Software and Tools

Below is a list of software and tools you will need for the purpose of this tutorial:

- **Postman** for executing RESTful API calls.
- **DBeaver** (or some other database visualisation tool)
- **IntelliJ** (or some other Java-capable IDE)
- **Java**
- **Maven**
- **MySQL Server**
- **Git and a GitHub Repository**

Project Setup

This assumes you have a repository in a location like `~/dev/personal/project` assuming a Unix file structure (Linux or MacOS). This doesn't cover creating a React app or anything else, other than strictly the API.

First, go to the the website `https://start.spring.io/` to begin using Spring's 'quick start' feature. A full list of modifications is as follows:

- **Project** - Maven
- **Language** - Java
- **Spring Boot** - 3.1.4 (or whatever version is default picked)
- **Project MetaData** - Customise accordingly. Pick **jar** and **java 17** for the Packaging and Java specifications.

Dependencies

I recommend the following dependencies:

- **Spring Web** because this is a Web API
- **LDAP** if you want to handle user login and user management
- **MySQL Database Driver** to handle databases.

Database

For each API you make, you will need a custom database for the project. I recommend using MySQL, because it is accessible on all machine types, and scales mostly 1:1 with Java itself, making it somewhat ideal for working with RESTful APIs. I recommend DBeaver to manage this database.

Package Structure

Make sure your package structure mimics the following structure:

```
src
    -> java
        -> com.sternberg.noah (literally whatever you want)
            -> models
                -> model relevant files
            -> db
                -> db relevant files
            -> controllers
                -> controller relevant files
            -> factories
                -> factory relevant files
            -> util (optional)
                -> All utility files
```

Database Package

In the database package we need two files: `Database.java` and `ConnectionSettings.java`. The database file will mimic our database and allow us to modify the database internally, and the connection settings file will handle the connnection to the database. For dev purposes, these are hard-coded values, but for prod scenarios, these will be some cloud-hosted database.

ConnectionSettings.java

Below is the exact connection settings file you'll want to use:

```
<package dec>

public class ConnectionSettings {
    private final static String CONNECTION_STRING = "jdbc:mysql://localhost/<database name>";
    private final static String ROOT_USERNAME = "root";
    private final static String ROOT_PASSWORD = "";
}
```

Database.java

In the database file, we need to construct the scripts that will create database tables as needed. To do this, I tend to do three things: write the scripts, store them in an array, and then write a function called `start()` to start up the database by executing all the scripts. This might look something like this:

```
<pacakage dec>
import static ConnectionSettings.CONNECTION_STRING;
import static ConnectionSettings.ROOT_USERNAME;
import static ConnectionSettings.ROOT_PASSWORD;

public class Database {

    private static String CREATE_TASK_TABLE = """
        CREATE TABLE IF NOT EXISTS taskmaster.tasks (
            id BIGINT NOT NULL UNIQUE AUTO_INCREMENT PRIMARY KEY,
            name VARCHAR(200) NOT NULL,
            description VARCHAR(200) NOT NULL,
            date BIGINT NOT NULL
        ) Engine = InnoDB;
    """;

    // Other Scripts

    private static String[] scripts = {CREATE_TASK_TABLE, // other scripts};

    public static void start() {
        for(String query : queries) {
            try(Connection connection = DriverManager.getConnection(
            CONNECTION_STRING, ROOT_USERNAME, ROOT_PASSWORD)) {
                Statement statement = connection.createStatement();
                statement.executeUpdate(query);
            }
            catch(Exception ex) {
                throw new RuntimeException(ex);
            }
        }
    }
}
```

Let's break this down bit by bit. First we construct our MySQL script, then add it to the array. Now let's look at the `start` function. At a high level, we iterate through our script and execute them in the database. The `try(Connection connection ...)` section establishes connection to the database. `Connection` is exactly as it sounds, but it uses the built-in Java `DriverManager`

to do it. Then we create a `Statement` to allow us to execute SQL scripts from within our established `Connection`. Then we call `executeUpdate` to finally call our script. This must be `try-catch` wrapped.

Structuring the API

Web APIs tend to (or at least in my experience with them) follow a very basic structure.

Models

Models are the simplest thing to construct by far. These represent actual things in the program. For instance, suppose I'm creating a task management app. I would need to make a model for a `task` in the code. If I'm making an API that deals with finance tracking, I might make a model for `expenses` and `deposits` etc. Some things remain the same for each model class you make.

- **1:1 with DB** : Every model should (with a few exceptions) have a 1:1 correspondence with the database definition of the model. You'll need a DB table for each model (we'll go more in depth on the DB in a separate section).
- **IDs** : Every model needs an id. This allows the DB to track these models accordingly. these should be private (like all fields) and a long (`private long id;`)
- **Getters** : All fields need `getters`. I don't tend to include `setters` but you can.
- **Constructors** : I always include 3 constructors, one for all fields, one for all fields minus the id (for internal tracking) and a private empty constructor for an override of `getInstance()`. More details in the example file

Example Model

Below is an example model for a task in TaskMaster:

```
public class Task {
    private long id;
    private long dueDate;
    private String name;
    private String description;

    // id included here
    public Task(long id, String name, long dueDate, String description) {
        this.id = id;
        // etc etc
    }

    // No id included here
    public Task(String name, long dueDate, String description) {
        this.name = name;
        // etc etc
    }

    private Task() {
    }

    // Used for unit testing purposes
    public static Task getInstance() {
        return new Task();
    }

    public long getId() {
        return this.id;
    }

    // All the other getters

}
```

This is it. This is all you need in a model. Setting them up this way allows Spring to take the objects and convert them nicely to a JSON format when returned to the front end, making them easily editable and machine (and human) readable.

Factories

Factories are used to handle the inherent modifications that you will need to make to the database. These are accessed from within the 'context' of the Controllers (this will make more sense later). For now, we will follow the following few rules:

- Every controller endpoint needs a factory function
- Every operation needs a function within the factory. Creating a model, getting a model, etc.
- Make all methods accessible **statically**.

Below is an example of a standard factory function for a TaskMaster controller:

```
<package declaration>
import static //.ConnectionSettings.*;
public class TaskFactory {

    // Take in the params you need to make a Task according to our DB definition
    public static Task createTask(String name, String description, long dueDate) {
        String hql =
            String.format("INSERT INTO taskmaster.tasks (name, description, date)" +
            "VALUES '%s', '%s', %d;", name, description, dueDate);

            try(Connection connection = DriverManager.getConnection(
            CONNECTION_STRING, ROOT_USERNAME, ROOT_PASSWORD)) {
                Statement statement = connection.createStatement();
                statement.executeUpdate(hql);
            }
            catch(Exception ex) {
                throw new RuntimeException(ex);
            }

            hql =
                String.format("SELECT * FROM taskmaster.tasks t" +
                "WHERE t.name = '%s' AND t.description = '%s' AND t.date = %d",
                name, description, date);

            try(Connection connection = DriverManager.getConnection(
            CONNECTION_STRING, ROOT_USERNAME, ROOT_PASSWORD)) {
                Statement statement = connection.createStatement();
                ResultSet set = statement.executeQuery(hql);

                while(set.next()) {
                    return new Task(
                        set.getLong("id"),
                        name, // or set.getString("name")
                        description,
                        dueDate
                    );
                }
            }
            catch(Exception ex) {
                throw new RuntimeException(ex);
            }
        return null;
    }
}
```

The general format for controller functions is to take in all the params you need to modify the database, then connect to the db, make the change, and return whatever is most appropriate (usually a model). Let's break down the above code bit by bit.

First, we statically import our ConnectionSettings. This stops us from needing to do `ConnectionSettings.CONNECTION_STRING` etc and allows us to, instead, just using `CONNECTION_SETTINGS`. Then, we define our function to be static (so we don't need to instantiate a TaskFactory object, rather just use `TaskFactory.createTask(// ...)`.

Then we define a `String hql`. I call this 'hql' because of my use of the Hibernate Query Language (HQL) in previous jobs. Feel free to call it "script" or whatever you fancy. When constructing this script, I use `String.format` to avoid having to concatenate Strings, or using StringBuilder. Anything is fine, use what you're most comfortable with. A little addendum on String.format is attached at the end of the doc.

We then establish connection to the database using the same technique we used when constructing the database connection in `Database.java`. Feel free to copy-paste the entire try-catch block to simplify things for you. After establishing connection, we make a `Statement` and then either `executeUpdate` or `executeQuery` depending on if we are doing an update or running a `ResultSet`-returning query. An addendum on this process is attached at the end of the doc.

If you are executing multiple queries, you can simply duplicate the try-catch block and create a fresh `statement`. The `Statement` class requires you to re-instantiate it after running each query, so you might be able to use one try-catch block, and just use `connection.createStatement()` to refresh it, but for parallelism, I tend to re-establish connection.

– This part left blank to allow better spacing –

Controllers

Controllers take the most work of all the API files in terms of unique Spring related things, but these are often the simplest classes to actually make. Each controller will correspond to the thing that it is modifying. For example, if we are modifiying Tasks, we would need a TaskController. If we are modifiying Deposit models, then we need a DepositController.
An addendum on all the annotations (@...) is attached at the end of the doc, but here is a sample controller:

```
<pacakge dec>

@RestController
@RequestMapping("/taskmaster/api/tasks")
public class TaskController {

    @GetMapping
    public List<Task> getAll() {
        return TaskFactory.getAll();
    }

    @GetMapping
    @RequestParam("/byId")
    public Task getTask(@RequestParam long id) {
        return TaskFactory.getById(id);
    }

    @PostMapping
    public Task createTask(
        @RequestParam String title,
        @RequestParam String description,
        @RequestParam long dueDate
    ) {
        return TaskFactory.create(title, description, dueDate);
    }

    @PutMapping
    public Task updateTask(
        @RequestParam long id,
        @RequestParam String title,
        @RequestParam String description,
        @RequestParam long dueDate
    ) {
        return TaskFactory.update(id, title, description, dueDate);
    }

    @DeleteMapping
    public boolean delete(@RequestParam long id) {
        return TaskFactory.delete(id);
    }
}
```

Now let's break this down. First and foremost, we tag our class as a `@RestController`. This is needed for the compiler to do its job while servicing the API. We also tag it with the annotation `@RequestMapping("/taskmaster/api/tasks")`. This is telling the API where to host the controller at. We structure our API in such a way when we go to `"<url>/taskmaster/api/tasks"` we reach this set of functions. We can then tag on any additional mappings (like `/byId`) to get to the needed functions.

We use the annotations `@GetMapping, @PostMapping, @PutMapping,` and `@DeleteMapping` to call the HTTP protocols `GET, POST, PUT`, and `DELETE` respectively. `GET` and `DELETE` are self explanatory, but `POST` puts a new item in the DB, while `PUT` *updates* an existing entry.

When we tag something as "`@RequestParam`" we are denoting that it belongs to the query, and we will get those from the front end when we call the API. This is true of every parameter to the controller classes.

Postman

When you have written a controller and are ready to test it, you can open Postman and use it to mock API calls. You can click the + icon to issue a new query. The URL to use while in dev mode is `http://localhost:8080/<your designation>`. So to test the TaskController we implemented in the previous example, we can go to `http://localhost:8080/taskmaster/api/tasks` (or `http://localhost:8080/taskmaster/api/tasks/byId` if we're testing the `getById()` function.

Set the protocol to whatever protocol you need (`GET, POST` etc) and then add the params you need. The `Key` must match the name of the param you set in the API. For instance, if you include a `@RequestParam dueDate` in the API, you must put "`dueDate`" in the `Key` field. The `Value` param can be whatever you want to pass to the API.

Then hit send. The API will either return the appropriate response, or a `Status Code`. There are a series of common status codes that you might want to know whilst debugging an API. `400` means you accessed a resource not mapped (i.e. put in a bad path) `500` means that there is an exception or error in the API that resulted from the call. `404` is what happens when you request something that doesn't exist (don't ask me how this differs from 400.)

Git from the Command Line

Using Git from the command line is the best way to use it. I reccomend, for all personal projects, focus on using the command line for all Git operations, while the pressure is low. If you get in a situation where things are all messed up, then use a GUI.

Here are all the commands you need to know to run git fluently from the command line:

- `git add .` to add all changes to be committed.
- `git add *.java` to add all files that end in ".java" (you can use this to add specific files
- `git commit -m "message"` to commit the added changes to the repository.
- `git push` to push your changes to the remote code repo.
- `git status` to show what has been added and what hasn't. (Press q to quit the status window)
- `git branch` to list branches
- `git branch <name>` to make a new branch
- `git checkout <branch>` to switch branches

Addenda

String.format

String.format is used to basically "slot" in variables into a String. We use weird quantifiers with percent signs.

- `%s` to swap in a String
- `%d` to swap in a double or int (I think)
- `%f` to swap in a float (maybe more I think)
- `%n` to include a new line.

executeUpdate vs executeQuery

Use `executeUpdate` to do things that you don't get results back from. Things like `UPDATE` and `INSERT INTO` statements. Use `executeQuery` where you get results (like `SELECT`).