#CS #school

Author: Oliwier Przewlocki

The first step is to set up the necessary dependencies for Maven in the `pom.xml` file

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>
<dependency>
    <groupId>io.github.wimdeblauwe</groupId>
    <artifactId>htmx-spring-boot</artifactId>
    <version>3.5.0</version>
</dependency>

<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-devtools</artifactId>
    <scope>runtime</scope>
    <optional>true</optional>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-configuration-processor</artifactId>
    <optional>true</optional>
</dependency>
<dependency>
    <groupId>org.projectlombok</groupId>
    <artifactId>lombok</artifactId>
    <optional>true</optional>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-test</artifactId>
    <scope>test</scope>
</dependency>

<dependency>
    <groupId>com.fasterxml.jackson.core</groupId>
    <artifactId>jackson-databind</artifactId>
</dependency>

<dependency>
    <groupId>com.sun.activation</groupId>
    <artifactId>javax.activation</artifactId>
    <version>1.2.0</version>
</dependency>

<dependency>
```

```
    <groupId>com.fasterxml.jackson.dataformat</groupId>
    <artifactId>jackson-dataformat-xml</artifactId>
</dependency>

<dependency>
    <groupId>org.springdoc</groupId>
    <artifactId>springdoc-openapi-starter-webmvc-ui</artifactId>
    <version>2.1.0</version>
</dependency>
```

Afterwards, the model for the data generation needs to be created to properly send them via a REST interface

```
@Getter
@Setter
@AllArgsConstructor
public class PreferredCandidate {
    private String candidateName;
    private String candidateVotes;
}
```

This class creates a PrefferedCandidate of a party. This will then be an element in a list inside of a Party object.

```
@Getter
@AllArgsConstructor
public class PartyData {
    private final String partyID;
    private final String amountVotes;

    @JacksonXmlElementWrapper(localName = "preferredCandidates")
    @JacksonXmlProperty(localName = "preferredCandidate")
    private List<PreferredCandidate> preferredCandidates;
}
```

This is also defined as a Jackson XML Property and a Wrapper. There should be multiple parties in an “ElectionData” Object, therefore there needs to be a wrapping object that has a list of parties along with metadata like the regionID, etc.

> **Note**
>
> The XmlProperty Annotations are optional, the fields are registered as xml properties by default, only upon a name-change it’s required to use.

```
@Getter
@Setter
@JacksonXmlRootElement(localName = "electionData")
public class WarehouseData {

    private String regionID;
    private String regionName;
    private String regionAddress;
    private String regionPostalCode;
    private String federalState;

    @Setter(AccessLevel.NONE)
    private String timestamp;


    @JacksonXmlElementWrapper(localName = "countingData")
    @JacksonXmlProperty(localName  = "party")
    private List<PartyData> parties = new ArrayList<>();

    public WarehouseData() {
       this.timestamp = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss.SSS").format(new Date());
    }
    public void addParty(String partyId, String amountVotes, List<PreferredCandidate> preferre
       PartyData party = new PartyData(partyId, amountVotes, preferredCandidates);
       this.parties.add(party);
    }
```

Additionally, there's an addParty method where you can add parties dynamically. Now a simulator needs to be created that generates the dummy data.

I will not paste the entire file based on the fact that it's straightforward and repetetive, here's an example party:

```
data.addParty("OEVP", String.valueOf(getRandomInt(200, 500)), Arrays.asList(
       new PreferredCandidate("Karl Nehammer", String.valueOf(getRandomInt(10, 100))),
       new PreferredCandidate("Elisabeth Köstinger", String.valueOf(getRandomInt(10, 100)))
));
```

The generation is done inside of a `getData` method that accepts a `String inID` that is then being used as the regionID.

Now there's a service bean that calls the `getData` method whenever needed.

```
@Service
public class WarehouseService {
    public WarehouseData getWarehouseData( String inID ) {
        WarehouseSimulation simulation = new WarehouseSimulation();
        return simulation.getData(inID);
```

```
    }
}
```

It's being used by two controllers, one for the normal REST-Requests (xml and json page)

```
@GetMapping(value = "/warehouse/{inID}/data", produces = { MediaType.APPLICATION_JSON_VALUE, M
@Tag(name = "Raw Data")
public ResponseEntity<WarehouseData> warehouseData(@PathVariable String inID,
                                                   @RequestParam(value = "format", required =
    WarehouseData data = service.getWarehouseData(inID);

    HttpHeaders headers = new HttpHeaders();
    if ("xml".equalsIgnoreCase(format)) {
        headers.setContentType(MediaType.APPLICATION_XML);
    } else {
        headers.setContentType(MediaType.APPLICATION_JSON);
    }
    return new ResponseEntity<>(data, headers, HttpStatus.OK);
}
```

The only difference between json and xml is the `contentType`. It translates the xml to json and vice versa dynamically based on the `contentType`. The other controller is for the API-calls. It's being used by the static webpage displaying the table.

```
@RestController
@AllArgsConstructor
@RequestMapping("/api")
public class WarehouseAPIController {

    private final WarehouseService service;

    @GetMapping("/parties")
    @Tag(name = "API")
    public List<PartyData> getParties() {
        WarehouseData data = service.getWarehouseData("001");
        return data.getParties();
    }}
```

This route is being accessed by the html-file located in `resources -> static -> gridTable.html` where there's a javascript section that uses `gridjs` to display the table

```
<h1>National Elections Results</h1>

<div id="wrapper"></div>

<script>
  new gridjs.Grid({
    columns: [
```

```
      'Party Name',
      'Total Votes',
      'Candidate Name',
      'Candidate Votes'
    ],
    server: {
      url: '/api/parties',
      then: data => data.flatMap(party => {
        return party.preferredCandidates.map(candidate => [
          party.partyID,
          party.amountVotes,
          candidate.candidateName,
          candidate.candidateVotes
        ]);
      })},
    pagination: {
        enabled: true,
        limit: 10
    },
    search: true,
    sort: true
  }).render(document.getElementById("wrapper"));
</script>

</body>
```

This basically does an asynchronous call to the `/api/parties` route and maps the data

Tip

The `gridjs` library is being imported via

```
<link href="https://unpkg.com/gridjs/dist/theme/mermaid.min.css" rel="stylesheet" />
<script src="https://unpkg.com/gridjs/dist/gridjs.umd.js"></script>
```

in the html's head