

**Computer Science Department**

**CCP6427: Cloud Engineering**

**Spring 2022**

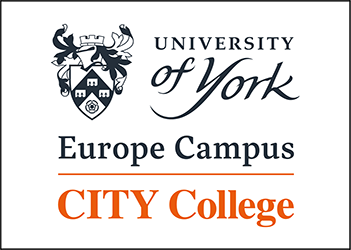
**Design and development of a microservices-based application**

**Submission Deadline: 22/5/2022**

**Actual Submission Date: 22/5/2022**

|  |
| --- |
| **Student Registration Number/s** |
| WMY21040 |

**Declaration**: *I have completed and submitted this work by myself without assistance from or communication with another person either external or fellow student. I understand that not working on my own will be considered grounds for unfair means and will result in fail mark for this work and might invoke disciplinary actions. It is at the instructor’s discretion to conduct an oral examination, if necessary, which will result in the award of the final grade for that particular piece of work.*



Report

**File Conversion Microservices with Eureka**

**Abstract (250 – 300 words)**

My abstract here.

**Keywords:** Microservices, Springboot, Eureka Netflix, Model View Controller (MVC), Representational State Transfer (REST), Service Discovery Protocol (SDP).

Table of Contents

[1. Introduction 1](#_Toc104150683)

[2. System Specification 1](#_Toc104150684)

[3. Design 1](#_Toc104150685)

[4. Implementation 3](#_Toc104150686)

[4.1 Sequential Analysis of Developed Classes 3](#_Toc104150687)

[4.2 Technology Specific Analysis 7](#_Toc104150688)

[4.3 Commitment to Richardson’s Maturity Model 7](#_Toc104150689)

[5. Testing 8](#_Toc104150690)

[6. Evaluation 8](#_Toc104150691)

[7. Conclusion 8](#_Toc104150692)

# 

**Table of Figures**

[Figure 3.1 Architectural Design Diagram 2](file:////Users/scientist/Documents/GitHub/cloud_cw_nikos_microservices/CC%20CW%20Report.docx#_Toc104150693)

[Figure 3.2 High-Level Design Diagram 3](file:////Users/scientist/Documents/GitHub/cloud_cw_nikos_microservices/CC%20CW%20Report.docx#_Toc104150694)

[Figure 4.1 Home Controller Code 3](file:////Users/scientist/Documents/GitHub/cloud_cw_nikos_microservices/CC%20CW%20Report.docx#_Toc104150695)

[Figure 4.2 User Controller Code 4](file:////Users/scientist/Documents/GitHub/cloud_cw_nikos_microservices/CC%20CW%20Report.docx#_Toc104150696)

[Figure 4.3 Exception Response Builder Code 4](file:////Users/scientist/Documents/GitHub/cloud_cw_nikos_microservices/CC%20CW%20Report.docx#_Toc104150697)

[Figure 4.4 User Responder Service Code 5](file:////Users/scientist/Documents/GitHub/cloud_cw_nikos_microservices/CC%20CW%20Report.docx#_Toc104150698)

[Figure 4.5 Text to PDF Controller Code 6](file:////Users/scientist/Documents/GitHub/cloud_cw_nikos_microservices/CC%20CW%20Report.docx#_Toc104150699)

[Figure 4.6 Text to PDF Conversion Code 6](file:////Users/scientist/Documents/GitHub/cloud_cw_nikos_microservices/CC%20CW%20Report.docx#_Toc104150700)

[Figure 4.7 PDF Formatting Code 7](file:////Users/scientist/Documents/GitHub/cloud_cw_nikos_microservices/CC%20CW%20Report.docx#_Toc104150701)

[Figure 0.1 Excerpted from Martin Fowler's Article on Richardson's Maturity Levels 7](#_Toc104150702)

# Introduction

In this project the microservices software engineering architecture will be put in practice through a minimal system. The system in question will be responsible for receiving plain text data, converting it to a pdf file and sending it back to the client application. The goal is to create a realistic and flexible microservice application which addresses traditional issues from service-oriented design.

# System Specification

The purpose of the developed system can be further elaborated upon by acknowledging its requirements in detail. The system has minimal business logic, but several architectural constraints require consideration. System requirements will be presented organised by type below.

**Architectural system requirements:**

* Microservices are designed independently of each other’s implementation, communicating strictly through HTTP requests.
* The use of Model View Controller (MVC) software design pattern.
* Registration of microservices to Netflix Eureka service registry.

**Functional system requirements:**

* Users expect a pdf file to be generated containing user provided plain text.

# Design

Harbouring understanding of the system specification allows for the creation of the software design. An appropriate representation to demonstrate the system’s design is through the use of a software architecture diagram with service-to-service communication.

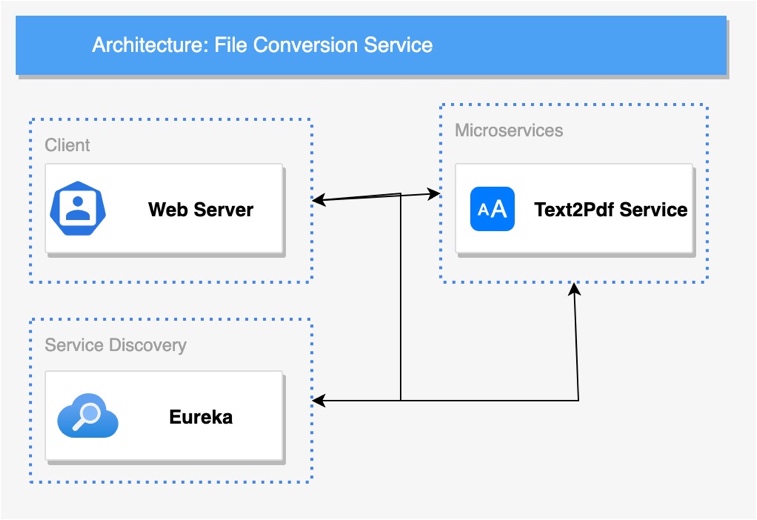
Elaborating on the figure, clients and microservices have to register to the Eureka service discovery. Eureka will map services using the service discovery protocol to allow for communication between registered systems through the lookup operation. Lookup only requires the provision of the service name from the requestor. Eureka also monitors the state of each registered service in order to react to system failures during runtime.

Figure 3.1 Architectural Design Diagram

The web server is responsible for providing to users a UI through which they can initiate the conversion process. The user sends a request containing the text that needs to be transformed and the relevant microservice (as routed by Eureka) is expected to serve the request. The text to pdf conversion service then receives a request for a conversion and performs the operation. Independent of a successful or erroneous outcome, a request is sent back to the web server with the file or exceptions that led to operation failure. Lastly, the web server reacts to the microservice response by serving the file or informing the user of errors that may have occurred.

By addressing the communication between the services and their purpose, an effort will be placed in explaining each service’s design individually. This will be performed with a diagram depicting each service’s package. Eureka will be excluded from the process due to its minimal setup and package footprint.

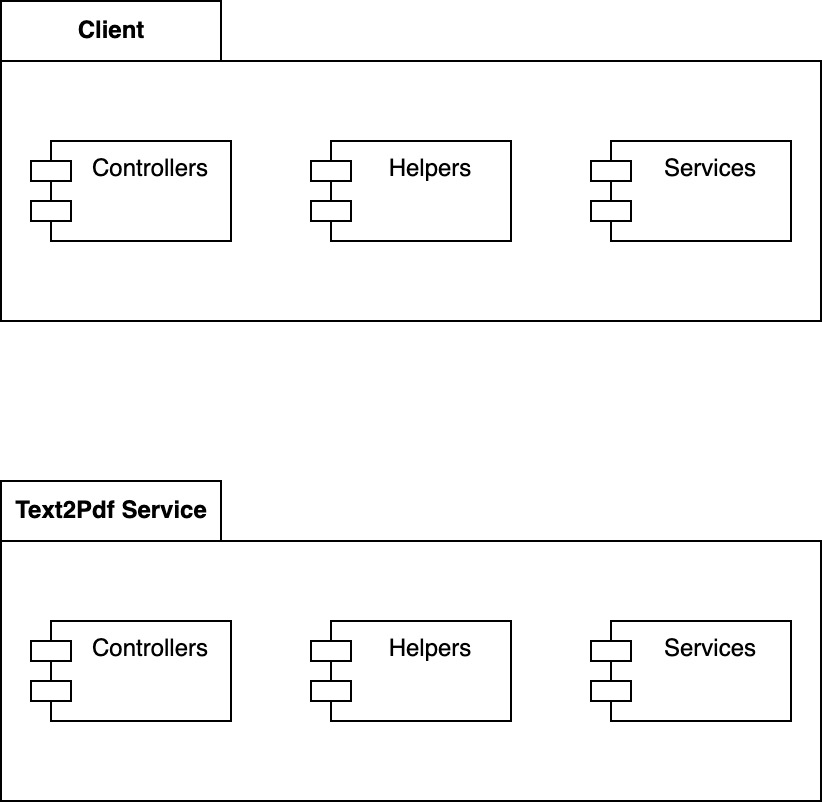
There are three common elements amongst the client and the service. Controllers in both applications will be used to provide external access to the service using REST API’s. These will contain minimal logic and delegate tasks retrieved from requests to other components. Services which will contain business logic are responsible of resolving requests. A notable exception is the client’s services which are only responsible with contacting other microservices to solve requests. In this case, the text to pdf microservice receives a request at its controller from the client. The text to pdf application will then ask its services to handle the requested operation with business logic and return a response. The response is then received by the client’s service which forwards it to the requestor through the controller. The text to pdf microservice may be accessed individually with requests, mainly intended for developers. In the case of an error, helpers will construct an exception stack trace that tracks exceptions across applications.

Figure 3.2 High-Level Design Diagram

# Implementation

The design of the software is further analysed at a lower level during implementation.

## Sequential Analysis of Developed Classes

Following the sequence of events that lead to the generation of a pdf that is downloaded by a user, the process and relevant classes will be analysed in this section.

Controllers are present in both applications and the home controller is the same in both solutions. It leads to the index page of each application.

@Controller  
public class HomeController {  
 // Provide access to the index page.  
 @RequestMapping("/")  
 public String home() {  
 return "index"**;** }  
}

Figure 4.1 Home Controller Code

In the client application, the user controller is responsible for receiving file conversion requests using the HTTP GET method. The appropriate service then handles this request. It also builds an appropriate error response if exceptions occur by issuing an operation to the exception response builder.

// Endpoints  
// Expect a GET request.  
@RequestMapping(method = RequestMethod.*GET***,** value = "/txt2pdf")  
public ResponseEntity<?> requestTextToPdf(String input) {  
 // Send request through the service to the text to pdf microservice.  
 try {  
 return userResponderService.sendTextToPDFRequest(input)**;** }  
 // Catch exceptions from the user responder service and send a context-full response.  
 catch (Exception exception) {  
 Map<String**,** String> body = ExceptionResponseBuilder.*buildExceptionResponse*(exception.toString()**,** UserController.class.getSimpleName())**;** return ResponseEntity.*internalServerError*().contentType(MediaType.*APPLICATION\_JSON*).body(body)**;** }  
}

Figure 4.2 User Controller Code

The exception response builder which is available in both applications shares the same logic. As a parameter any existing stack trace and the class name of the caller class are added to the response. A timestamp is also generated for this stack trace. Then as a whole a JSON structure is returned back to the controller in order to be sent as a response to the user.

public class ExceptionResponseBuilder {  
 // Creates an exception response by assembling the body.  
 public static Map<String**,** String> buildExceptionResponse(String content**,** String creator){  
 // Create JSON structure and add provided data  
 Map<String**,** String> body = new HashMap<>()**;** body.put("message"**,** content)**;** body.put("failed\_at"**,** creator)**;** // Add Timestamp  
 SimpleDateFormat dateFormat = new SimpleDateFormat("HH:mm:ss dd-MM-yyyy")**;** String timeStamp = dateFormat.format(new Date())**;** body.put("timestamp"**,** timeStamp)**;** return body**;** }  
}

Figure 4.3 Exception Response Builder Code

When the user responder service receives a conversion request, it will prepare a request to be sent to the text to pdf microservice. A specified response type is added as a header which is then assembled in the request using rest template. The exchange method is used to explicitly select the HTTP method to be used. If the request fails to be sent, an exception is forwarded to the user controller.

// Sends a request for a txt2pdf conversion to an external microservice with the user input.  
public ResponseEntity<?> sendTextToPDFRequest(String input) {  
 try {  
 // Setup Headers & URL  
 String url = serviceUrl + "/txt2pdf?input={input}"**;** HttpHeaders headers = new HttpHeaders()**;** // Accept PDF media types responses  
 headers.setAccept(Collections.*singletonList*(MediaType.*APPLICATION\_PDF*))**;** HttpEntity<String> entity = new HttpEntity<>("body"**,** headers)**;** // Send a GET request to the text to pdf microservice's controller with the user input.  
 // Expect a byte[] format response.  
 return restTemplate.exchange(url**,** HttpMethod.*GET***,** entity**,** byte[].class**,** input)**;** } catch (Exception e) {  
 throw new RuntimeException("Failed to send request to microservice, cause: " + e)**;** }  
}

Figure 4.4 User Responder Service Code

On the text to pdf microservice once a request is received, it will instruct the text to pdf service to initiate the conversion. If the conversion succeeds, the controller will prepare the response headers and the pdf file name. Otherwise, the operation stops and an exception stack trace is attached to a new HTTP response (generated using exception response builder).

@RequestMapping(method = RequestMethod.*GET***,** value = "/txt2pdf")  
public ResponseEntity<?> serveTextToPdf(String input) {  
 try {  
 // Create pdf name based on the date and time (flexible for frequent users, no duplicate names)  
 DateFormat dateFormatter = new SimpleDateFormat("dd-MM-yyyy:hh:mm:ss")**;** String currentDateTime = dateFormatter.format(new Date())**;** // Prepare Headers to open PDF on the client and allow downloading.  
 String headerKey = "Content-Disposition"**;** String headerValue = "inline; filename=pdf\_" + currentDateTime + ".pdf"**;** // Get converted result  
 byte[] res**;** res = this.textToPDFService.convertTextToPdf(input)**;** // Send a successful response  
 return ResponseEntity.*ok*()  
 .header(headerKey**,** headerValue)  
 .contentType(MediaType.*APPLICATION\_PDF*)  
 .body(res)**;** }  
 // Catch exceptions from the text to pdf service and send a context-full response.  
 catch (Exception exception) {  
 Map<String**,** String> body = ExceptionResponseBuilder.*buildExceptionResponse*(exception.toString()**,** TextToPdfController.class.getSimpleName())**;** return ResponseEntity.*internalServerError*().contentType(MediaType.*APPLICATION\_JSON*).body(body)**;** }  
}

Figure 4.5 Text to PDF Controller Code

The text to pdf service receives the request for conversion with the user input and performs the conversion. A byte array output stream and a document are required. The document comprises of a pdf writer and pdf document. In the case input was not provided in the request, sample text is used. A minimal format method is responsible for providing the necessary structure to the document. In case an exception occurs during the creation or writing of the pdf file, the operation stops and an exception is sent to the controller.

public byte[] convertTextToPdf(String input) {  
 try {  
 // Create Pdf  
 final ByteArrayOutputStream output = new ByteArrayOutputStream()**;** var writer = new PdfWriter(output)**;** var pdf = new PdfDocument(writer)**;** var document = new Document(pdf)**;** // Provide some text if the input was null  
 if (input == null)  
 input = "Seems like no input was provided, here is some text for you!"**;** // Format  
 format(document**,** input)**;** // Finalize Operation  
 document.close()**;** return output.toByteArray()**;** } catch (Exception e) {  
 throw new RuntimeException("Exception thrown when converting text to pdf at: " + e)**;** }  
}

// All pdf formatting should occur within. The document is preserved without a return.

Figure 4.6 Text to PDF Conversion Code

The pdf format method at the moment only creates a paragraph using the user input, but the de-coupling allows for easier addition of new formatting requirements in the future.

private void format(Document document**,** String input) {  
 document.add(new Paragraph(input))**;**}

Figure 4.7 PDF Formatting Code

## Technology Specific Analysis

An analysis of technologically bound elements of the system will be performed aiming to demonstrate a deeper understanding of the software.

The

## Commitment to Richardson’s Maturity Model

An evaluation of the REST API endpoints and HTTP requests will highlight the level of commitment to Richardson’s maturity model.



Figure 4.8 Excerpted from Martin Fowler's Article on Richardson's Maturity Levels [1]

In order to assure Level 0 is reached, we need to use HTTP methods for external communication. This is reached in the software as proven in the previous section. Level 1 requires the use of resources. In this system since there is no persistence endpoints do not utilise the REST resource system. In Level 2 it is required that HTTP verbs are used appropriately following their semantic definition. In this project the GET request is used explicitly since all operations are reproducible (idempotence) and do not alter resources. Level 3 requires HATEOAS Hypermedia implementation not present in this version of the system.

# Testing

The software was tested to ensure that during development errors introduced can be more easily identified. Also, testing enabled the discovery of unknown bugs during development. Overall, 77% of all line of codes and 90% of methods were tested although the type of testing possessed varied aims.

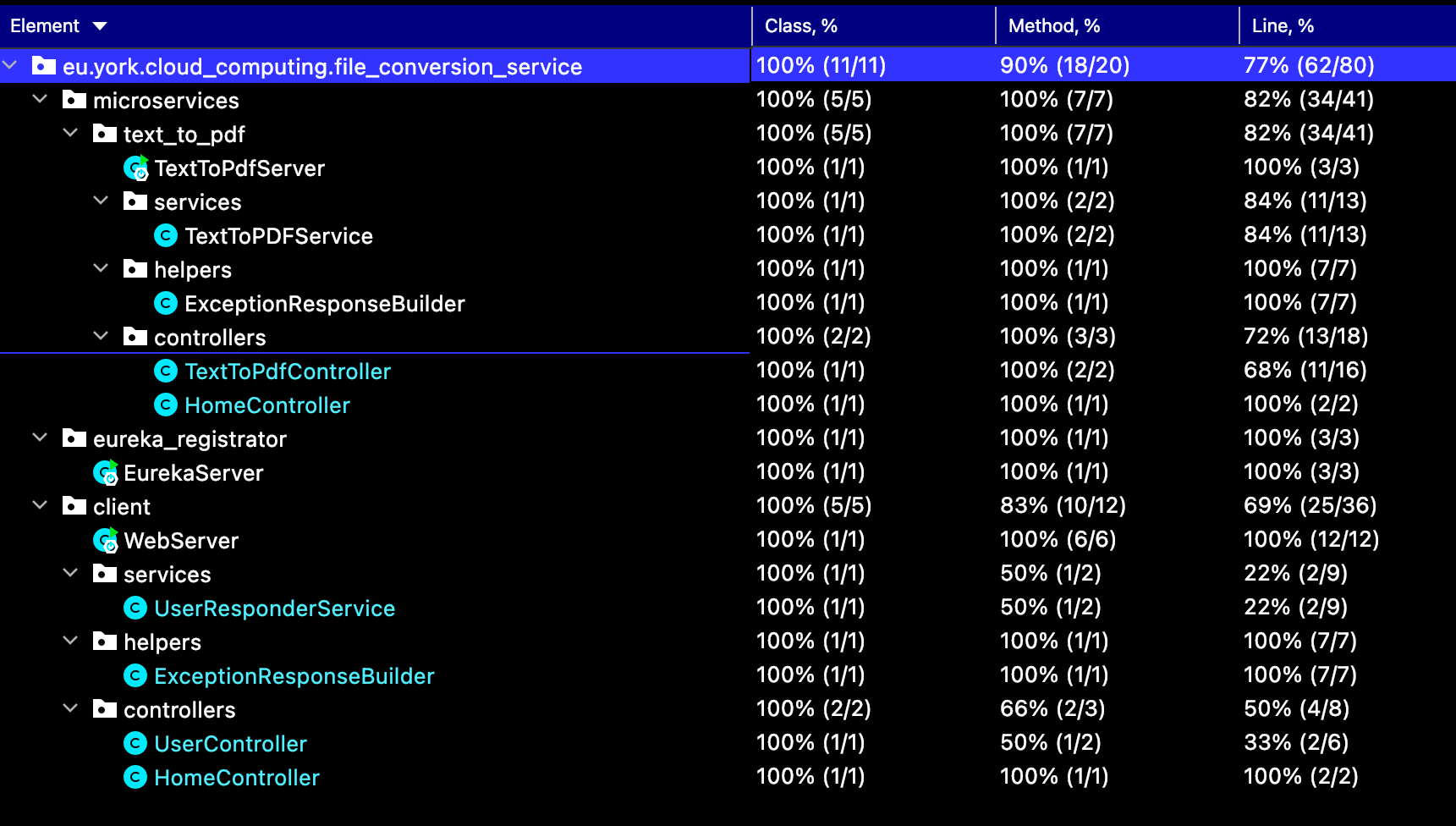


Figure 5.1 Testing Coverage

Both home controllers had identical tests. A unit test ensured that during and after creation the controller never had a null state. An integration test which run a mocked HTTP request to make sure the system would redirect to the appropriate page and return the expected HTTP code.

@SpringBootTest(classes = HomeController.class)  
@AutoConfigureMockMvc  
class HomeControllerTest {  
 @Autowired  
 MockMvc mockMvc**;** HomeController homeController = new HomeController()**;** // Unit Test  
 @Test  
 void controllerNotNull() {  
 *assertNotNull*(homeController)**;** }  
  
 // Integration Test  
 @Test  
 void controllerRedirectSuccess() throws Exception {  
 mockMvc.perform(*get*("/").content("index")).andExpect(*status*().isOk()).andExpect(*forwardedUrl*("index"))**;** }  
}

Figure 5.2 Home Controller Test

The exception response builder constituted of a single test that checked if the return was not null and contained the input values in the JSON response.

class ExceptionResponseBuilderTest {  
  
 @Test  
 void buildExceptionResponse() {  
 Map<String**,** String> res = ExceptionResponseBuilder.*buildExceptionResponse*("test"**,** "someone")**;** String expectedInString = "someone"**;** String expectedInString2 = "test"**;** // Make sure it's not null  
 *assertNotNull*(res)**;** // Check if result contains the given parameters  
 *assertTrue*(res.toString().contains(expectedInString) && res.toString().contains(expectedInString2))**;** }  
}

Figure 5.3 Exception Response Builder Test

The Eureka server had an integration test which asserted that the main method should run successfully.

class EurekaServerTest {  
  
 // Integration Test  
 @Test  
 void eurekaServerLoadsMainMethod() {  
 EurekaServer.*main*(new String[] {})**;** }  
}

Figure 5.4 Eureka Server Test

The user controller was tested for integration with the rest of the system just to ensure that it never reaches a null state.

@SpringBootTest(classes = {UserController.class**,** UserResponderService.class})  
class UserControllerTest {  
 @MockBean  
 UserResponderService userResponderService**;** UserController userController = new UserController(userResponderService)**;** // Integration Test  
 @Test  
 void controllerNotNull() {  
 *assertNotNull*(userController)**;** }  
}

Figure 5.5 User Controller Test

The web server was tested to make sure its main method runs successfully in integration tests and the application context never reached a null state during that process.

class WebServerTest {  
 // Integration Tests  
 @Test  
 void webServerLoadsMainMethodContextNotNull() {  
 WebServer.*main*(new String[] {})**;** // Context not null  
 *assertNotNull*(WebServer.*applicationContext*)**;** }  
}

Figure 5.6 Web Server Test

The text to pdf controller was tested for status code, data type of return and other header data conformance.

@SpringBootTest(classes = {TextToPdfController.class**,** TextToPDFService.class})  
class TextToPdfControllerTest {  
 TextToPDFService textToPDFService = new TextToPDFService()**;** TextToPdfController textToPdfController = new TextToPdfController(textToPDFService)**;** @Test  
 void serveTextToPdfReturnsPDFStatusSuccess() throws Exception {  
 ResponseEntity<?> res = textToPdfController.serveTextToPdf("hello")**;** // Appropriate status code received  
 *assertEquals*(res.getStatusCode()**,** HttpStatus.*OK*)**;** // Appropriate content disposition received  
 String expectedContentDis = ContentDisposition.*builder*("inline").build().toString()**;** String resContentDis = res.getHeaders().getContentDisposition().toString()**;** *assertTrue*(resContentDis.contains(expectedContentDis))**;** // Appropriate media type  
 String expectedMediaType = MediaType.*APPLICATION\_PDF*.toString()**;** String resMediaType = res.getHeaders().getContentType().toString()**;** *assertEquals*(resMediaType**,** expectedMediaType)**;** }  
  
 @Test  
 void serveTextToPdfGivenNullReturnsNonNullStatusSuccess() throws Exception {  
 ResponseEntity<?> res = textToPdfController.serveTextToPdf(null)**;** // Not Null  
 *assertNotNull*(res)**;** // Appropriate status code received  
 *assertEquals*(res.getStatusCode()**,** HttpStatus.*OK*)**;** }  
}

Figure 5.7 Text to PDF Controller Test

Lastly, the text to pdf server was tested for integration with the rest of the system for a successful main method operation.

class TextToPdfServerTest {  
  
 // Integration Tests  
 @Test  
 void textToPdfServerLoads() {  
 TextToPdfServer.*main*(new String[] {})**;** }  
}

Figure 5.8 Text to PDF Server Test

Overall, due to time constraints the testing effort was minimized to improve other elements of the software, but still applied to a level which can help current and future development efforts.

# Evaluation

In any project evaluation from the developer is a critical component which can improve their skill, the continuation of the project and reveal further insight into development.

There were some improvements to be made that were not actually integrated to the final product:

* A more sophisticated system of exception handling which would read on the client side if an error has occurred by expecting an exception builder response JSON. This would allow then the redirection of the user to a page where a user-friendly error message would be shown instead.
* Traditionally a Continuous Integration (CI) solution would be used to automate test runs, but due to the small size of the project this was omitted. Usually, the tool the developer is familiar with for such use cases is GitHub Actions.

# Conclusion

**References**

[1] F. Martin, “Richardson Maturity Model,” *steps toward the glory of REST*, 2010. [Online]. Available: https://www.martinfowler.com/articles/richardsonMaturityModel.html. [Accessed: 22-May-2022].