Project Part 9 - Server Side Validation

14/20 Points

8/3/2021

Attempt 1

REVIEW FEEDBACK
8/3/2021

Attempt 1 Score: **14/20**

View Feedback

Unlimited Attempts Allowed

Details

Due: Tuesday, August 3
Points: 20 points
Deliverables: `[Name]BookstoreOrder.war` uploaded to the CS5244 server
Resources: server-order.zip (https://drive.google.com/file/d/1zzJaxGbip8O3KkWAP32t7kxhYuzOjY14/view?usp=sharing)

Setup

Establish a new project `[Name]BookstoreOrder` according to our standard setup procedure. Copy the relevant files from your Validate project and make the relevant name changes. Before running your client module, you may have to install vuelidate, validator, and (if you are still using it for debugging purposes) vue-json-tree-view. If so, just run:

```
npm install --save vuelidate validator vue-json-tree-view
```

Run your project. It should behave the same as your last project.

To prepare your server for receiving bookstore orders, let's add some more dependencies for JSON processing.

Try Again

```
implementation group: 'com.sun.xml.bind', name: 'jaxb-core', version: '2.2.11'
implementation group: 'com.sun.xml.bind', name: 'jaxb-impl', version: '2.2.11'
implementation group: 'javax.activation', name: 'activation', version: '1.1.1'
```

This will let us receive and process JSON requests via HTTP POSTs.

1. Placing the Order from the Client

To place the order we will submit the cart and the customer form details to the server. The server will process the order and return with details of the placed order, or an error condition. On success, we must clear the cart and land on the confirmation page. When an error occurs will display a "please try again" message.

In `Checkout.vue` the `submitOrder` method will dispatch a `placeOrder` action to the store. We will either set the `checkoutStatus` to `OK` or `SERVER_ERROR` based on the server response.

Replace the setTimeout statement in the submitOrder method with the code below.

```
this.$store
   .dispatch('placeOrder', {
     name: this.name,
     address: this.address,
     phone: this.phone,
```

Try Again

```
        ccExpiryYear: this.ccExpiryYear
      })
      .then(() => {
        this.checkoutStatus = 'OK'
        this.$router.push({ name: 'confirmation' })
      })
      .catch(reason => {
        this.checkoutStatus = 'SERVER_ERROR'
        console.log('Error placing order', reason)
      })
```

Add the following action to your Vuex store to place an order.

```
placeOrder({ commit, state }, customerForm) {
  return ApiService.placeOrder({
    cart: state.cart,
    customerForm: customerForm
  }).then(() => {
    commit('CLEAR_CART')
  })
},
```

Note that we are sending an order (the cart and the customer form) to the `ApiService`. For some clarity we are using

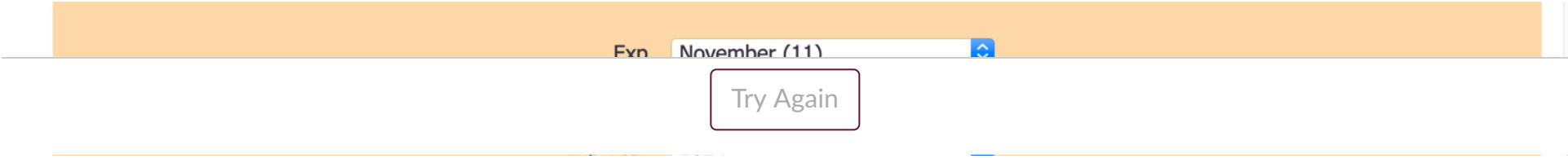
Try Again

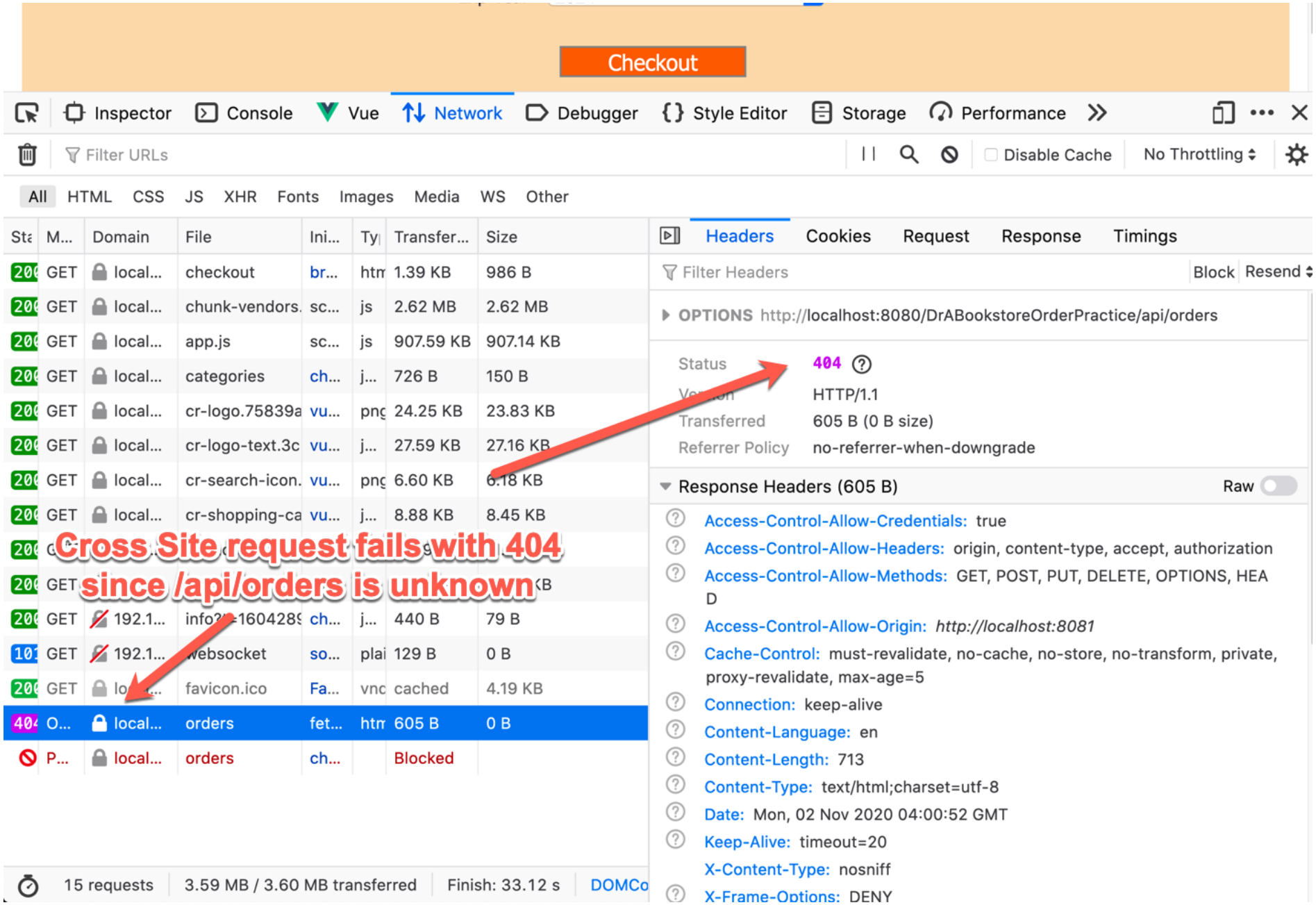
Add the following placeOrder function to ApiService.

```
placeOrder(order) {
    console.log("POSTing to " + `${apiUrl}/orders`);
    return fetch(`${apiUrl}/orders`, {
      method: "POST",
      body: JSON.stringify(order),
      headers: {
        "Content-Type": "application/json"
      }
    }).then(stream => {
      if (stream.ok) {
        return stream.json();
      }
      throw new Error("Network response was not ok.");
    });
  }
```

Note that we are changing the "fetch" call to use an HTTP POST verb rather than the GET by default, and that we are sending the entire order (cart and customer form) as one large JSON string in the body of the POST message.

Run your project, and verify using developer tools that the order is sent as the request payload You should find that the server responds with a 404 response, because the server side does not know about how to POST to /api/orders yet. See the screenshot below for an example of what to verify.




Checkout
Inspector
Console
Vue
Network
Debugger
Style Editor
Storage
Performance
Filter URLs
Disable Cache
No Throttling
All HTML CSS JS XHR Fonts Images Media WS Other
Sta M... Domain File Ini... Ty Transfer... Size
200 GET local... checkout br... htm 1.39 KB 986 B
200 GET local... chunk-vendors. sc... js 2.62 MB 2.62 MB
200 GET local... app.js sc... js 907.59 KB 907.14 KB
200 GET local... categories ch... j... 726 B 150 B
200 GET local... cr-logo.75839a vu... png 24.25 KB 23.83 KB
200 GET local... cr-logo-text.3c vu... j... 27.59 KB 27.16 KB
200 GET local... cr-search-icon. vu... png 6.60 KB 6.18 KB
200 GET local... cr-shopping-ca vu... j... 8.88 KB 8.45 KB
Cross Site request fails with 404
since /api/orders is unknown
200 GET 192.1... info 160428 ch... j... 440 B 79 B
101 GET 192.1... websocket so... plai 129 B 0 B
200 GET favicon.ico Fa... vnc cached 4.19 KB
404 O... local... orders fet... htm 605 B 0 B
P... local... orders ch... Blocked
15 requests 3.59 MB / 3.60 MB transferred Finish: 33.12 s DOMCo
Headers Cookies Request Response Timings
Filter Headers Block Resend
OPTIONS http://localhost:8080/DrABookstoreOrderPractice/api/orders
Status 404
HTTP/1.1
Transferred 605 B (0 B size)
Referrer Policy no-referrer-when-downgrade
Response Headers (605 B) Raw
Access-Control-Allow-Credentials: true
Access-Control-Allow-Headers: origin, content-type, accept, authorization
Access-Control-Allow-Methods: GET, POST, PUT, DELETE, OPTIONS, HEAD
Access-Control-Allow-Origin: http://localhost:8081
Cache-Control: must-revalidate, no-cache, no-store, no-transform, private, proxy-revalidate, max-age=5
Connection: keep-alive
Content-Language: en
Content-Length: 713
Content-Type: text/html;charset=utf-8
Date: Mon, 02 Nov 2020 04:00:52 GMT
Keep-Alive: timeout=20
X-Content-Type: nosniff
X-Frame-Options: DENY

Try Again

2. Receiving the Order on the Server: Preparations

Let us get ready for receiving and validating the incoming order on the server.

Unzip the provided server-order.zip file in the server folder.

You should see these new folders in the business folder:

`cart`: contains model objects to capture cart information from placing the order

`order`: contains model objects and services to capture and validate an order

`customer`: a `CustomerForm` captures the customer details from the order

`api`: an exception handler to process validation failures is provided.

Add an instance of OrderService to ApplicationContext.

Following our singleton pattern from earlier, let's add `OrderService` to the `ApplicationContext` in the same way as we exposed `BookDao` and `CategoryDao`. This way `ApplicationContext` becomes the owner of all objects useful to the API code.

Add the following code to build the order service and let it cooperate with `bookDao` to the end of `ApplicationContext`'s constructor:

```
        orderService = new DefaultOrderService();
```

Try Again

The OrderService cooperates with a BookDao to place orders. This is how we arrange for classes owned by the application context to co-operate without knowing about the application context.

Add the following field to the ApiResource class.

```
    private final OrderService orderService = ApplicationContext.INSTANCE.getOrderService();
```

This provides access to the order service for the API resource.

We have finished establishing access to our `OrderService` for use in our `ApiResource` class. Let's continue server preparations to be able to validate an order once it is received.

3. Validating the Order on the Server: Preparations

We will make all validation errors throw an `ApiException.InvalidParameter` exception. This may look a little strange - in Java exceptions can have subclasses and it is a nice way to distinguish grouped exception types. Here for consistency, we will make all exceptions thrown from our API server be subclasses of ApiException - this lets us handle exceptions centrally and simplifies the code (lest we need many catch clauses for many exceptions).

Try Again

Change your ApiException class to include a static subclass `InvalidParameter`.

```
public class ApiException extends RuntimeException {

    public ApiException(String message) {
        super(message);
    }

    public ApiException(String message, Throwable cause) {
        super(message, cause);
    }

    public static class InvalidParameter extends ApiException {
        public InvalidParameter(String message) {
            super(message);
        }

        public InvalidParameter(String message, Throwable t) {
            super(message, t);
        }
    }
}
```

We will use this `InvalidParameter` exception for all validation issues in this project.

All of our ApiExceptions that are thrown from our Resource object can be managed using an `ExceptionMapper` class in Jersey. To see how this works, have a look at the `ApiExceptionHandler` class provided in the project resources in the `api`

Try Again

By default, we will return a 500 "internal server error" code. However, if it is our validation exception subclass, we send back 400 "bad request" error code. We will see an example of this shortly.

4. Receiving the Order on the Server

We are now ready to actually receive the order from the browser.

Add this placeOrder method to ApiResource.java

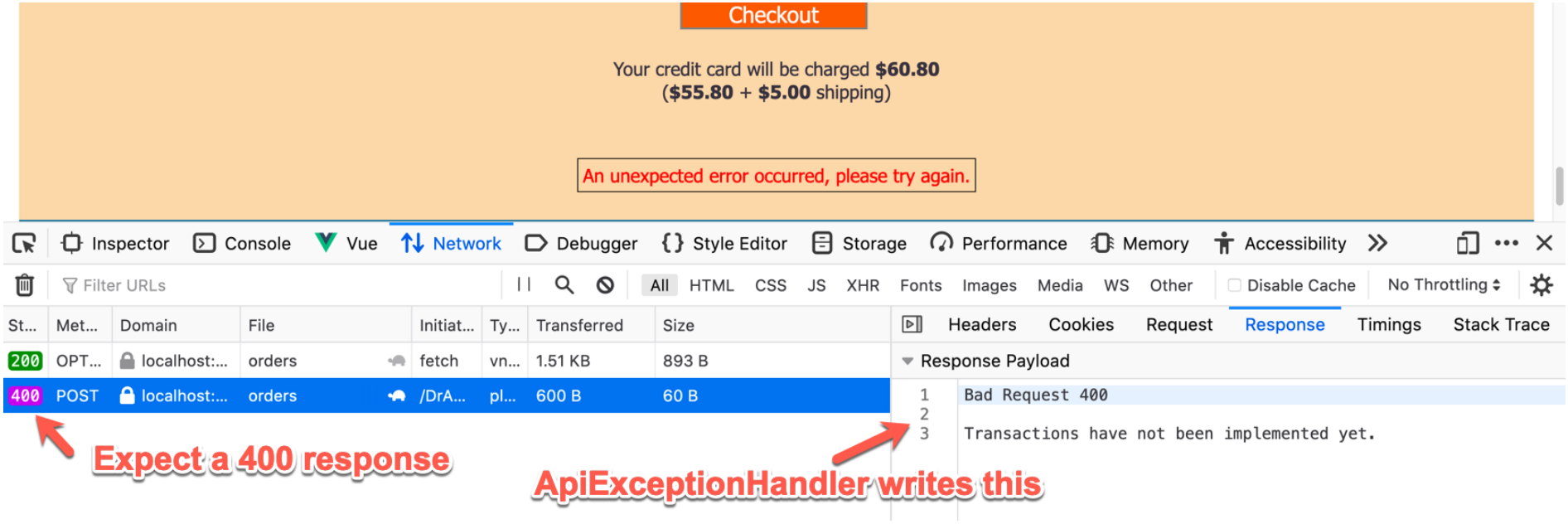
```
@POST
@Path("orders")
@Consumes(javax.ws.rs.core.MediaType.APPLICATION_JSON)
@Produces(javax.ws.rs.core.MediaType.APPLICATION_JSON)
public OrderDetails placeOrder(OrderForm orderForm) {

        try {
                long orderId = orderService.placeOrder(orderForm.getCustomerForm(), orderForm.getCart());
                throw new ApiException.InvalidParameter("Transactions have not been implemented yet.");

                // NOTE: MORE CODE PROVIDED NEXT PROJECT

        } catch (ApiException e) {
                // NOTE: all validation errors go through here
                throw e;
        } catch (Exception e) {
                throw new ApiException("order placement failed", e);
        }
}
```

Try Again

Notice that we have automatically taken the JSON from the client and turned it into an instance of an `OrderForm` class as the method parameter. The Jersey/JAXB framework we are using does that automatically. The `placeOrder` method both consumes and produces JSON objects. After an order has been placed, this method returns an `OrderDetails` object.

You might notice that for now we are placing the order and always failing with a validation error "Transactions have not been implemented yet". The next project will complete the implementation of transactions on the server side.

Run your project. When you submit a valid checkout request, you should expect to see a 400 ERROR code with the exception message printed out.



5. Server Validations

Try Again

You are required to complete server side validations in the `business.order.DefaultOrderService` class. The tasks are listed as `TODO` comments in that class. The `name` field and some cart validations have been provided as guidance.

The following customer form validations are required:

- All fields (including name and address): should be present and non-null and non-empty
- Name and address must have at least 4 and at most 45 characters in length.
- Phone: after removing all spaces, dashes, and parens from the string it should have exactly 10 digits
- Email: should not contain spaces; should contain a "@"; and the last character should not be "."
- Credit card number: after removing spaces and dashes, the number of characters should be between 14 and 16.
- Expiration date: the month and year should be the current month and year or later

The following shopping cart validations are required:

- The cart should contain at least one item
- Each cart item has quantity of books between 1 and 99
- Each cart item's price should match the price for the item from the database
- Each cart item's category should match the category for the item from the database.

The validations should each have a distinct short clear message that displays upon invalid input. You should expect to see 'Transactions have not been implemented yet' with a 400 code with valid input.

Testing Your Validations

Try Again

It may have occurred to you that it will be hard to actually test the validations because we will not be able to easily enter invalid data in our browser client.

However, we can test how our server responds in isolation from the browser using any one of the wide array of API testing tools. For this project, we can actually stick with IntelliJ's REST console.

In IntelliJ, open the server/test/resources/test.http file.

Using http files we can create and send HTTP requests with data of our choice on demand and see the responses.

You have been given a sample starting request for your API categories. Change the `[Name]` in the first line to your bookstore name. Click on the green triangle to the left of the "GET". You should see the nicely formatted JSON response with categories come back.

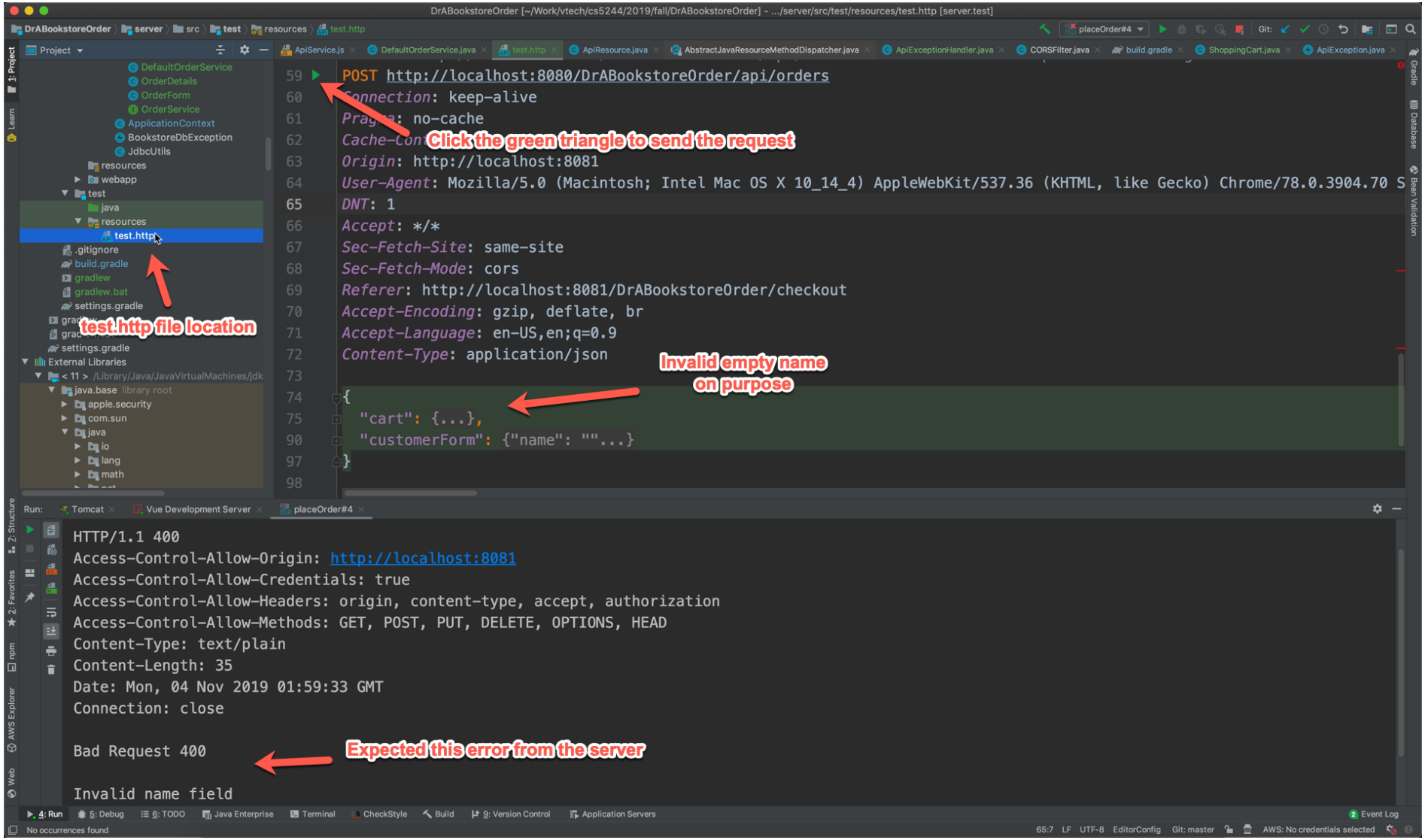
There is a convenient way to easily capture the HTTP request for placing an order.

Follow the next steps in your browser to capture the placeOrder request

- Open the developer tools to the network tab
- Browse to your website in the browser, and clear your cart if it has any items.
- Add one book to the cart, and proceed to checkout.
- Clear the network tab of any existing requests using the trash bin.
- Fill in the form field with valid data, and checkout.
- No matter the response, we can capture the request:
  - Find the POST (not OPTIONS) request to `/api/orders` second to the top of your network
  - Right click, and choose Copy > Copy as cURL
- Paste the clipboard into the bottom of your `test.http` file underneath the `###` separator.

Try Again

Here is an example of the name validation failing.



Style Requirements

Try Again

While this project has no specific style requirements beyond what was asked for in the previous projects, please be aware that Dr. K will be reviewing your entire site for style. Therefore, if you have not fixed anything stylistically that you have been meaning to fix, please do so for this project. All style requirement (implicit or explicit) from past projects apply, so it might be good to review them. Also, if Dr. K made comments to your Project 3, please make sure all of those comments have been addressed, even if Dr. K did not take off points in the comment. Finally, you should also use this opportunity to modify you web site based on any comments in the peer reviews that resonated with you.

---

http://cs5244.cs.vt.edu:8080/GraceBookstoreOrder/

Try Again