4.2 Verify payloads in API tests (with `cy.request()`)

You already saw this, but here's a more focused example:

```

it('verifies response payload for GET /api/users', () => {
  cy.request('GET', `${baseUrl}/api/users`).then((response) => {
    // Status
    expect(response.status).to.eq(200)

    // Body content
    expect(response.body).to.be.an('array')
    response.body.forEach((user) => {
      expect(user).to.have.all.keys('id', 'name', 'email')
    })
  })
})

```

---

## 5. Putting It All Together – Mental Model

When to use what?

- `cy.intercept()`

- I'm doing **UI tests**
  - I want to:
    - Spy on network calls (see what UI sends/receives)
    - Mock backend responses
    - Make tests deterministic
  - **cy.request()**
    - I'm doing **pure API tests**
    - I don't care about the page / DOM
    - I want to:
      - Verify status codes
      - Check response JSON structure/content
      - Test multiple endpoints quickly
- 

## 6. Suggested Practice for You

Because you're a beginner, here's a small practice plan:

### 1. REST API practice

- Create a simple Express API with `/api/users` (GET, POST).
- Write `rest_api.cy.js` using `cy.request()` for these endpoints.

### 2. UI + REST

- Build a small React page that:
  - On load: `GET /api/users` and displays list

- Has a form to `POST /api/users`
- Write Cypress UI tests:
  - Use `cy.intercept('GET', '/api/users')` with mock data
  - Verify UI shows the mocked users
  - Use `cy.intercept('POST', '/api/users')` to verify correct request body

### 3. GraphQL

- Use any simple GraphQL endpoint in your demo app
- Intercept `POST /graphql`
- Check `operationName` and `variables`
- Reply with mocked `data`

---

If you want, next step I can:

- Give you a **tiny Express + React example** just for this topic, and
- Write the exact Cypress tests for that app (so you can copy-paste and run).