**Spring Frame Work – Test Driven Development (TDD)**

**November 2022**

Practices on how TDD works

REST Backend server application built using Spring Boot and web client using React

Requirements implemented one at a time

Requirements defined and implementation guided

Program contains:

* User sign up functionality
* React router and Redux for browser functionality and layout
* Users are listed and contents browsed
* User profile details can be updated

TDD – Write test before writing any production code



**User Sign Up Page - backend**

* User sign up functionality implemented
* Password of user hashed and saved to database

**Sign Up Form – frontend**

How states works in React

**Post User – backend**

* Test code written before writing any production code
* In Spring, HTTP annotations with controller will be used
* Junit – used to run tests
* Test classes and methods used
* Different types of tests – Unit Tests and Integration Tests
* Unit Tests – focus on small amount of code (i.e. method)
* Integration tests – entire web application to run – real user scenario
* Random port used for testing
* Test to be run in a controlled environment - not reliant on external dependencies thus define profile to run tests
* Use test annotation for each test method
* Naming of test method is NB for maintaining and debugging
* Set profile validity parameters – username, displayname, pwd
* Lombok used for code regeneration vs. boiler plate code
* POST - /api/1.0/users – good idea to append API and version of end point may evolve and change
* Test to see if we’re receiving what is expected, called Assertion method – more fluid syntax



Test 1 works due to having end point of post user mapping

**Saving User to Database – backend**

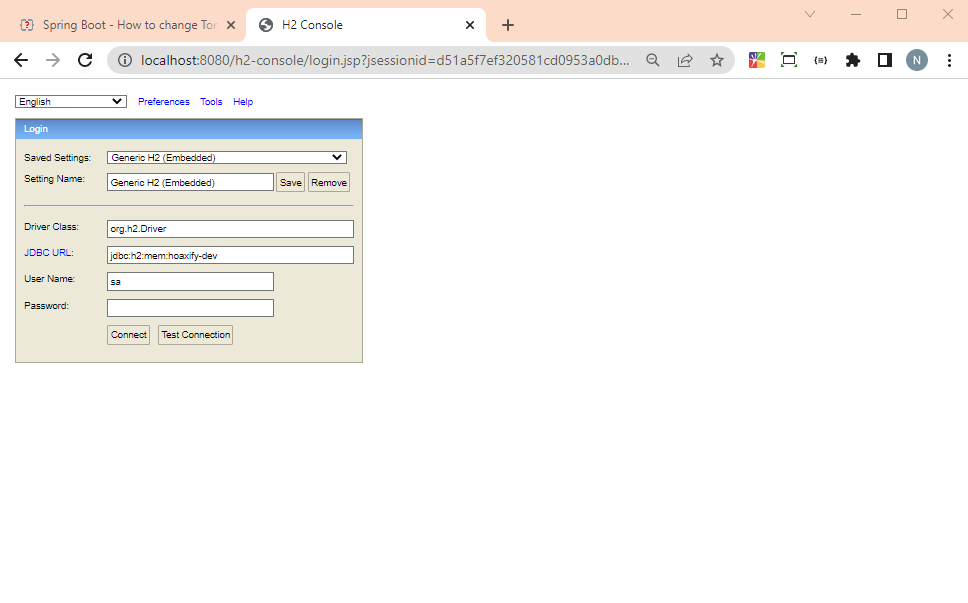
* API extracted to constant
* Assert if user stored in database
* Run query in H2 database
* JPA repo comes with most db query methods
* Update test class
* Tests are run in a random order
* New test runs before old one.
* All starts in a known state and back to original after test
* Clean up method clears db before each test

**Response Body – backend**

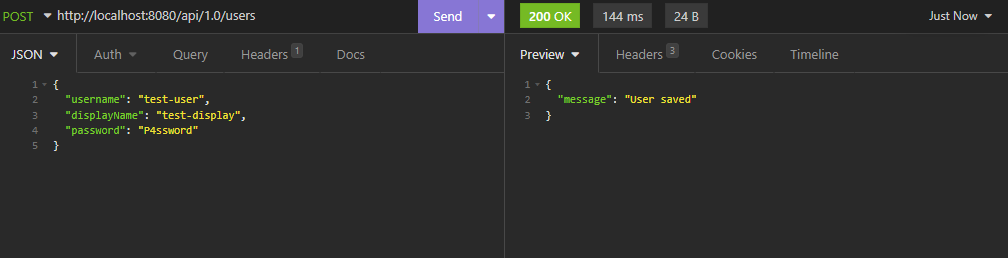
* Response type – generic response template
* TDD cycles assists in refactoring code
* Can check response letters being received
* Best way to write a test is to do a single assertion – focus on single requirement in each test
* Application requirements easier to pass if approach taken
* Easy to find out which one would fail with new requirements. If fail, debug

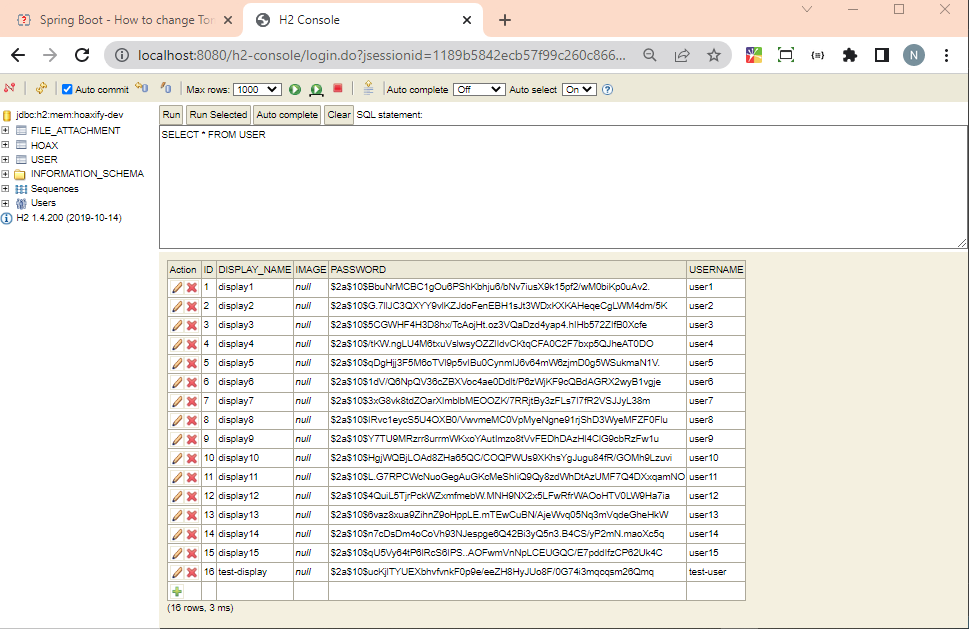
**Password Hashing – backend**

* Data stored in mem – h2
* When application is closed, data lost
* Mem – db to be stored in memory
* Dynamic JDBC URL in STS, input it into field in h2 console



* Create user object as JSON – used Insomnia REST testing tool

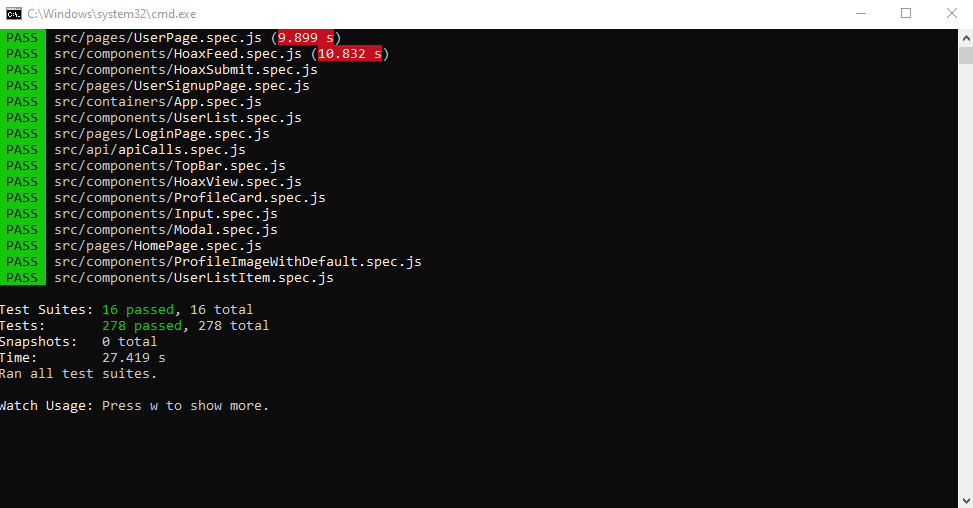




* Password hashed using creeped hashing algorithm
* Encoder dependency installed

**User Signup Page - frontend**

* Test project in react by running npm start and npm test
* Test files placed under components
* Spec.js or test.js
* Spec.js – specifies what the target component is doing
* Test.js –
* Import rendering aspect from @testing-library/react
* Import rendering aspect from @testing-library/jest-dom/extend-expect – additional expect functionality
* Describe function used to organizing tests
  + 2 parameters – description and function including test functions, layout 2nd parameter
* It function – component being testing, takes description and function to run
* Render user sign up page
* Expect
* Tests to be running automatically



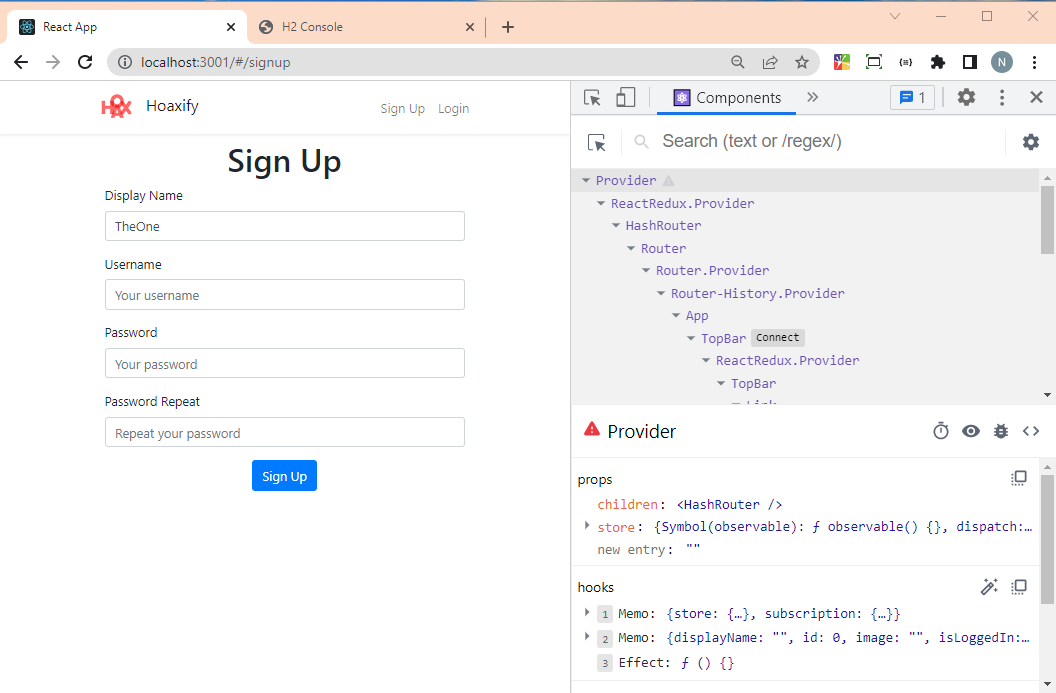
**Form Layout – front end**

* Input field required for testing
* Page with multiple sign up pages rendered one after another
* Clean document after running each test by using “before each” method from react testing library - beforeEach(cleanup)
* Import clean up function – no longer needed

**Handling Input Change – front end**

* Test group for interactions – user sign up sent to backend
* Simulate user put interaction with fireEvent
* Change function –field first param, change event second parameter
* Target field – value field
* Value to have own interstate
* State, json object, added to value component
* Must have callback function

**]**



**Click Handling – front end**

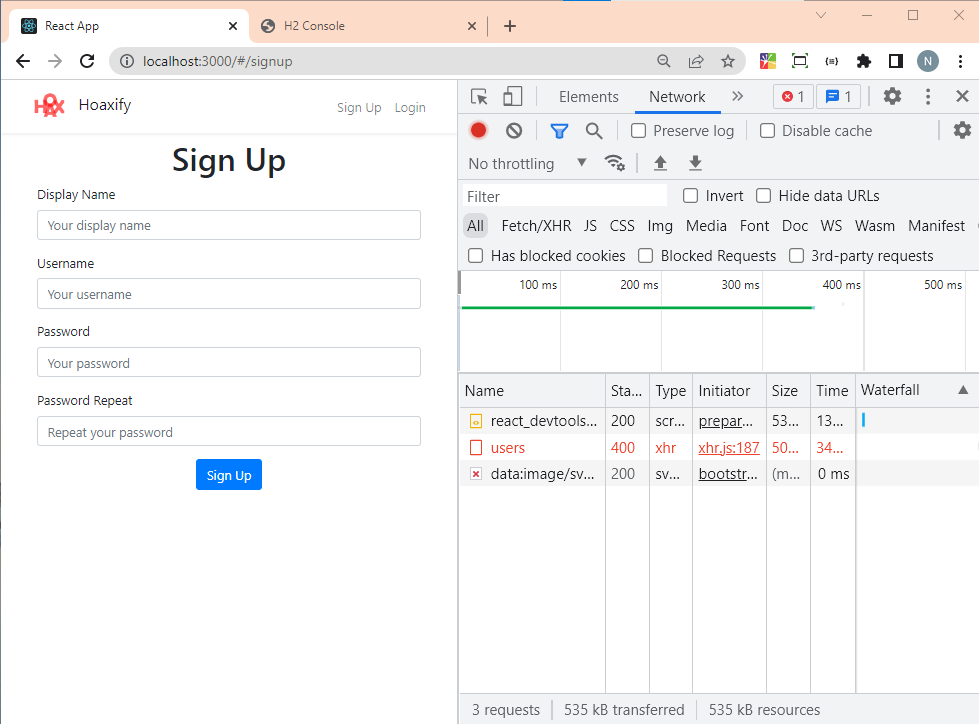
* Mock functions – instant HTTP request sent to backend
* Mock functions – actions objects
* Actions object – post signup(mock function)
* FireEvents click function – target component – button
* Refactor code - extract parts from tests, set variables using let keyword to assign actual components to them

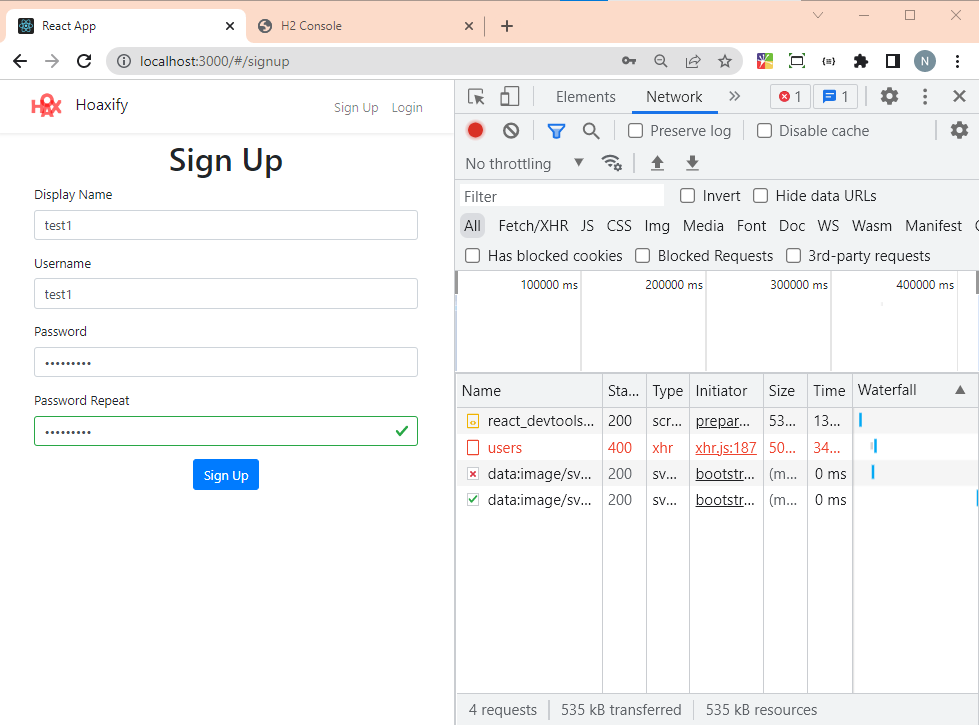
**Styling – front end**

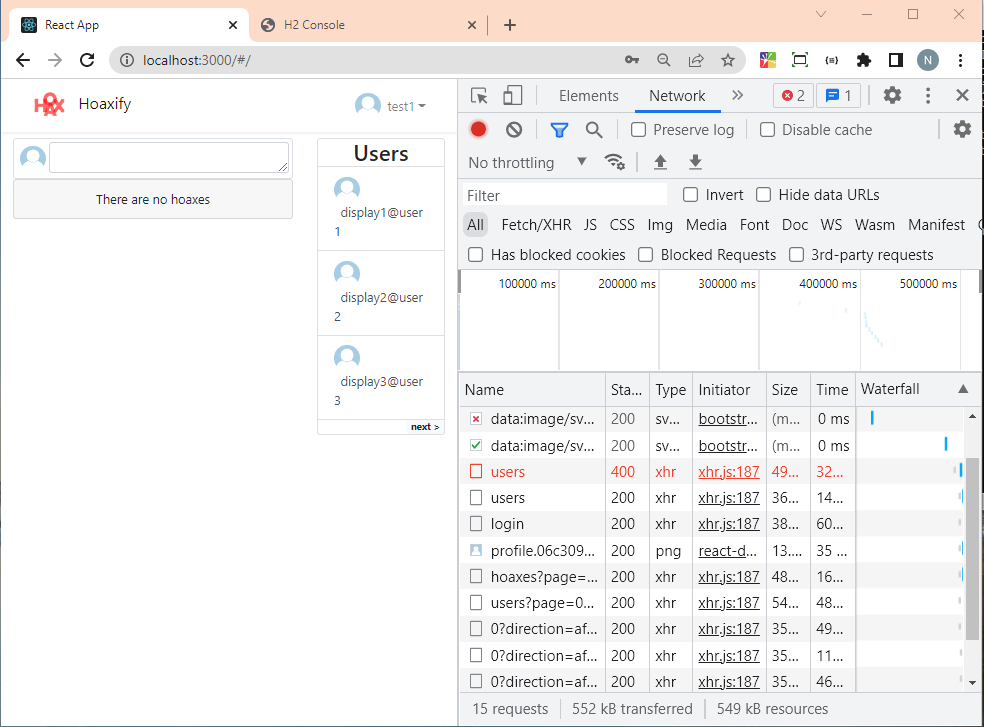
* No need to write tests for styles

**Sending requests to backend – frontend**

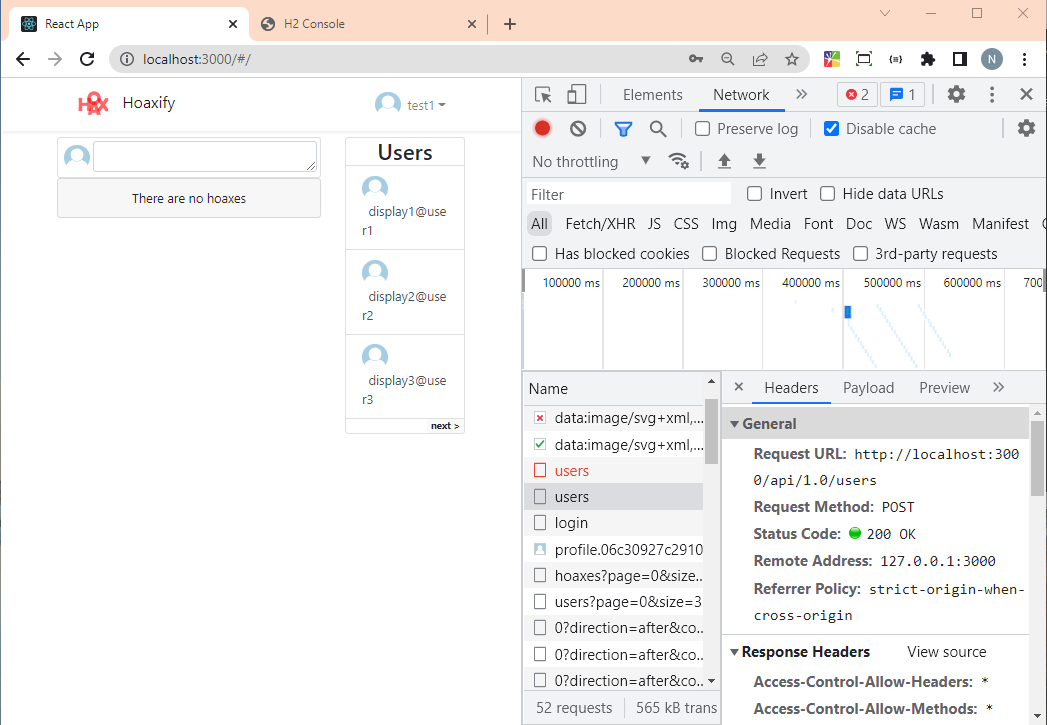
* Using axios
* apiCalls
* implement sign up function using post function takes 2 parameters
* Developer tools
* Sign up should be passed as prop in index.js
* Each API request sent to webpack, webpack to proxy to backend

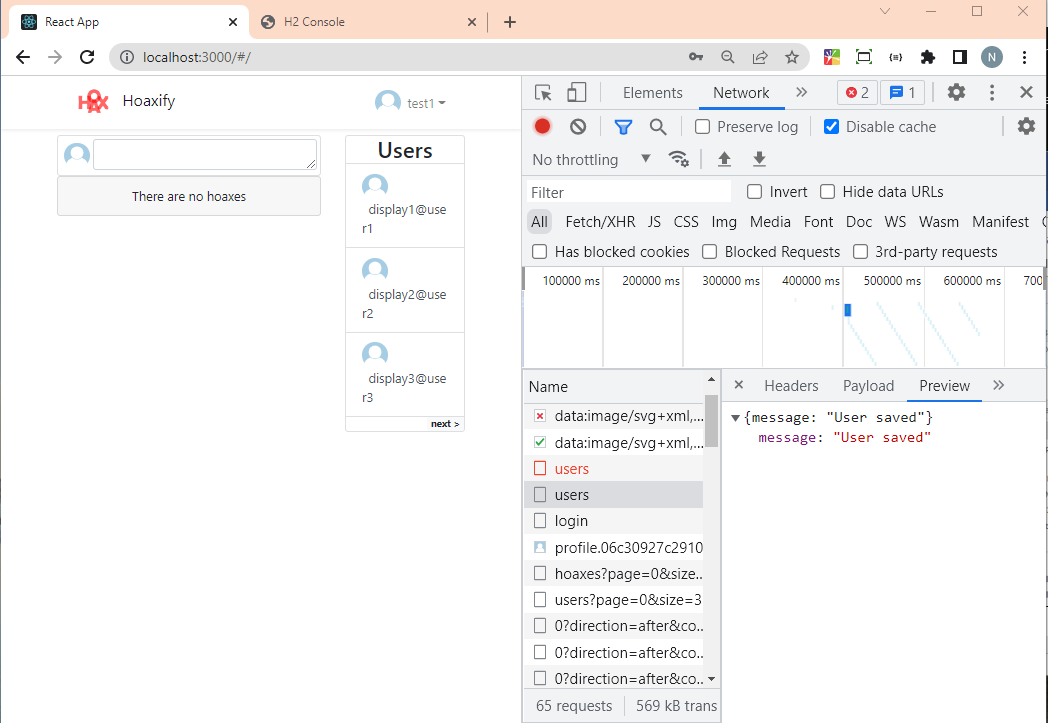
****

****

****

Response from backend received

****

****

Successfully sent sign up request to backend

**Progress Indicator - frontend**

* User cannot send another HTTP request whilst another is logged in
* Disable sign up functionality
* Slow down network speed – timeout function
* Spinner component used in button component
* Await keyword

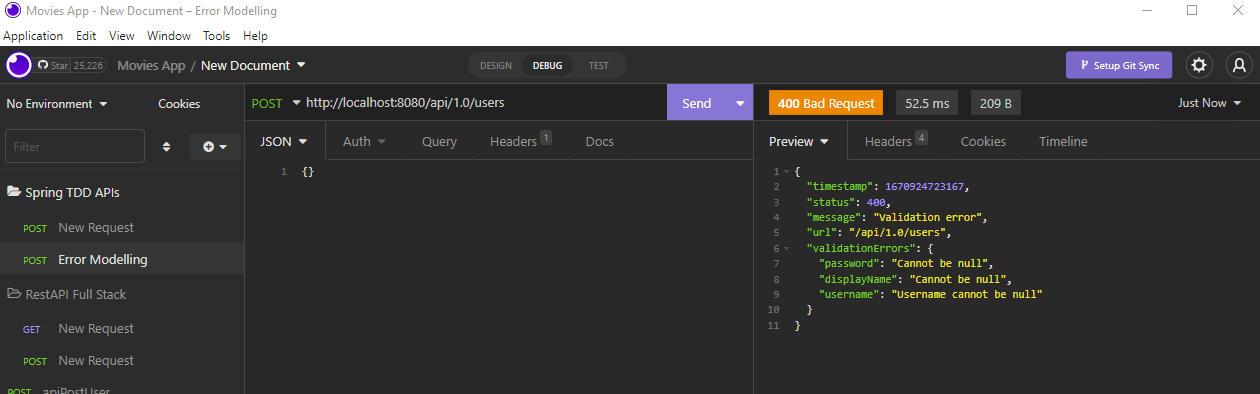
**Section 4 – Validation**

**2. User Validation – backend**

* User and User Controller Test
* Exceptions thrown when user credentials are not met
* Set user constraints – username, display name, password etc.

**3. Error Modelling – backend**

* Send empty response via Insomnia
* Response generated by Spring in validation
* API error received as response – ApiError
* Handle MethodArgumentNotValidException
* API error object seen in validation

****

**4. Error Internalization – backend**

* Validation message properties in Hibernate file
* Set custom messages in ValidationMessages
* Can accept language response in different languages
* Validation for user credentials