

QA Academy Project 2 - Music Book John Fisher

Project Presentation and Overview



Project Spec

- My main focus upon receiving the Project Spec was to read and fully understand it, with emphasis on the Scope, Constraints, and MVP.

This project was then planned around the noted constraints, using:

- Jira - Scrum/Kanban Board
- mySQL - Database
- Java - Using Eclipse IDE to build back-end
- HTML, CSS, Javascript - Using VSCode to build Front-End
- MockMVC - to test back-end
- Git/GitHub - Version Control



Project Planning Phase

Once the requirements, constraints, and MVP was ascertained, I began planning my project using a Jira Scrum Board.

- 5 Epics were created, to break each part of the project into manageable pieces
- Then, a number of of User Stories were created for each epic to further break down the tasks into the needs of Users and Developers
- Child issues for these User Stories were then created to specify what work had to be done to complete them in more detail.

Once this one done, the first sprint could be started (documentation, setting up Version Control, etc.) There were 5 Sprints in total, all of which were completed by their deadlines.




 Status category ▾

Epic	FEB	MAR
<ul style="list-style-type: none"> JMP-1 Create Documentation for the Pr... JMP-2 Create a Java Back-end for the pr... JMP-3 Create tests for back-end of Appl... JMP-4 Create a Front-end using HTML ... JMP-5 Use a Version Control System to ... 		
+ Create Epic		

JMP Sprint 5



 Epic 

TO DO

IN PROGRESS 2 ISSUES

As a developer, I would like to create a UML, so that I can better understand the functionality of the application.

CREATE DOCUMENTATION FOR TH..

 JMP-14

As a User, I would like to have a README.md file, so that I can understand how to use the app and how it has been made.

CREATE DOCUMENTATION FOR TH...

 JMP-15

Note the still In Progress Documentation User Stories, which revolve around making this presentation, the README.md, and the UML.

The below image displays a completed Epic and User Stories.

The screenshot displays a Jira board with a Kanban workflow. The board is organized into columns representing time periods: FEB, MAR, APR, and MAY. An epic titled 'Create a Java Back-end for the project' is expanded, revealing a list of tasks. The task 'JMP-27: As a User, I would like the app to perform CRUD functionality...' is currently in progress, positioned in the MAR column. Other tasks include 'JMP-28: As a Developer, I would like the app to have a user interface...', 'JMP-29: As a Developer, I would like the app to have a database...', 'JMP-30: As a Developer, I would like the app to have a login system...', 'JMP-31: Create relevant Domain Objects...', and 'JMP-32: Create methods to perform CRUD operations...'. The right sidebar provides a detailed view of the selected task 'JMP-27', showing its description, a 'Done' button, and a progress bar indicating 100% completion.



Back-End Development

Once the planning phase and initial documentation was complete, I could begin developing the back-end of my app.

A total of 5 classes/interfaces were created: Music(domain), MusicRepo(repo), MusicService and ServiceInterface(service), and MusicController(web).

A Database is also required for the app to function and to persist data. Below is an image of the Music.java file, setting the fields and columns for the app:

```
@Entity
public class Music {

    @Id //This is the Primary Key
    @GeneratedValue(strategy = GenerationType.IDENTITY) //This AutoIncrements
    private Integer id;

    @Column(nullable = false) //Marks column as NOT NULL
    private String song;

    @Column(nullable = false)
    private String album;

    @Column(nullable = false)
    private String artist;

    @Column(nullable = false)
    private Integer releaseDate;

    @Column(nullable = false)
    private String artistCountry;

    //Constructors
    public Music() {
        super();
    }

    public Music(Integer id, String song, String album, String artist, Integer releaseDate, String artistCountry) {
        super();
        this.id = id;
        this.song = song;
        this.album = album;
        this.artist = artist;
        this.releaseDate = releaseDate;
        this.artistCountry = artistCountry;
    }
}
```

Back-End Development

The below images show the ServiceInterface.java file to allow for basic CRUD functionality. This is expanded on with the MusicRepo.java, which opens up the app to add more functionality in the future.

```
@Service
public class MusicService implements ServiceInterface<Music> {

    private MusicRepo MRepo;

    @Autowired
    public MusicService(MusicRepo MRepo) {
        this.MRepo=MRepo;
    }

    //Creates new entry
    public Music create(Music m) {
        Music created = this.MRepo.save(m);
        return created;
    }

    //Reads all entries
    public List<Music> getAll(){
        return this.MRepo.findAll();
    }

    //Reads entry with specified ID
    public Music getIndividual(Integer id) {
        Optional<Music> found = this.MRepo.findById(id);
        return found.get();
    }

    //Updates existing entry
    public Music update(Integer id, Music newMusic) {
        Music existing = this.MRepo.findById(id).get();
        existing.setSong(newMusic.getSong());
        existing.setAlbum(newMusic.getAlbum());
        existing.setArtist(newMusic.getArtist());
        existing.setReleaseDate(newMusic.getReleaseDate());
        existing.setArtistCountry(newMusic.getArtistCountry());
        Music updated = this.MRepo.save(existing);
        return updated;
    }
}
```

```
package com.qa.project2.service;

import java.util.List;

public interface ServiceInterface<T> {

    T create(T t);

    List<T> getAll();

    T getIndividual(Integer id);

    T update(Integer id, T t);

    void delete(Integer id);

}
```

Back-End Development

The MusicService implements the ServiceInterface and the MusicRepo to perform the required functions. Then, the MusicController.java uses the Service class to create and tell the APIs what to do with with each URL. Note the '@**Mapping'.

```
@CrossOrigin
@RestController
public class MusicController {

    private MusicService service;

    @Autowired //Tells spring to fetch the MusicService
    public MusicController(MusicService service) {
        this.service = service;
    }

    @PostMapping("/create")
    public ResponseEntity<Music> createMusic(@RequestBody Music m) {
        Music created = this.service.create(m);
        ResponseEntity<Music> response = new ResponseEntity<Music>(created, HttpStatus.CREATED);
        return response;
    }

    @GetMapping("/getAll")
    public ResponseEntity<List<Music>> getAllMusic() {
        return ResponseEntity.ok(this.service.getAll());
    }

    @GetMapping("/get/{id}")
    public Music getMusic(@PathVariable Integer id) {
        return this.service.getIndividual(id);
    }

    @PutMapping("/update/{id}")
    public ResponseEntity<Music> updateMusic(@PathVariable Integer id, @RequestBody Music newMusic) {
        Music body = this.service.update(id, newMusic);
        ResponseEntity<Music> response = new ResponseEntity<Music>(body, HttpStatus.ACCEPTED);
        return response;
    }
}
```

```
@Service
public class MusicService implements ServiceInterface<Music> {

    private MusicRepo MRepo;

    @Autowired
    public MusicService(MusicRepo MRepo) {
        this.MRepo=MRepo;
    }

    //Creates new entry
    public Music create(Music m) {
        Music created = this.MRepo.save(m);
        return created;
    }

    //Reads all entries
    public List<Music> getAll(){
        return this.MRepo.findAll();
    }

    //Reads entry with specified ID
    public Music getIndividual(Integer id) {
        Optional<Music> found = this.MRepo.findById(id);
        return found.get();
    }

    //Updates existing entry
    public Music update(Integer id, Music newMusic) {
        Music existing = this.MRepo.findById(id).get();
        existing.setSong(newMusic.getSong());
        existing.setAlbum(newMusic.getAlbum());
        existing.setArtist(newMusic.getArtist());
        existing.setReleaseDate(newMusic.getReleaseDate());
        existing.setArtistCountry(newMusic.getArtistCountry());
        Music updated = this.MRepo.save(existing);
        return updated;
    }
}
```

Testing of Back-End

MockMVC tests were set up to test the functionality of the back-end before working on the Front-End. This was used alongside Postman, as this allowed you to perform the requests and use the APIs without an implemented Front-End. Below is an example of the MockMVC tests, showing total coverage, and an example of Postman performing the required functions.

```
@SpringBootTest(webEnvironment=WebEnvironment.RANDOM_PORT)
@AutoConfigureMockMvc //sets up MockMVC Object
@Sql(scripts= {"classpath:music-schema.sql", "classpath:music-data.sql"}, executionPhase=ExecutionPhase.BEFORE_TEST_METHOD)
@ActiveProfiles("test")
public class MusicControllerIntegrationTest {

    @Autowired //pulls MockMvc object from context
    private MockMvc mvc; //Performs request (acts as Postman)

    @Autowired
    private ObjectMapper mapper; //Java to JSON converter that Spring uses

    @Test
    void testCreate() throws Exception {
        Music testMusic = new Music(null, "Alpha Incipiens", "Zopilote Machine", "The Mountain Goats", 1994, "USA");
        String testMusicAsJSON = this.mapper.writeValueAsString(testMusic);
        RequestBuilder req = post("/create").contentType(MediaType.APPLICATION_JSON).content(testMusicAsJSON);

        Music testCreatedMusic = new Music(3,"Alpha Incipiens", "Zopilote Machine", "The Mountain Goats", 1994, "USA");
        String testCreatedMusicAsJSON = this.mapper.writeValueAsString(testCreatedMusic);
        ResultMatcher checkStatus = status().isCreated(); //201 - Created
        ResultMatcher checkBody = content().json(testCreatedMusicAsJSON); //Checks Body

        //Sends Requests and Checks
        this.mvc.perform(req).andExpect(checkStatus).andExpect(checkBody);
    }
}
```

	Coverage	Covered Instructions	Missed Instructions	Total Instructions
10 Mar 2022 11:11:52)	94.5 %	701	41	742
2AMusicProject2				

SpringDemo / http://localhost:8080/get/0

GET http://localhost:8080/get/1

Params Authorization Headers (6) Body Pre-request Script Tests

Query Params

KEY
Key

Body Cookies Headers (8) Test Results

Pretty Raw Preview Visualize JSON

```
1
2  "id": 1,
3  "song": "From Here to Utopia",
4  "album": "The Volatile Utopian Real Estate Market",
5  "artist": "Pat the Bunny",
6  "releaseDate": 2014,
7  "artistCountry": "USA"
8
```


Front-End(HTML)

Once the tests and Postman confirmed the app was functional, Front-End Development could begin! I started by structuring my webpage with HTML before anything else - I wished to utilise Buttons and Forms to display the data in a table, and to Post/Put/Delete data based on Form inputs. An example of this can be seen in the below snippet:

```
<button class="btn btn-dark" onclick="showHide(1)">Show Music By...</button>
<br />
<div id="m1" style="display: none">
  <table class="findbytable">
    <button class="btn btn-secondary" onclick="openF(9);">ID</button>
    <button class="btn btn-secondary" onclick="openF(1);">Song</button>
    <button class="btn btn-secondary" onclick="openF(2);">Album</button>
    <button class="btn btn-secondary" onclick="openF(3);">Artist</button>
    <button class="btn btn-secondary" onclick="openF(4);">
      Release Date
    </button>
    <button class="btn btn-secondary" onclick="openF(5);">
      Artist Country
    </button>
    <button class="btn btn-danger" onclick="showHide(2)">Close</button>
  </table>
</div>
<button
  class="btn btn-dark"
  action="http://localhost:8080/getAll"
  onclick="showData(1)"
>
  Show All Entries...</button>
<br />
<div id="f9" style="display: none">
  <form class="forms">
    <h2>Enter an ID number</h2>
    <label for="id">ID</label>
    <input id="id" type="text" placeholder="ID..." name="id" />

    <button type="button" class="btn btn-success" onclick="showData(2)">
      Find
    </button>
    <button type="button" class="btn btn-danger" onclick="closeF(9)">
      Close
    </button>
  </form>
</div>
```

Front-End: JavaScript(JS)

In order for these buttons and forms to function as desired, functions had to be created in JavaScript (example below). The majority of JS methods however, were to allow the forms/buttons to manipulate the data in the database - First by getting the JSON data, then by showing this on the webpage(see other example).

```
function showHide(obj) {
  for(x=1;x<=2;x++) {
    if (x == obj) {
      document.getElementById('m'+x).style.display = 'block';
    } else {
      document.getElementById('m'+x).style.display = 'none';
    }
  }
}

function openF(obj) {
  for(x=1;x<=9;x++) {
    if (x == obj) {
      document.getElementById('f'+x).style.display = 'block';
    }
  }
}

function closeF(obj) {
  for(x=1;x<=9;x++) {
    if (x == obj) {
      document.getElementById('f'+x).style.display = 'none';
    }
  }
}
```

```
let getData = async () => {
  let response = await fetch('http://localhost:8080/getAll');
  if (response.status !== 200) {
    throw new Error("Request Failed");
  }
  console.log("Request Successful");
  let jsonData = await response.json();
  console.log(jsonData);
  return jsonData;
}

let showData = async (i) => {
  let returnedData
  if (i == 1) {returnedData = await getData();}
  if (i == 2) {returnedData = await getId(); return;}
  if (i == 3) {returnedData = await getSong();}
  if (i == 4) {returnedData = await getAlbum();}
  if (i == 5) {returnedData = await getArtist();}
  if (i == 6) {returnedData = await getReleaseDate();}
  if (i == 7) {returnedData = await getArtistCountry();}
  let allTable = `
    <tr>
      <th>Id</th>
      <th>Song</th>
      <th>Album</th>
      <th>Artist</th>
      <th>Release Date</th>
      <th>Artist Country</th>
    </tr>`;
  for (let d=0;d<returnedData.length;d++) {
    allTable += `
      <tr>
        <td>${returnedData[d].id}</td>
        <td>${returnedData[d].song}</td>
        <td>${returnedData[d].album}</td>
        <td>${returnedData[d].artist}</td>
        <td>${returnedData[d].releaseDate}</td>
        <td>${returnedData[d].artistCountry}</td>
      </tr>`;
  }
  console.log(allTable);
  document.getElementById("music").innerHTML = allTable;
}
```

Front-End: CSS

Once the Front-End was functional, I was able to implement CSS and Bootstrap to beautify and improve the UI/UX. I used Bootstrap for the Buttons, and everything else was created in CSS to create a background, change fonts, etc.

```
body {
  background-image: url(music_background.jpg);
  background-attachment: fixed;
  background-size: 1920px 1080px;
  font-family: American Typewriter, serif !important;
}

.headers {
  position: relative;
  left: 100px;
  width: 600px;
  border: 2px solid gray;
  border-radius: 5px;
  background-color: darkslategray;
  color: coral;
}

input[type="text"] {
  background-color: gray;
}

::placeholder {
  color: brown;
}

.findbytable {
  margin: 0px;
}

.forms {
  position: relative;
  width: 400px;
  margin: 8px;
  border: 2px solid coral;
  border-radius: 5px;
  background-color: darkslategray;
  color: coral;
}

.maintable {
  position: absolute;
  margin-left: 30%;
  margin-bottom: 50%;
  border-radius: 5px;
  background-color: darkslategray;
  border: 2px solid darkslategray;
  color: coral;
}
```

Welcome to the QA MusicBook!

Add or view your favourite songs and artists!

Show Music By...

ID Song Album Artist Release Date Artist Country Close

Id	Song	Album	Artist	Release Date	Artist Country
2	Hell and You	Volume 1	Amigo the Devil	2018	USA

Show All Entries...

Enter a Song Name

Song Hell and You Find Close

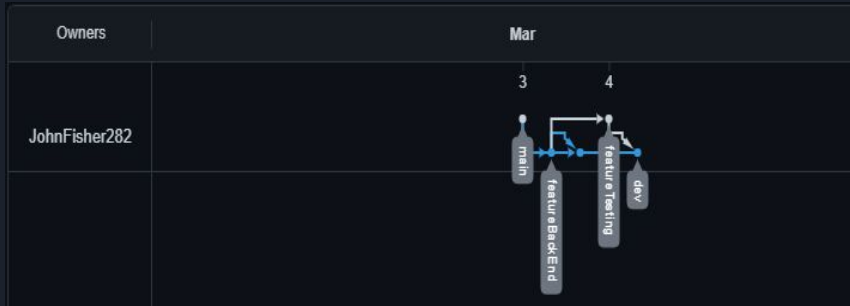
Add New Music

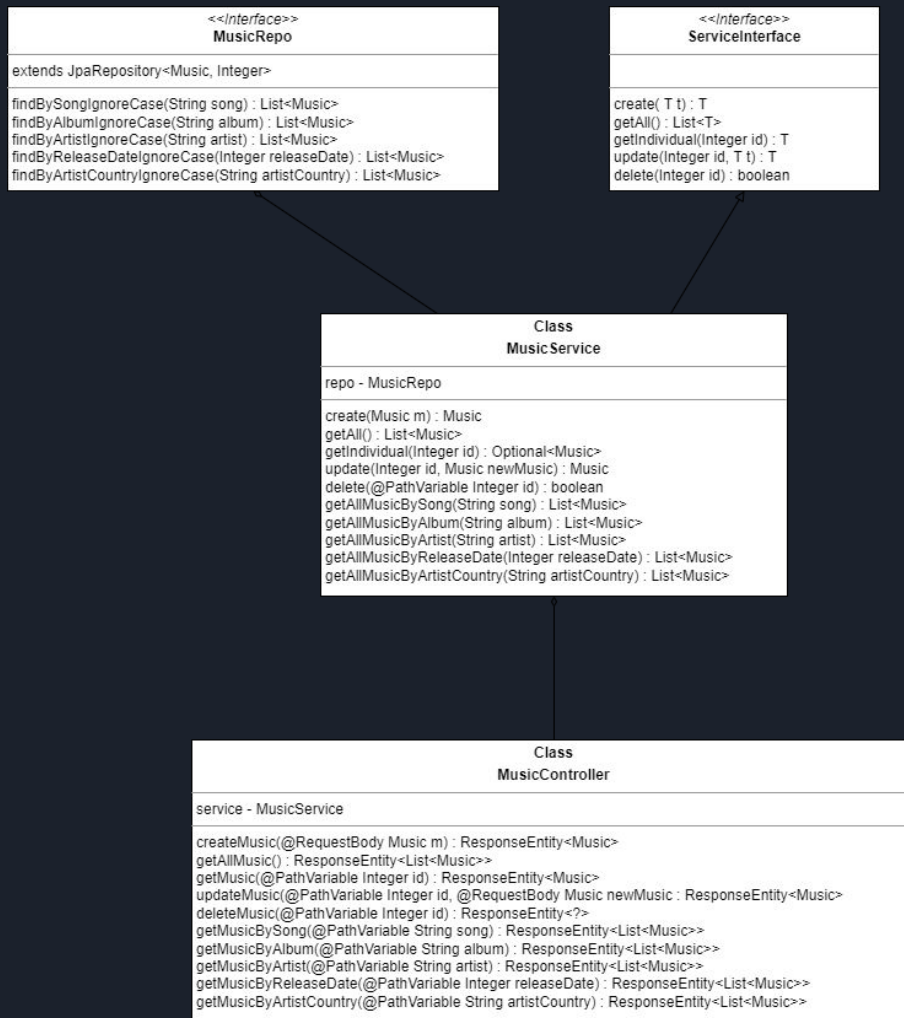
Update an Entry


Delete an Entry

Version Control & Documentation

Git was used for Version Control throughout the project. 2 Git Repositories were created - one for Front-End and one for Back-End. This allowed me to easily break the work up into portions and push it to dev as it was ready, maintaining a working version of the project at all times. Separate branches were used for Testing, creating functionality, Documentation, etc. See the below Network Graphs from each Repo (These have not yet been pushed to Main at the time of creating this presentation.)







	Risk Description	Evaluation	Likelihood (1 to 10)	Impact Level/Importance	Responsibility	Response	Control Measures
1.	IDE Crashes	Code is potentially lost	2	9	IDE / Hardware	Restore to most recent version	Frequently save work and back-up. Have up to date Git branches.
2.	Power Outage	Code potentially lost / hardware failure	1	10	Energy Provider / Act of God	Restore to most recent version / replace hardware	Frequently save and back-up, utilize Git; use surge protectors
3.	Incomplete work deployed to main branch	Non-working version of project on main branch	5	7	Developer	Revert to last working version on Git	Utilize dev branch to merge all feature branches to. Restrict access to push to main.
4.	Bugs prevent app from working	App does not work due to unresolved bugs	10	2	Developer/Testers	Test app extensively using integration tests	Consider testing when planning project; ensure app is tested before deployment
5.	App does not meet requirements	Project does not meet MVP	6	8	Developer	Prioritize basic client requirements; understand Project Spec	Fully consider project spec before planning project; prioritize essential tasks



Demo & Retrospective

Overall, I feel this project was a great improvement on my performance over my First Project as I better understood a lot of the tools/concepts used in completing it. The Back-End went much smoother than anticipated as I understood how all the classes interacted with each other.

I feel Git was used effectively, however in hindsight I would have broken up the commits into smaller commits/pushes. It is still slightly nerve-wracking when pushing to dev/main branches, as I want to be sure nothing goes wrong!

Javascript was certainly the biggest struggle during this project, and this was planned for accordingly. I had great difficulty in getting my data to show on the webpage and could definitely have benefited from studying the Fetch API further. The browser console proved invaluable in tackling this.

Formatting my webpage proved difficult towards the end, as I used a combination of Bootstrap and CSS - due to my unfamiliarity with Bootstrap, I had a hard time manipulating these elements as desired. In future, I would likely primarily use Bootstrap.

I also would have liked to carry out more extensive testing, using JUnit and Selenium - particularly Selenium for Front-End Tests. Currently, there is no testing in place for this and has been done entirely using the Console and trial-and-error.



Thanks for Listening!

Overall, I thoroughly enjoyed this project and look forward to more in the future! I particularly loved working with a Front-End, as it was extremely rewarding and satisfying to see your work materialise in a working UI!