Java Spring / Spring Boot Security

Spring Boot Application:

AppSecSpringSecurityPart1

Agenda

- Overview
- Mass Assignment Insecure Binder Configuration
- Input Validation Controls
  - Request Body
  - Request Parameters (path variables & query parameters)
  - Custom Validation
- Access Controls

Overview of the Spring Boot project

Overview

- The purpose of this document is to give a “walkthrough” of the security concepts configured in the AppSecSpringSecurityPart1 Spring Boot Project.
- The main security related concepts implemented in this project are:
  - Mass Assignment: Insecure Binder Configuration
  - Input Validation Controls
  - Access Controls
- Each Class/method used in the Spring Boot project contains comments explaining what functionality it is performing.
- This document will also have screenshots of the main ideas for this project, so it is not required to run it locally on our own computer to see how it all works.

How to download GitHub repo and import project into Eclipse (Optional)

- **Download GitHub repository**:
- Go to the repository -> Click on the “Code” drop down button (next to the “About” section) -> Choose “Download ZIP”

- **Import the code into Eclipse**:
- Open Eclipse IDE
- Ensure Spring Tool Suite is installed. Click on – Help -> Eclipse Market Place -> Search for “Spring Tools Suite” -> Select & Install “Spring Tools 4 (aka Spring Tool Suite 4)”
- Click on - File -> Import -> Maven -> Existing Maven Projects
- In the “Root Directory” section, browse to the folder that holds the code for the Spring Boot project
- Click on Finish
- Now the project should be available on the Eclipse IDE UI, and we can run it as a “Spring Boot App”.

Configuring the ‘application.properties’ file

- Overview:
- The file is created by default under the /src/main/resources folder.
- This file is used to configure the port number and the in memory h2-database used for the Spring Boot application.
- The file has the configuration details and comments.

Creating the Model/Entity Class

- Overview:
- The Student.java Class will be created under the “com.example.demo.model” package.
- The model/entity class is used to define the components of the application.
  - Answers the “What is a Student?” question.
- Using the @Entity annotation, the Class will be treated as a database table in the h2-database. The fields in the Class are essentially the columns in the table.
- The Class has more details and comments.

Creating the Repository Class

- Overview:
- The StudentRepo.java Class will be created under the “com.example.demo.repo” package.
- The purpose of the class is to allow us to perform CRUD operations on the Student Model Class.
- The JpaRepository Class has many methods that can be used to perform these CRUD operations.
- The Class has more details and comments.

Creating the Controller Class

- Overview:
- The StudentRestController.java Class will be created under the "com.example.demo.controller" package.
- A Controller Class usually contains most of the business logic of the application. It is used to interact with the Model/Entity Classes, and it determines what response ("View") to send back to the end user when a request is made to a particular endpoint/page.
- Currently there are 5 methods in this Class used for creating, retrieving and deleting a Student Object.
- The Class has more details, comments and test data to use.

Implementing Security Controls

- The next sections will cover the security related controls configured in the Spring Boot Project.
- The Classes/methods in the project will have comments that explain the security updated changes that were made.
- These comments will be prefixed with the words – “**Security Update:**”

Mass Assignment: Insecure Binder Configuration

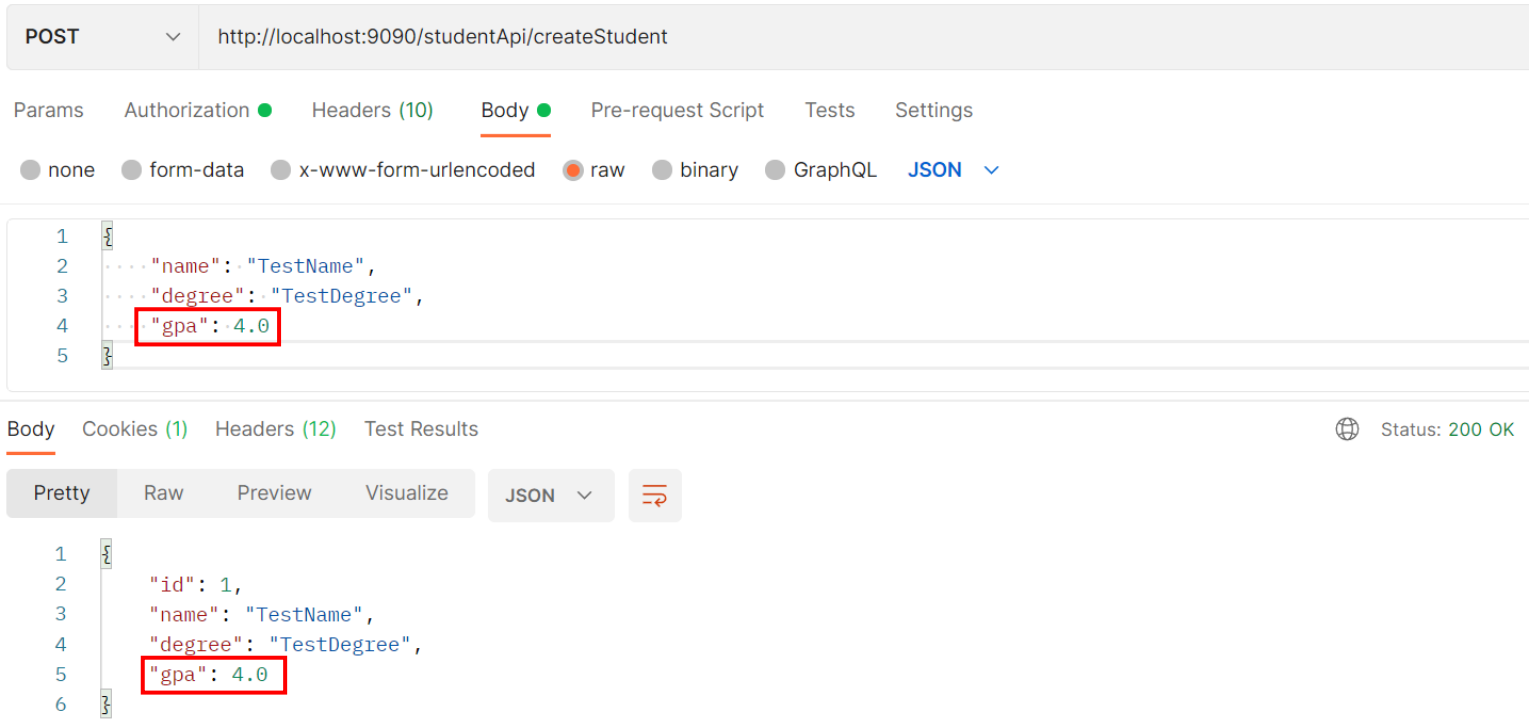
Mass Assignment: Insecure Binder Configuration (PoC)

- **Issue**: Frameworks, like Spring, allow a model object to be automatically instantiated and populated with HTTP request parameters whose names match an attribute of the class to be bound. If any of these fields are used in a security critical context, an attacker can change these values and bypass security controls.
- **Solution**: We can use the @JsonIgnore annotation to disallow sensitive fields from being automatically bound to the model Class by HTTP requests, when the @RequestBody annotation is being used to bind the HTTP request body to an Object. Another general way to prevent mass assignment is to use a different binding Class which does not include any of the sensitive fields we don't want changed from the HTTP request.
- Check out the following Classes:
  - Student.java (private double gpa – field)
  - StudentRestController.java (createStudent() – method)
- **Resources**:
- https://cheatsheetseries.owasp.org/cheatsheets/Mass_Assignment_Cheat_Sheet.html
- https://stackoverflow.com/questions/46840174/what-is-the-solution-for-mass-assignment-insecure-binder-configuration-vulnerab
- https://dev.to/wakeupmh/how-to-avoid-insecure-binder-configuration-in-java-2m7d

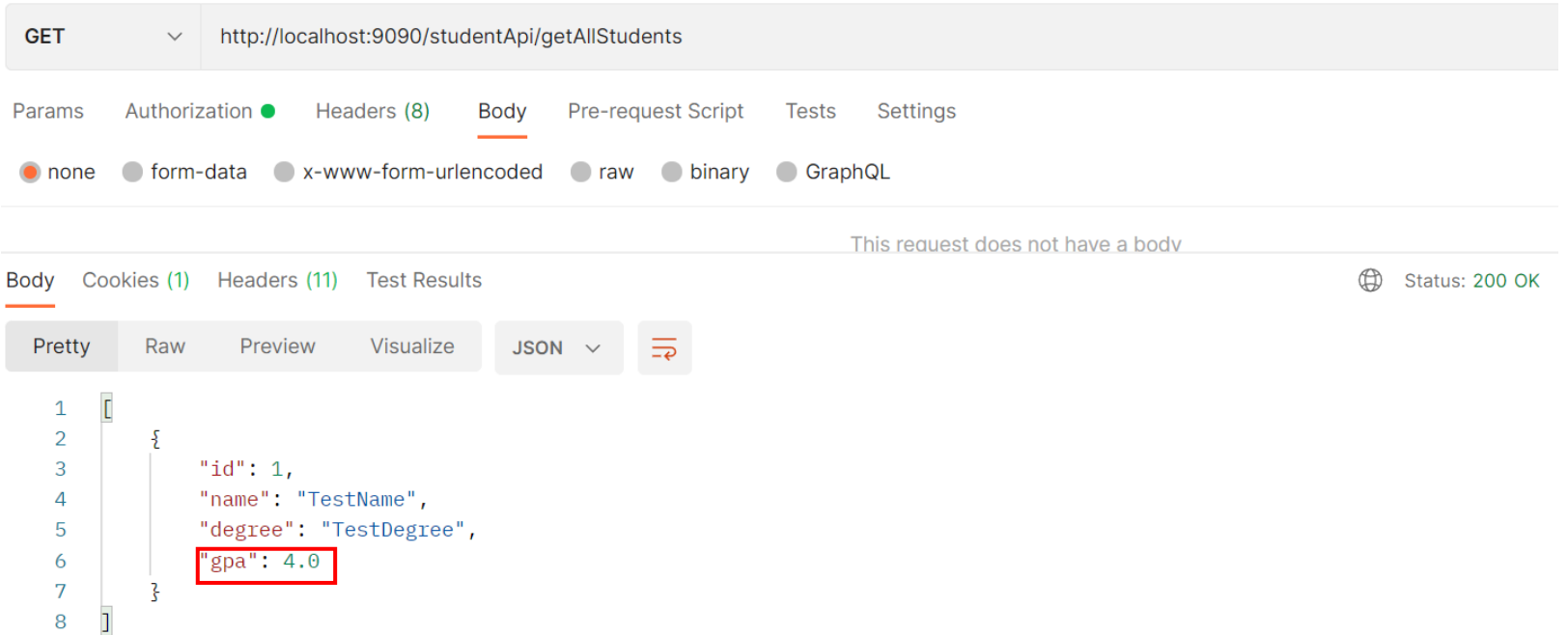
Vulnerable Configuration

- These are the screenshots of the results from **not applying** the @JsonIgnore annotation on the “private double gpa;” field in the Student.java Class.

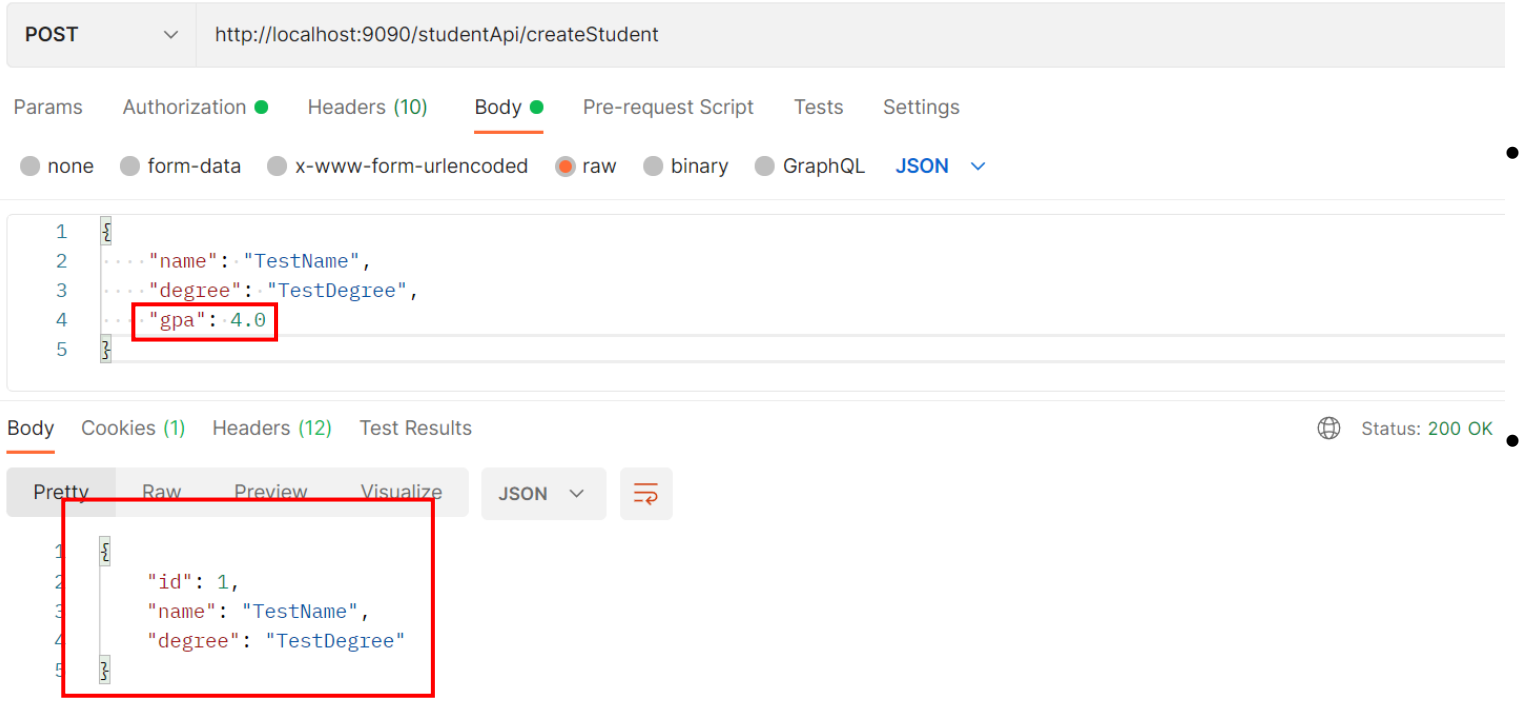
- When submitting a POST request to create a Student Object, the application is allowing the “gpa” field to be bound to the Object using the request.



- When retrieving all the Students using the /getAllStudents endpoint, we can see that the “gpa” field in the response, returned the same value that was used to create the Student in the POST request^.



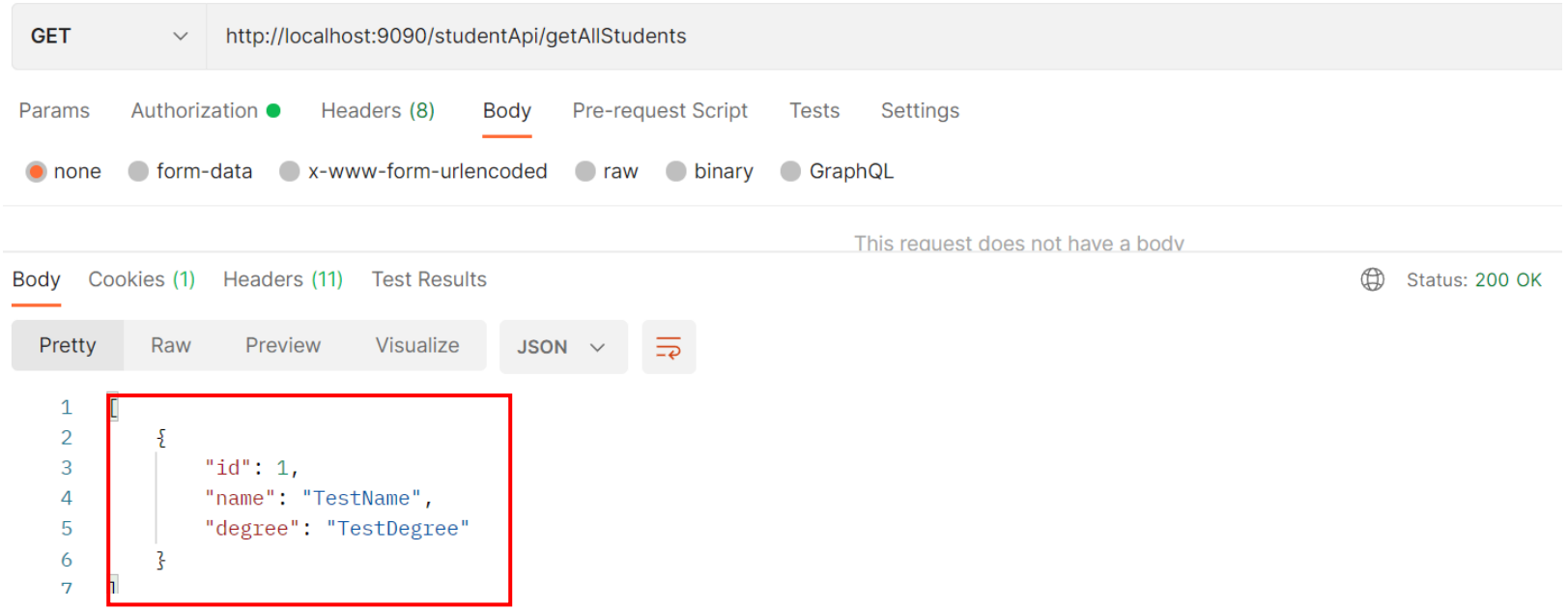
Secure Configuration



- These are the screenshots of the results from **applying** the @JsonIgnore annotation on the “private double gpa;” field in the Student.java Class.

- When submitting a POST request to create a Student Object, the application is not allowing the “gpa” field to be bound to the Object using the request.

- When retrieving all the Students, we can see that the “gpa” field is not visible in the response because the field was not allowed to be bound when the POST request was submitted. This is because of the @JsonIgnore annotation being applied to the “gpa” field in the Student Class.



Input Validation Controls

Input Validation Controls

- Spring/Spring Boot framework makes it easy to implement input validation on parameters coming from a request body (POST/PUT), path variables and query parameters, by inserting validation annotations constraints.
- For these annotation constraints to work, we need to add the following dependency in the pom.xml file:

```
<dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-validation</artifactId>
</dependency>
```

- We can also use Java custom validation methods to validate the incoming data, instead of using the validation annotation constraints.

Validating JPA Entities – Persistent Layer (Model/Entity) Classes – Annotation Constraints

- Input validation should be implemented on the "Model/Entity" Classes, but it should not be the only place where validation takes place. The reason why is because this validation will only trigger anytime, we use the repository to interact with an "Object" in the database. However, if the application is using data in any way before the repository touches it, then the application will be handling unvalidated data.
- **Implement Input Validation Controls on the Model Class**:
  - Student.java Class – all the fields have proper input validation annotation constraints, such as an allow list of characters using the @Pattern annotation.
  - Now anytime the repository is creating/updating a Student Object into the database, these constraints will be used to validate the data. If any requests violate these constraints, then the request will be rejected.
- **"Misconfigurations"**:
  - If the application is using the data for any business logic before the repository touches it, then it will be handling unvalidated input.
- **Proof of concept**:
  - The only validation constraints implemented for this PoC, is on the fields in the Student.java Class.
  - In the StudentRestController.java Class under the createStudent() method, there are 2 lines of code that are called before the repo saves the Student Object in the database:
    - System.out.println(student.getName());
    - System.out.println(student.getDegree());

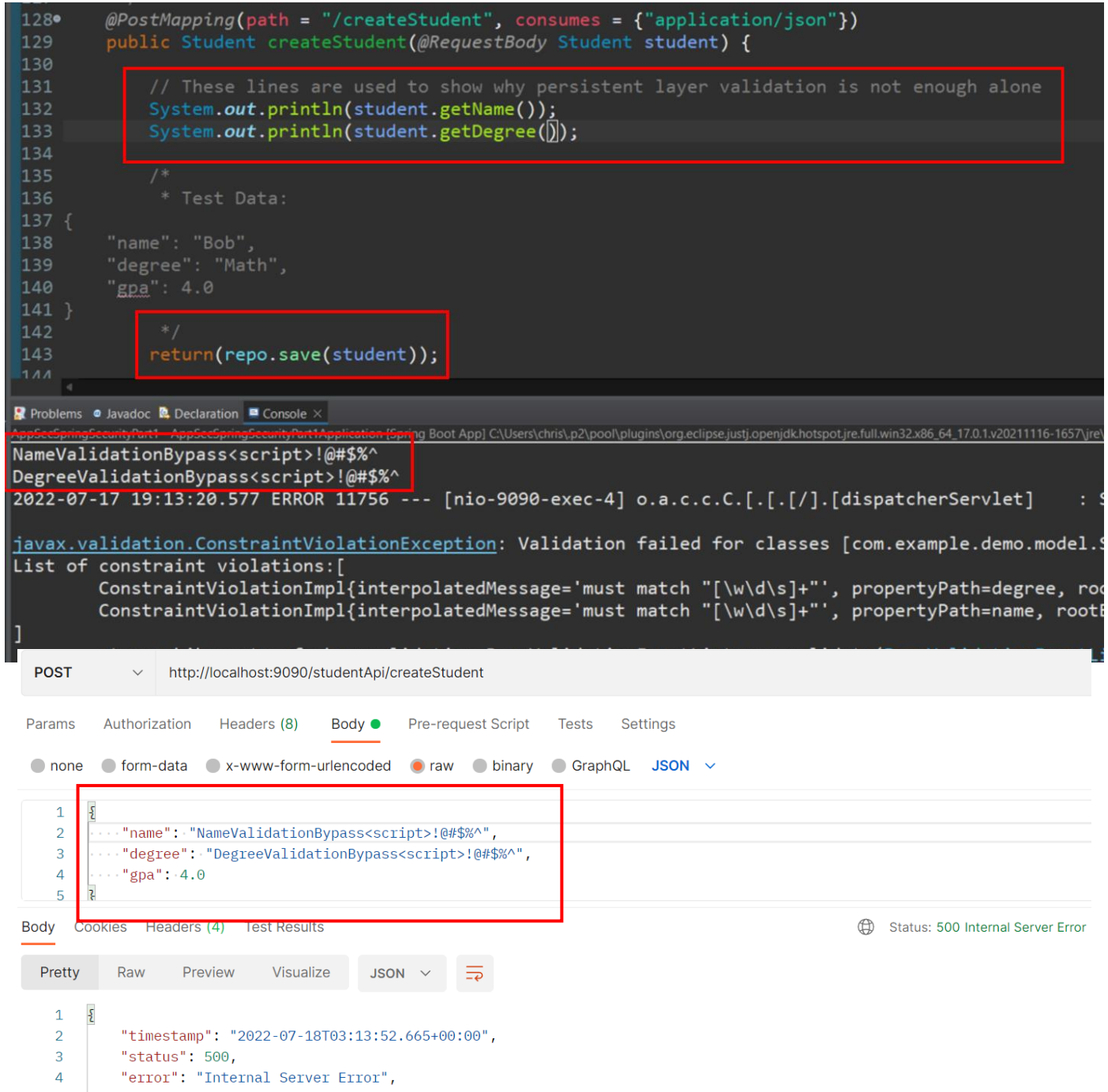
- The fields (name & degree) in the Student.java (Model/Entity) Class, have validation constraints applied to their initialization.

- Only letters, digits and spaces are allowed.

Student.java

```
46       *
47       * See: createStudent() method in the StudentRestController.java Class
48       */
49
50      /*
51       * The @Id annotation is used to define the primary key of the table.
52       * The @GeneratedValue means that Spring will auto increment this value whenever
53       * a new Student Object is created.
54       */
55      @Id
56      @GeneratedValue
57      @Min(0)
58      private int id;
59
60      @Pattern(regexp = "[\\w\\d\\s]+")
61      @NotEmpty
62      private String name;
63
64      @Pattern(regexp = "[\\w\\d\\s]+")
65      @NotEmpty
66      private String degree;
67
68      /*
```

- Submit a POST request to the /createStudent endpoint and include special characters in the name and degree fields.
- Here we can see that the 2 lines of print statements were executed successfully before the persistent layer (Model/Entity) validation was initiated.
- **** The next sections cover how to implement input validation as soon as the data enters the application, so that the application does not use any unvalidated data in any business logic.



Validating a Request Body – Annotation Constraints

- For POST/PUT requests, it is common to pass a JSON payload in the body. Spring automatically maps the JSON payload to a Java Object. This is implemented by using the **@RequestBody** annotation.
- In order to properly validate the request body of an incoming HTTP request before handling the data, the following needs to be implemented:
    1. The Model/Entity Class that the request body is mapping to, needs to have proper validation annotation constraints on its fields.
    2. The @Valid annotation needs to be applied to the parameter that is mapped to the @RequestBody annotation on the method.

- **Proof of concept**:
    1. The fields in the Student.java Class need to have proper input validation annotations.
    2. The createStudent() method in the StudentRestController.java Class, needs to use the @Valid annotation with the complex parameter type of -> Student.
- ***** If either one of those 2 requirements are not implemented, then the request body will not be validated properly, and the "validation" is considered broken/misconfigured. Unless custom validation is implemented before the data is used.**

Secure Configuration

- Just like the last section, the Student.java Class has proper validation annotation constraints, which is required here.

- As we know, this validation will only trigger once the repository touches this model class.

```
47      */
48•     @Id
49      @GeneratedValue
50      @Min(1)
51      private int id;
52
53•     @Pattern(regexp = "[\\w\\d\\s]+")
54      @NotEmpty
55      private String name;
56
57•     @Pattern(regexp = "[\\w\\d\\s]+")
58      @NotEmpty
59      private String degree;
60
61•     /*
```

- Now in order for Spring to validate the request body before the application uses the data for any business logic, we need to apply the @Valid annotation to the Student type parameter.

```
127     */
128•    @PostMapping(path = "/createStudent", consumes = {"application/json"})
129     public Student createStudent(@RequestBody @Valid Student student) {
130
131         // These lines are used to show why persistent layer validation is not enough alone
132         System.out.println(student.getName());
133         System.out.println(student.getDegree());
134
135         /*
136          * Test Data:
137 {
138     "name": "Bob",
139     "degree": "Math",
```

- By applying the @Valid annotation here, we are telling Spring to validate the input before doing anything else. ***This works correctly if the Model Class – Student – also has proper validation constraints on all it's fields.

Secure Configuration

```
127      */
128      @PostMapping(path = "/createStudent", consumes = {"application/json"})
129      public Student createStudent(@RequestBody @Valid Student student) {
130
131          // These lines are used to show why persistent layer validation is not enough alone
132          System.out.println(student.getName());
133          System.out.println(student.getDegree());
134
135          /*
```

```
Problems  Javadoc  Declaration  Console
AppSecSpringSecurityPart1 - AppSecSpringSecurityPart1Application [Spring Boot App] C:\Users\chris\.p2\pool\plugins\org.eclipse.justj.openjdk.hotspot.jre.full.win32.x86_64_17.0.1.v20211116-1657\jre\bin\ja
:2:04:17.391  INFO 14844 --- [nio-9090-exec-2] o.a.c.c.C.[Tomcat].[localhost].[/]       : Initializing Sp
:2:04:17.391  INFO 14844 --- [nio-9090-exec-2] o.s.web.servlet.DispatcherServlet        : Initializing Se
:2:04:17.392  INFO 14844 --- [nio-9090-exec-2] o.s.web.servlet.DispatcherServlet        : Completed initi
:2:04:17.486  WARN 14844 --- [nio-9090-exec-2] .w.s.m.s.DefaultHandlerExceptionResolver : Resolved [org.s
```

- Now if we submit a POST request with invalid data, the application will respond with a 400 Bad Request and the 2 println() statements do not execute this time.
- This is the way the input validation should work if it is configured correctly.

POST http://localhost:9090/studentApi/createStudent

Params Authorization Headers (8) Body Pre-request Script Tests Settings

none form-data x-www-form-urlencoded raw binary GraphQL JSON

```
{
    "name": "NameValidationBypass<script>!@#$%^",
    "degree": "DegreeValidationBypass<script>!@#$%^",
    "gpa": 4.0
}
```

Body Cookies Headers (4) Test Results

Status: 400 Bad Request

Pretty Raw Preview Visualize JSON

```
{
    "timestamp": "2022-07-18T03:04:17.494+00:00",
    "status": 400,
    "error": "Bad Request",
    "path": "/studentApi/createStudent"
}
```

Validating Request Parameters - Path Variables/Query Parameters – Annotation Constraints

- Validating request parameters using annotation constraints is different than for a request body:
  - We need to apply the validation constraint annotations directly in the request parameters on the methods.
  - We also need to add Spring's **@Validated** annotation at the Class level, where the validation annotations are being set.

- **Proof of concept**:
  1. The getOneStudentById() method in the StudentRestController.java Class will contain the @Min(3) annotation for the path variable "StudentId".
  2. The @Validated annotation also needs to be applied at the Class level for the StudentRestController.java Class.
- ***** If either of these steps are missing, then Spring will not validate the input from the path/request parameters and the "validation" is considered broken/misconfigured. Unless custom validation is implemented before the data is used in any business logic.**

Secure Configuration

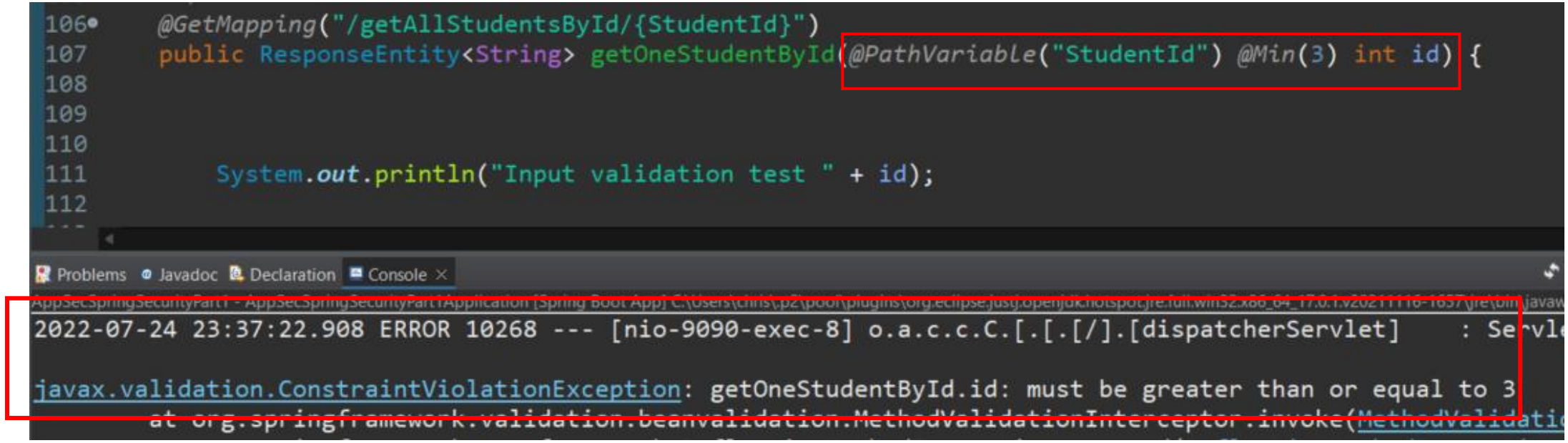
- The REST Controller is annotated with the @Validated at the Class level.
- And the request parameter “StudentId” has validation constraints – @Min(3).
- If either of these configurations are missing, then the input validation will not work at all.

```
52 @RestController
53 @RequestMapping("/studentApi")
54 @Validated
55 public class StudentRestController {
56
57     // The @Autowired is used here to create a Stude
58     @Autowired
59     private StudentRepo repo;
60
```

```
106     @GetMapping("/getAllStudentsById/{StudentId}")
107     public ResponseEntity<String> getOneStudentById(@PathVariable("StudentId") @Min(3) int id) {
108
109
110
111         System.out.println("Input validation test " + id);
112
113
```

Secure Configuration

- Here the @Min(3) annotation was applied along with the @Validated annotation at the Class level.
- The println() did not execute because Spring threw an error as soon as the invalid request was submitted. Any business logic will not execute unless the data conforms with the constraints.



Vulnerable Configurations

- Here we can see that the @Min annotation is not applied to the request parameter “StudentId”.
- So, the println() was executed.

```
106•    @GetMapping("/getAllStudentsById/{StudentId}")
107     public ResponseEntity<String> getOneStudentById(@PathVariable("StudentId") int id) {
108
109
110
111         System.out.println("Input validation test " + id);
112
```

Problems Javadoc Declaration Console ×

AppSecSpringSecurityPart1 - AppSecSpringSecurityPart1Application [Spring Boot App] C:\Users\chris\.p2\pool\plugins\org.eclipse.justj.openjdk.hotspot.jre.full.win32.x86_64_17.0.1.v2021111

```
2022-07-24 23:38:01.110  INFO 20036 --- [nio-9090-exec-1] o.a.c.c.C.[Tomcat].[localhost].[/]
2022-07-24 23:38:01.110  INFO 20036 --- [nio-9090-exec-1] o.s.web.servlet.DispatcherServlet
2022-07-24 23:38:01.111  INFO 20036 --- [nio-9090-exec-1] o.s.web.servlet.DispatcherServlet
Input validation test 2
```

```
105•    @GetMapping("/getAllStudentsById/{StudentId}")
106     public ResponseEntity<String> getOneStudentById(@PathVariable("StudentId") @Min(3) int id) {
107
108
109
110         System.out.println("Input validation test " + id);
111
```

Problems Javadoc Declaration Console ×

AppSecSpringSecurityPart1 - AppSecSpringSecurityPart1Application [Spring Boot App] C:\Users\chris\.p2\pool\plugins\org.eclipse.justj.openjdk.hotspot.jre.full.win32.x86_64_17.0.1.v20211116-1657\jre

```
2022-07-24 23:40:49.248  INFO 18108 --- [nio-9090-exec-2] o.a.c.c.C.[Tomcat].[localhost].[/]       :
2022-07-24 23:40:49.249  INFO 18108 --- [nio-9090-exec-2] o.s.web.servlet.DispatcherServlet        :
2022-07-24 23:40:49.250  INFO 18108 --- [nio-9090-exec-2] o.s.web.servlet.DispatcherServlet        :
Input validation test 2
```

- Here the @Min(3) annotation was applied, but the @Validated annotation was removed from the Class level.
- The “StudentId” value of 2, passed through the validation and the println() was executed, even though the @Min(3) was configured.

Custom Validation – No Annotation Constraints

- Even though the @ annotation validation constraints are “easy” to use, the application can be using custom validation to validate the incoming data before it touches any business logic.

- **Proof of concept**:
    1. The StudentValidations.java Class was created under the “com.example.demo.validators” package. This Class uses regex to ensure that a String parameter only contains words, digits, or spaces and the length is less than 20 characters. It will return the Boolean value of “true” if the String meets these constraints.
    2. The getOneStudentByName() method in the StudentRestController.java Class uses the custom validation Class/method to validate the data.
    3. If the incoming data does not meet these constraints, then the application will reject the request and will not process the data.

```
31      * returned.
32      */
33•     public boolean isValid(String value) {
34
35          Pattern pattern = Pattern.compile("[\\w\\d\\s]+");
36
37          Matcher matcher = pattern.matcher(value);
38
39          if (matcher.matches() && value.length() < 20) {
40
41              return true;
42
43          } else {
44
45              return false;
46
47          } // end if/else
48
```

- Custom Validator that will return True only if the String value contains only letters, digits, spaces and is less than 20 characters.

```
169•    @GetMapping("/getAllStudentsByName/{StudentName}")
170     public ResponseEntity<String> getOneStudentByName(@PathVariable("StudentName") String name)
171
172         /*
173          * Custom Validation Example:
174          *
175          * Here instead of using annotation validation constraints (@Pattern, @Min, etc.),
176          * we can use a custom validation method.
177          *
178          * If the path variable input does not meet the validation checks set in the isValid() m
179          * the application will return an error message and the business logic will not be proce
180          */
181
182         boolean valid = studValid.isValid(name);
183
184         System.out.println("Boolean value is " + valid);
185
186
187         try {
188
189             if (valid) {
190
191                 // Test used to validate this is not processed when the "name" does not meet val
192                 System.out.println("Input validation test " + name);
193
194                 Student student = repo.findByName(name);
```

- The “name” parameter, which is tied to the “StudentName” path variable, will be validated using the isValid() method in the StudentValidations.java Class.

- When injecting the value “Test123” the Boolean value was equal to true and the 2nd println() statement was executed.

```
181
182         boolean valid = studValid.isValid(name);
183
184         System.out.println("Boolean value is " + valid);
185
186
187         try {
188
189             if (valid) {
190
191                 // Test used to validate this is not processed when the "name" does not meet validations
192                 System.out.println("Input validation test " + name);
193
194                 Student student = repo.findByName(name);
```

Problems Javadoc Declaration Console

```
AppSecSpringSecurityPart1 - AppSecSpringSecurityPart1Application [Spring Boot App] C:\Users\chris\.p2\pool\plugins\org.eclipse.justj.openjdk.hotspot.jre.full.win32.x86_64_17.0.1.v20211116-1657\jre\bin\javaw.exe
Boolean value is true
Input validation test Test123
```

```
181
182         boolean valid = studValid.isValid(name);
183
184         System.out.println("Boolean value is " + valid);
185
186
187         try {
188
189             if (valid) {
190
191                 // Test used to validate this is not processed when the "name" does not meet validations
192                 System.out.println("Input validation test " + name);
193
194                 Student student = repo.findByName(name);
```

Problems Javadoc Declaration Console

AppSecSpringSecurityPart1 - AppSecSpringSecurityPart1Application [Spring Boot App] C:\Users\chris\.p2\pool\plugins\org.eclipse.justj.openjdk.hotspot.jre.full.win32.x86_64_17.0.1.v20211116-1657\jre\bin\javaw.exe

```
Boolean value is false
```

- When injecting the value “Test123<><>” the Boolean value was equal to false and the 2nd println() statement was not executed.
- Only valid data is allowed to be processed by the application.

GET http://localhost:9090/studentApi/getAllStudentsByName/Test123<><>

Params Authorization Headers (8) Body Pre-request Script Tests Settings

none form-data x-www-form-urlencoded raw binary GraphQL

This request does not have a body

Body Cookies (1) Headers (10) Test Results Status: 400 Bad Request

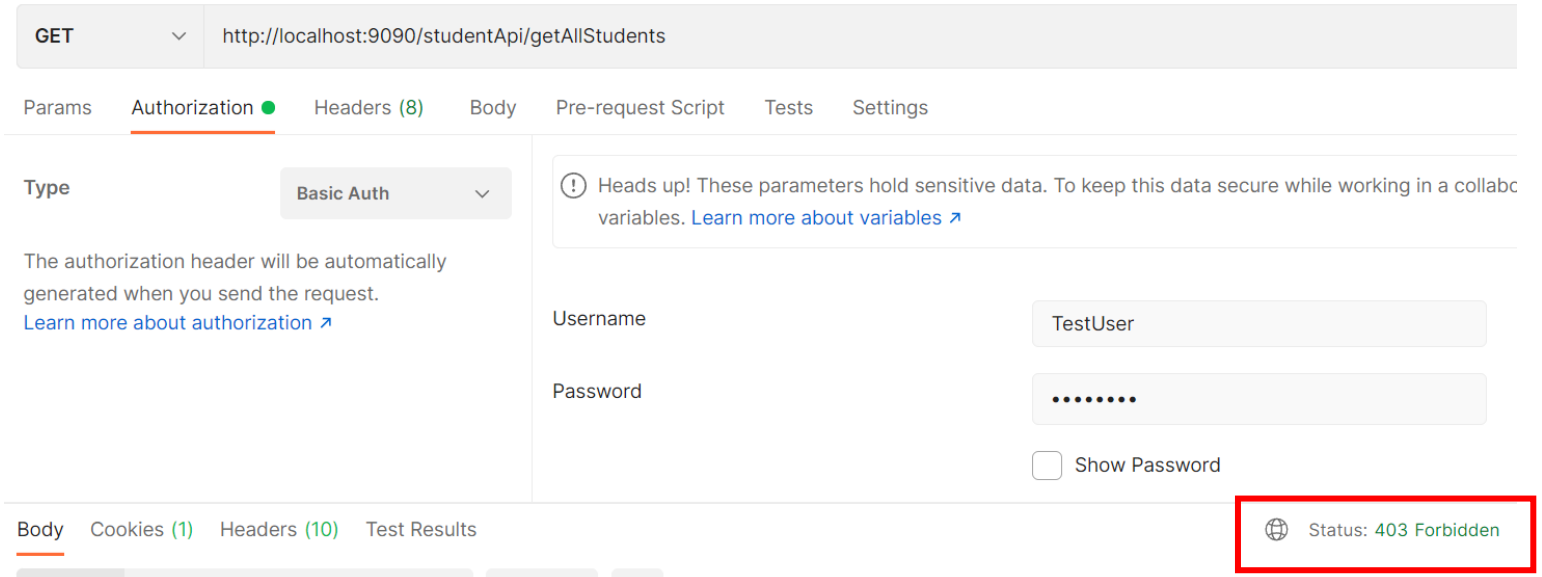
Pretty Raw Preview Visualize Text

```
1   Student name did not pass input validation checks.
```

Access Controls

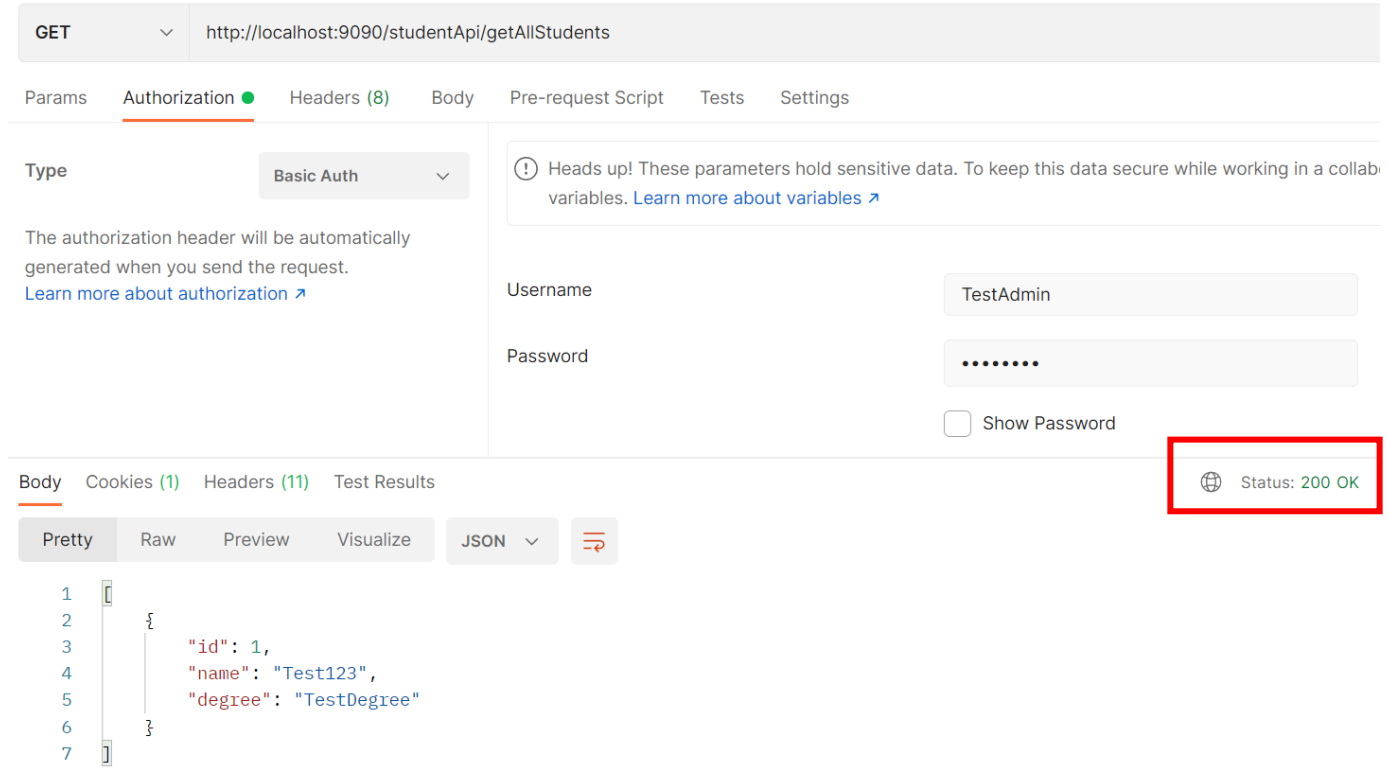
Access Controls Configuration

- The SecurityConfigurer.java Class will hold the Authentication and URL level Authorization controls.
- 2 in memory users were created for testing purposes:
  - User Creds = TestUser:TestPass ; Role = USER
  - User Creds = TestAdmin:TestPass ; Role = ADMIN
- The user with the role “USER” will only be allowed to create a Student.
- The user with the role “ADMIN” will be allowed to access all endpoints.
- HTTP basic authentication is used here.
- The Class has more details and comments.

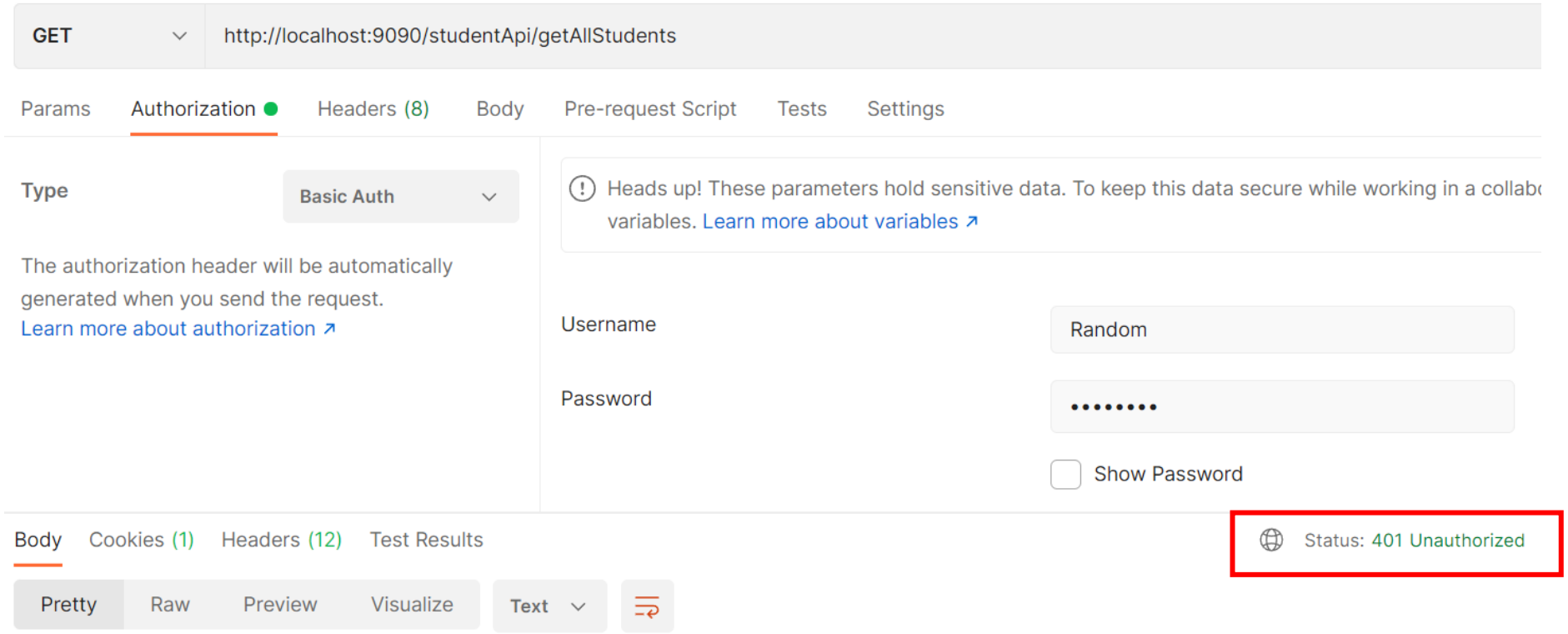


- The user “TestUser” can only create a Student, they are not authorized to access the /getAllStudents endpoint.
- The application responds with a 403 Forbidden since the user is not allowed to access the resource.

- The user “TestAdmin” can access all the endpoints, so the application responds with a 200 OK.



- When the user does not exist, the application returns a 401 Unauthorized.



Resources

- https://www.youtube.com/watch?v=UfOxcrxhC0s
- https://www.udemy.com/course/spring-security-fundamentals/
- https://reflectoring.io/bean-validation-with-spring-boot/
- https://spring.io/guides
- https://www.codejava.net/frameworks/spring-boot/spring-boot-form-validation-tutorial
- https://www.baeldung.com/spring-boot-bean-validation
- CSRF for REST APIs Spring - https://stackoverflow.com/questions/51026694/spring-security-blocks-post-requests-despite-securityconfig