**Problem.1 Arriving in Kathmandu**

*Your friend is a mountaineer and he needs your help. Your first task is to find him, so you went to Kathmandu and found some notes at his quarters.*



Write a program that **decrypts messages**, which containinformationaboutcoordinates. You are looking for **names of peaks** in the Himalayas and their [geohash](https://en.wikipedia.org/wiki/Geohash) coordinates. Keep reading lines until you receive the "**Last note**" message.

Here is your **cipher**:

* **Name of the peak**
  + It is consisted of **letters (upper and lower), numbers** and some of the following characters between its letters – "**!**" "**@**" "**#**" "**$**" "**?**". Example for valid names: “!@K?#2!#” (K2).
* **The length of the geohashcode**
  + Begins **after** the "**=**" (equals) sign and is consisted only of numbers.
* **The geohash code**
  + Begins after these symbols – "**<<**", may contain anything and the message always ends with it.

**Examples for valid input:**

"!Ma$$ka!lu!@=9<<ghtucjdhs" – all the components are there – **name of the peek**, **length** of the geohashcode and a **geohashcode**.

"!@Eve?#rest!#=7<<vbnfhfg"

**Examples of invalid input:**

"anna@fg<<jhsd@bx!=4" – **their order is wrong**. The name should be first, the length after and the code last.

"#n...s!n-<<tyuhgf4" – **the length is missing** and the **name contains dots.**

**"**Nan$ga!Parbat=8<<gh2tn – **the** **length** of the geohash code doesn't match the given number.

The **geohash code** you are looking for is with **length** **exactly** as much as the **given length** in the message and the information must be in the **exact given order**, otherwise it is considered **invalid**. If you find it, print the following message:

"**Coordinates found! {nameOfMountain} -> {geohashcode}**"

Otherwise print: “**Nothing found!**” after every **invalid** message.

## Input / Constraints

* You will be receiving strings until you get the “**Last note**” message.

## Output

* If you find the right coordinates, print: "**Coordinates found! {nameOfMountain} -> {geohashcode}**".
* If the message is invalid, print: "**Nothing found!**".

## Examples

|  |  |
| --- | --- |
| **Input** | **Output** |
| !@Ma?na?sl!u@=7<<tv58ycb4845  E!ve?rest=.6<<tuvz26  !K@2.,##$=4<<tvnd  !Shiha@pan@gma##9<<tgfgegu67  !###Anna@pur@na##=16<<tv5dekdz8x11ddkc  Last note | Nothing found!  Nothing found!  Nothing found!  Nothing found!  Coordinates found! Annapurna -> tv5dekdz8x11ddkc |
| **Comments** | |
| The first line is invalid, because the length – **7**, **doesn't** **match** the **length** of the **code**.  The second line is invalid, because the **length** should be consisted **only** of **numbers**.  The third line is invalid, because the name contains **symbols** that are **not** allowed – **".", ",".**  The forth line is invalid, because the **"="** sign before the length is **missing**.  The fifth line is valid, so we print the appropriate message. | |
|  | |
| Ka?!#nch@@en@ju##nga@=3<<thfbghvn  =9Cho?@#Oyu<<thvb7ydht  Nan??ga#Par!ba!t?=16<<twm03q2rx5hpmyr6  Dhau??la#gi@ri?!#=3<<bvnfhrtiuy  Last note | Nothing found!  Nothing found!  Coordinates found! NangaParbat -> twm03q2rx5hpmyr6  Nothing found! |

**Problem 2. On the Way to Annapurna**

*You’ve hired a Sherpa and he has a list of supplies you both need to go on the way. He has passed you some notes and you have to order them correctly in a diary before you start circling around the town’s stores.*



Create a program, that lists **stores** and the **items** that can be found in them. You are going to be receiving **commands** with the information you need until you get the "**End**" command. There are **three possible commands**:

* "**Add**->{Store}->{Item}"
  + **Add** the **store** and the **item** in your diary. If the store already **exists**, add just the item.
* **"Add**->{Store}->{Item},{Item1}…,{ItemN}"
  + **Add the store and the items to** your notes. **If the store already exists** in the diary – **add just the items** to it.
* "**Remove**->{Store}"
  + **Remove the store** and its items from your diary, **if it exists**.

In the end, print the collection **sorted by the count of the items** in **descending order** and **then by the names of the stores**, again, **in descending order** in the following format:

**Stores list:**

**{Store}**

**<<{Item}>>**

**<<{Item}>>**

**<<{Item}>>**

## Input / Constraints

* You will be receiving information until the “**END**” command is given.
* There will always be **at least one** store in the diary.
* Input will always be **valid**, there is no need to check it explicitly.

## Output

* Print the list of stores in the format given above.

## Examples

|  |  |
| --- | --- |
| **Input** | **Output** |
| Add->PeakSports->Map,Navigation,Compass  Add->Paragon->Sunscreen  Add->Groceries->Dried-fruit,Nuts  Add->Groceries->Nuts  Add->Paragon->Tent  Remove->Paragon  Add->Pharmacy->Pain-killers  END | Stores list:  PeakSports  <<Map>>  <<Navigation>>  <<Compass>>  Groceries  <<Dried-fruit>>  <<Nuts>>  <<Nuts>>  Pharmacy  <<Pain-killers>> |
| **Comments** | |
| First, we receive the "**Add**" command with a couple of items and we have to add the store and the items to. We keep doing that for each line of input and when we receive the "**Remove**" command, we delete the store and its items from our records. In the end we print the stores sorted by the **count** of their **items** and **then by** their **names**. | |
|  | |
| Add->Peak->Waterproof,Umbrella  Add->Groceries->Water,Juice,Food  Add->Peak->Tent  Add->Peak->Sleeping-Bag  Add->Peak->Jacket  Add->Groceries->Lighter  Remove->Groceries  Remove->Store  END | Stores list:  Peak  <<Waterproof>>  <<Umbrella>>  <<Tent>>  <<Sleeping-Bag>>  <<Jacket>> |

**Basic CRUD – Himalayas Rescue Register**



*You have received information from a Sherpa, who was on his way back from Annapurna, that your friend was seen with a group of climbers a few days ago and you need to register all the information you can gather for them and give it to the mountain rescue team. You can’t stay and help them, you have to go as soon as possible, so you’ve decided to create a program that registers climbers and keeps the needed information about them that could help in some way.*

You have been tasked to create a simple application for registering missing people in the mountains. The application should hold **mountaineers**, which are the main app **entities**. The app is called **RescueRegister**.

The functionality of the application should support **creating**, **listing, editing**, **deleting** mountaineers.

The application should **persist** the data into a **database**.

**Overview**

Your application should be built on **one** of the **following technologies**:

**JavaScript**

* **NodeJS** + **ExpressJS** frameworks
* **Handlebars.js** view engine
* **Mongoose** ORM
* **MongoDB**

**PHP**

* **Symfony** framework
* **Twig** view engine
* **Doctrine** ORM
* **MySQL** database

**Java**

* **Spring** framework (**Spring MVC** + **Spring Boot** + **Spring Data**)
* **Thymeleaf** view engine
* **JPA** / **Hibernate ORM** + **Spring Data** data access
* **MySQL** database

**C#**

* **ASP.NET Core** framework (**ASP.NET MVC** + **Entity Framework Core**)
* **Razor** view engine
* **Entity Framework Core** ORM
* **SQL Server** database\

**Data Model**

The **Mountaineer** entity holds **5 properties**:

* **id** – technology-dependent identifier (**ObjectID** for JavaScript, **int** for all other technologies)
* **name** – non-empty text
* **age** – non-null integer number
* **gender** – non-empty text
* **lastSeenDate** – non-empty text

**Project Skeletons**

You will be given the applications’ **skeletons**, which hold about **90%** of the logic. You’ll be given some **files** (**controllers**, **models** etc.). The files will have **partially implemented logic**, so you’ll need to write some code for the application to **function properly**.

The application’s views will be given to you fully implemented. You only need to include them in your business logic.

**Each technology** will have its **own skeleton**, and the **different** **skeletons** may **differ** in **terms** of **what is given to you** and **what is to be implemented**.

Everything that has been given to you inside the skeleton is **correctly implemented** and if you write your code **correctly**, the application should work just fine. You are free to change anything in the Skeleton on your account.

**User Interface**

This is the user interface or how the application’s pages should look in their final form (fully implemented). You have several pages, described below:

**Index Page**

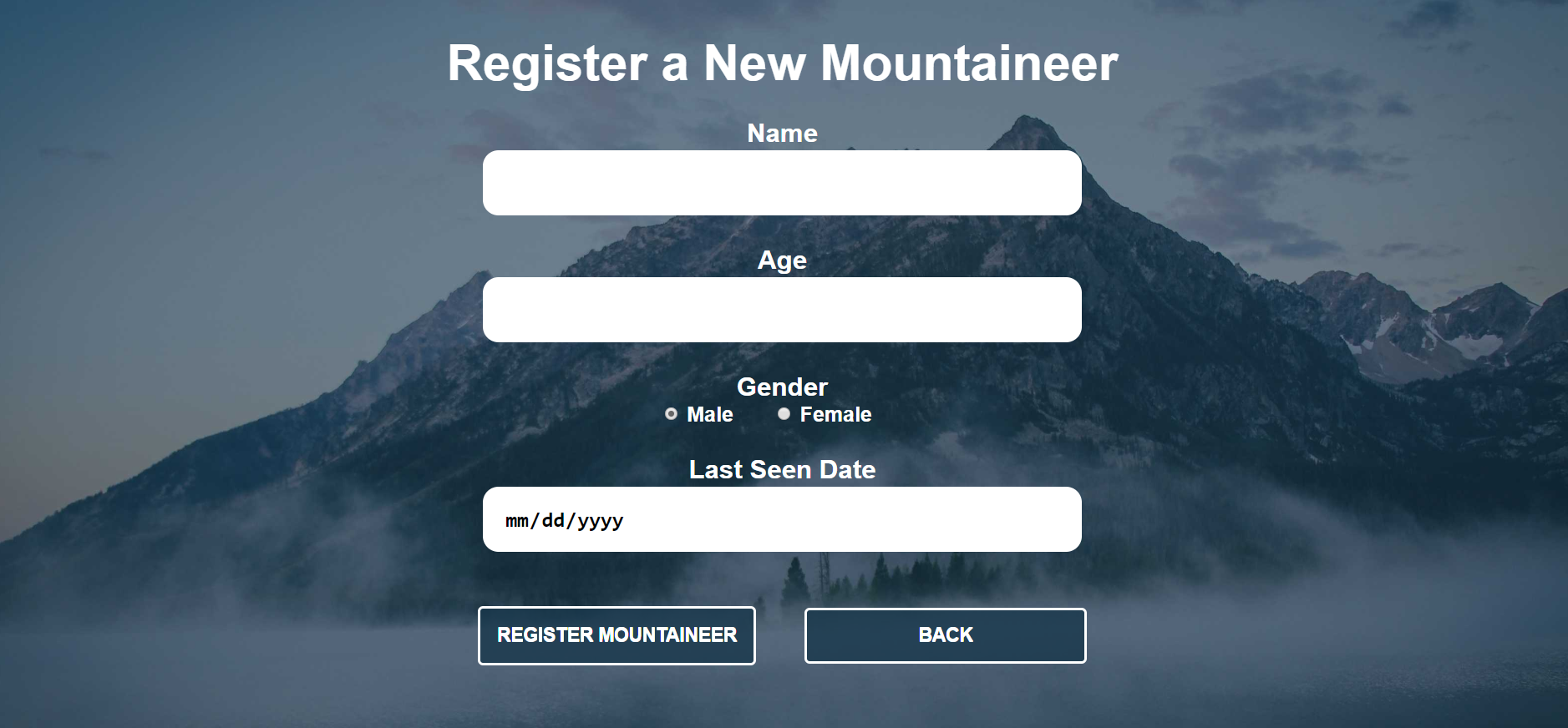
**Route: "/"**

Displays **all** the **mountaineers** from the database with **options** to **edit** or **delete** them.



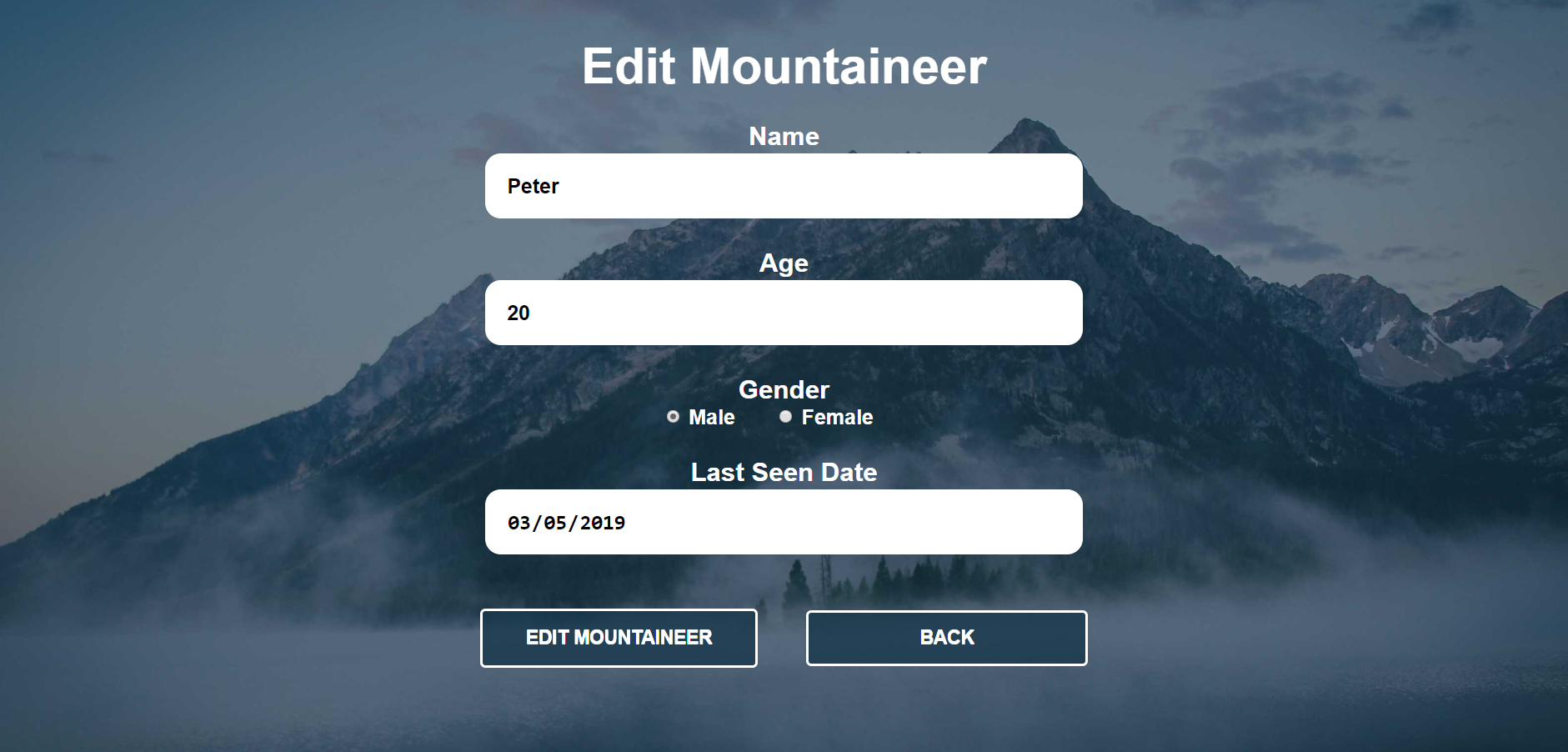
**Create Page**

**Route: "/create"**



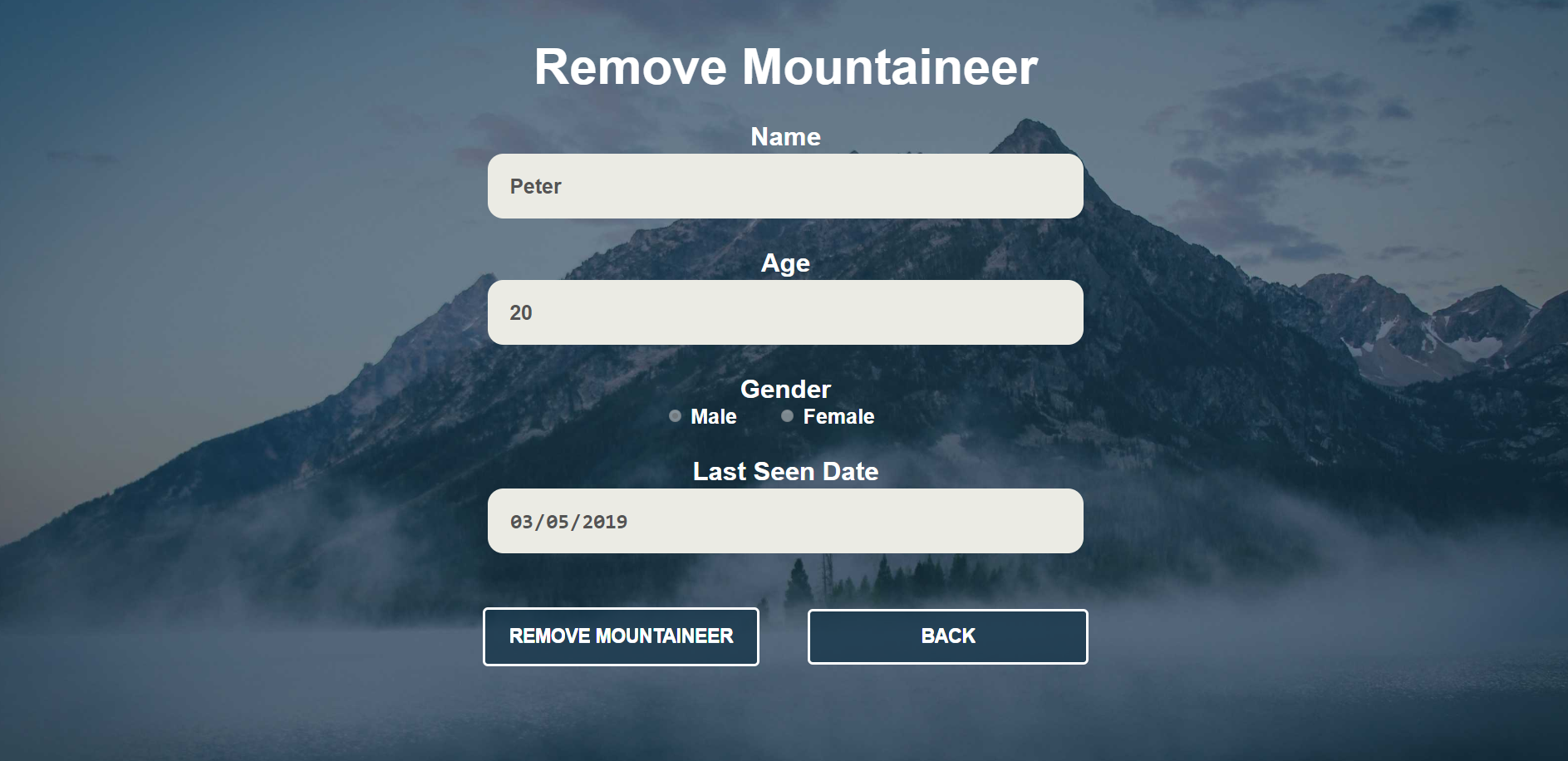
**Edit Page**

**Route: "/edit/{id}"**



**Delete Page**

**Route: "/delete/{id}"**



**Problem**

As you can see the different pages are on different routes. Most of the routing logic will be given to you in the **Skeleton**, but you should make sure that the application **works properly**.

Implement the "**RescueRegister" app** using only **your technology.**

**Setup**

Before you start working, make sure you **download all the dependencies** (packages) required for your technology and **set up** the **databases**! Below are instructions on how to do this:

**PHP and Symfony**

1. Make sure you've started your **MySQL server** (either from **XAMPP** or standalone)
2. Open a **Terminal in PHPStorm** or **shell** / **command prompt** / **PowerShell** window in the **root directory**: [Shift] + [Right click] 🡺 [Open command window here]
3. Enter the "**composer install**" command to restore its **Composer dependencies**   
   (described in **composer.json**)
4. Enter the "**php bin/console doctrine:database:create --if-not-exists**" command
5. Done!

**JavaScript and Node.js**

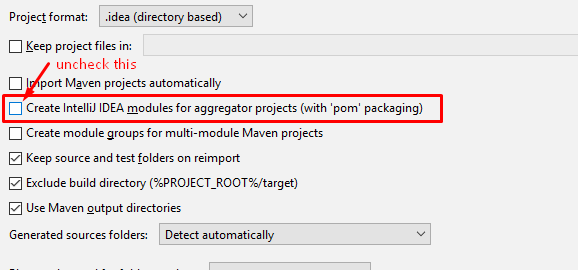
1. Go into the **root directory** of the project (where the **index.js** file is)
2. Open a **shell** / **command prompt** / **PowerShell** window in the **root directory**: [Shift] + [Right click] 🡺 [Open command window here]
3. Enter the “**npm install**” command to restore its **Node.js dependencies** (described in **package.json**)
4. Type **node index.js** to start the server
5. Done!

**C# and ASP.NET**

The C# project will automatically resolve its **NuGet dependencies** (described in **packages.config**) using the NuGet package restore when the project is built.

**Java and Spring MVC**

When you import your project, you should **uncheck "**Create IntelliJ IDEA modules for aggregator projects (with 'pom' packaging)**"**:



This project is **set up to use Java jdk 1.8.** If your version is different, you can change it in **Maven dependencies** like this:



The Java project will automatically resolve its **Maven dependencies** (described in **pom.xml**) when the project is built.