Building a Full-Stack App with Angular, PHP and MySQL

You'll create an example REST API CRUD Angular application with PHP and a MySQL backend.

You will be creating a simple RESTful API that supports GET, POST, PUT and DELETE requests and allow you to perform CRUD operations against a MySQL database to create, read, update and delete records from a database.

For the application design, It's a simple interface for working with vehicle insurance policies. For the sake of simplicity, you are only going to add the following attributes to the `policies` database table:

- `number` which stores to the insurance policy number,
- `amount` which stores the insurance amount.

This is of course far from being a complete database design for a fully working insurance system. Because at least you need to add other tables like employees, clients, coverage, vehicles, and drivers, etc. And also the relationships between all these entities.

Prerequisites

IWe'll assume you have the following prerequisites:

- The MySQL database management system installed on your development machine,
- PHP installed on your system (both these first requirements are required by the back-end project),
- Node.js 8.9+ and NPM installed in your system. This is only required by your Angular project.
- You also need to have a working experience of PHP and the various functions that will be used to create the SQL connection, getting the GET and POST data and returning JSON data in your code.

- You need to be familiar with TypeScript.
- Basic knowledge of Angular is preferable but not required since you'll go from the first step until you create a project that communicates with a PHP server.

Creating the PHP Application

Let's start by creating a simple PHP script that connects to a MySQL database and listens to API requests then responds accordingly by either fetching and returning data from the SQL table or insert, update and delete data from the database.

Go ahead and create a folder structure for your project using the following commands:

```
$ mkdir angular-php-app
$ cd angular-php-app
$ mkdir backend
```

We create the `angular-php-app` that will contain the full front-end and back-end projects. Next, we navigate inside it and create the `backend` folder that will contain a simple PHP script for implementing a simple CRUD REST API against a MySQL database.

Next, navigate into your `backend` project and create an `api` folder.

```
$ cd backend
$ mkdir api
```


Inside the `api` folder, create the following files:

```
$ cd api
$ touch database.php
$ touch read.php
$ touch create.php
$ touch update.php
$ touch delete.php
```

Open the `backend/api/database.php` file and add the following PHP code step by step:

```php
<?php
header("Access-Control-Allow-Origin: *");
header("Access-Control-Allow-Methods: PUT, GET, POST, DELETE");
header("Access-Control-Allow-Headers: Origin, X-Requested-With, Content-Type, Accept\
");
```

These lines are used to add response headers such as CORS and the allowed methods (PUT, GET, DELETE and POST).

Setting CORS to * will allow your PHP server to accept requests from another domain where the Angular 7 server is running from without getting blocked by the browser because of the **Same Origin Policy**. In development, you'll be running the PHP server from `localhost:8080` port and Angular from `localhost:4200` which are considered as two distinct domains.

Next, add:

```php
define('DB_HOST', 'localhost');
define('DB_USER', 'root');
define('DB_PASS', 'YOUR_PASSWORD');
define('DB_NAME', 'mydb');
```

These variables hold the credentials that will be used to connect to the MySQL database and the name of the database.

> **Note**: Make sure you change them to your actual MySQL credentials. Also make sure you have created a database with a policies table that has two `number` and `amount` columns.

Next, add:

```php
function connect()
{
  $connect = mysqli_connect(DB_HOST ,DB_USER ,DB_PASS ,DB_NAME);

  if (mysqli_connect_errno($connect)) {
    die("Failed to connect:" . mysqli_connect_error());
  }

  mysqli_set_charset($connect, "utf8");

```

```
  return $connect;
}

$con = connect();
```

This will allow you to create a connection to the MySQL database using the `mysqli` extension.

That's all for the `database.php` file

Implementing the Read Operation

Now, let's implement the read operation. Open the `backend/api/read.php` file and add the following code:

```
<?php
/**
 * Returns the list of policies.
 */
require 'database.php';

$policies = [];
$sql = "SELECT id, number, amount FROM policies";

if($result = mysqli_query($con,$sql))
{
  $i = 0;
  while($row = mysqli_fetch_assoc($result))
  {
    $policies[$i]['id']    = $row['id'];
    $policies[$i]['number'] = $row['number'];
    $policies[$i]['amount'] = $row['amount'];
    $i++;
  }

  echo json_encode($policies);
}
else
{
  http_response_code(404);
}
```

This will fetch the list of policies from the database and return them as a JSON response. If there is an error it will return a 404 error.

Implementing the Create Operation

Let's now implement the create operation. Open the `backend/api/create.php` file and add the following code:

```
<?php
require 'database.php';

// Get the posted data.
$postdata = file_get_contents("php://input");

if(isset($postdata) && !empty($postdata))
{
  // Extract the data.
  $request = json_decode($postdata);


  // Validate.
  if(trim($request->number) === '' || (float)$request->amount < 0)
  {
    return http_response_code(400);
  }

  // Sanitize.
  $number = mysqli_real_escape_string($con, trim($request->number));
  $amount = mysqli_real_escape_string($con, (int)$request->amount);


  // Create.
  $sql = "INSERT INTO `policies`(`id`,`number`,`amount`) VALUES (null,'{$number}','{\
$amount}')";

  if(mysqli_query($con,$sql))
  {
    http_response_code(201);
    $policy = [
      'number' => $number,
      'amount' => $amount,
```

```
      'id'    => mysqli_insert_id($con)
    ];
    echo json_encode($policy);
  }
  else
  {
    http_response_code(422);
  }
}
```

Implementing the Update Operation

Open the backend/api/update.php file and add the following code:

```
<?php
require 'database.php';

// Get the posted data.
$postdata = file_get_contents("php://input");

if(isset($postdata) && !empty($postdata))
{
  // Extract the data.
  $request = json_decode($postdata);

  // Validate.
  if ((int)$request->id < 1 || trim($request->number) == '' || (float)$request->amou\
nt < 0) {
    return http_response_code(400);
  }

  // Sanitize.
  $id     = mysqli_real_escape_string($con, (int)$request->id);
  $number = mysqli_real_escape_string($con, trim($request->number));
  $amount = mysqli_real_escape_string($con, (float)$request->amount);

  // Update.
  $sql = "UPDATE `policies` SET `number`='$number',`amount`='$amount' WHERE `id` = '\
{$id}' LIMIT 1";

```

```
  if(mysqli_query($con, $sql))
  {
    http_response_code(204);
  }
  else
  {
    return http_response_code(422);
  }
}
```

Implementing the Delete Operation

Open the `backend/api/delete.php` file and add the following code:

```
<?php

require 'database.php';

// Extract, validate and sanitize the id.
$id = ($_GET['id'] !== null && (int)$_GET['id'] > 0)? mysqli_real_escape_string($con\
, (int)$_GET['id']) : false;

if(!$id)
{
  return http_response_code(400);
}

// Delete.
$sql = "DELETE FROM `policies` WHERE `id` ='{$id}' LIMIT 1";

if(mysqli_query($con, $sql))
{
  http_response_code(204);
}
else
{
  return http_response_code(422);
}
```

In all operations, we first require the `database.php` file for connecting to the MySQL database and then we implement the appropriate logic for the CRUD operation.

Serving the PHP REST API Project

You can next serve your PHP application using the built-in development server using the following command:

```
$ php -S 127.0.0.1:8080 -t ./angular-php-app/backend
```

This will run a development server from the `127.0.0.1:8080` address.

Creating the MySQL Database

In your terminal, run the following command to start the MySQL client:

```
$ mysql -u root -p
```

The client will prompt for the password that you configured when installing MySQL in your system.

Next, run this SQL query to create a `mydb` database:

```
mysql> create database mydb;
```

Creating the `policies` SQL Table

Next create the `policies` SQL table with two `number` and `amount` columns:

```
mysql> create table policies( id int not null auto_increment, number varchar(20), am\
ount float, primary key(id));
```

Now, you are ready to send GET, POST, PUT and DELETE requests to your PHP server running from the `127.0.0.1:8080` address.

For sending test requests, you can use REST clients such as Postman or cURL before creating the Angular UI.

Leave your server running and open a new terminal.

Wrap-up

In this section, you have created a PHP RESTful API that can be used to execute CRUD operations against a MySQL database to create, read, update and delete insurance policies.

You have enabled CORS so you can use two domains `localhost:8000` and `localhost:4200` for respectively serving your PHP and Angular apps and being able to send requests from Angular to PHP without getting blocked by the Same Origin Policy rule in modern web browsers.

Creating the Angular Front-End

In this section, you'll learn how to use `HttpClient` to make HTTP calls to a REST API and use template-based forms to submit data.

Now, that you've created the RESTful API with a PHP script, you can proceed to create your Angular project.

Installing Angular CLI

The recommended way of creating Angular projects is through using Angular CLI, the official tool created by the Angular team. The latest and best version yet is Angular CLI 8 so head back to another terminal window and run the following command to install the CLI:

```
$ npm install -g @angular/cli
```

> **Note**: This will install Angular CLI globally so make sure you have configured npm to install packages globally without adding `sudo` in Debian systems and macOS or using an administrator command prompt on Windows. You can also just fix your npm permissions if you get any issues

Generating a New Project

That's it! You can now use the CLI to create an Angular project using the following command:

```
$ cd angular-php-app
$ ng new frontend
```

The CLI will ask you if **Would you like to add Angular routing?** type **y** because we'll need routing setup in our application. And **Which stylesheet format would you like to use?** Select **CSS**.

Wait for the CLI to generate and install the required dependencies and then you can start your development server using:

```
$ cd frontend
$ ng serve
```

You can access the frontend application by pointing your browser to the `http://localhost:4200` address.

Setting up `HttpClient` & Forms

Angular provides developers with a powerful HTTP client for sending HTTP requests to servers. It's based on the `XMLHttpRequest` interface supported on most browsers and has a plethora of features like the use of RxJS Observable instead of callbacks or promises, typed requests and responses and interceptors.

You can set up `HttpClient` in your project by simply importing the `HttpClientModule` in your main application module.

We'll also be using a template-based form in our application so we need to import `FormsModule`.

Open the `src/app/app.module.ts` file and import `HttpClientModule` then ad it to the `imports` array of `@NgModule`:

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
import { HttpClientModule } from '@angular/common/http';
import { FormsModule } from  '@angular/forms';

import { AppRoutingModule } from './app-routing.module';
import { AppComponent } from './app.component';

@NgModule({
  declarations: [
    AppComponent
  ],
  imports: [
    BrowserModule,
    HttpClientModule,
    FormsModule,
    AppRoutingModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

That's it! You can now use `HttpClient` in your components or services via dependency injection.

Creating an Angular Service

Let's now, create an Angular service that will encapsulate all the code needed for interfacing with the RESTful PHP backend.

Open a new terminal window, navigate to your `frontend` project and run the following command:

```
$ ng generate service api
```

This will create the `src/app/api.service.spec.ts` and `src/app/api.service.ts` files with a minimum required code.

Open the `src/app/api.service.ts` and start by importing and injecting `HttpClient`:

```
import { Injectable } from '@angular/core';
import { HttpClient } from '@angular/common/http';

@Injectable({
  providedIn: 'root'
})
export class ApiService {

  constructor(private httpClient: HttpClient) {}
}
```

We inject `HttpClient` as a private `httpClient` instance via the service constructor. This is called **dependency injection**. If you are not familiar with this pattern. Here is the definition from Wikipedia:

> In software engineering, dependency injection is a technique whereby one object (or static method) supplies the dependencies of another object. A dependency is an object that can be used (a service). An injection is the passing of a dependency to a dependent object (a client) that would use it. The service is made part of the client's state.[1] Passing the service to the client, rather than allowing a client to build or find the service, is the fundamental requirement of the pattern.

Also, this is what Angular docs says about dependency injection in Angular:

> Dependency injection (DI), is an important application design pattern. Angular has its own DI framework, which is typically used in the design of Angular applications to increase their efficiency and modularity. Dependencies are services or objects that a class needs to perform its function. DI is a coding pattern in which a class asks for dependencies from external sources rather than creating them itself.

You can now use the injected `httpClient` instance to send HTTP requests to your PHP REST API.

Creating the Policy Model

Create a policy.ts file in the `src/app` folder of your project and add the following TypeScript class:

```
export  class  Policy {
    id: number;
    number:  number;
    amount:  number;
}
```

Defining the CRUD Methods

Next, open the `src/app/api.service.ts` file and import the `Policy` model and the RxJS `Observable` interface:

```
import { Policy } from  './policy';
import { Observable } from  'rxjs';
```

Next, define the `PHP_API_SERVER` variable in the service:

```
export class ApiService {
  PHP_API_SERVER = "http://127.0.0.1:8080";
```

The `PHP_API_SERVER` holds the address of the PHP server.

Next, add the `readPolicies()` method that will be used to retrieve the insurance policies from the REST API endpoint via a GET request:

```
  readPolicies(): Observable<Policy[]>{
    return this.httpClient.get<Policy[]>(`${this.PHP_API_SERVER}/api/read.php`);
  }
```

Next, add the `createPolicy()` method that will be used to create a policy in the database:

```
  createPolicy(policy: Policy): Observable<Policy>{
    return this.httpClient.post<Policy>(`${this.PHP_API_SERVER}/api/create.php`, pol\
icy);
  }
```

Next, add the `updatePolicy()` method to update policies:

```
  updatePolicy(policy: Policy){
    return this.httpClient.put<Policy>(`${this.PHP_API_SERVER}/api/update.php`, poli\
cy);
  }
```

Finally, add the `deletePolicy()` to delete policies from the SQL database:

```
  deletePolicy(id: number){
    return this.httpClient.delete<Policy>(`${this.PHP_API_SERVER}/api/delete.php/?id\
=${id}`);
  }
```

That's all for the service.

Creating the Angular Component

After creating the service that contains the CRUD operations, let's now create an Angular component that will call the service methods and will contain the table to display policies and a form to submit a policy to the PHP backend.

head back to your terminal and run the following command:

```
$ ng generate component dashboard
```

Let's add this component to the Router. Open the `src/app/app-routing.module.ts` file and add a `/dashboard` route:

```
import { NgModule } from '@angular/core';
import { Routes, RouterModule } from '@angular/router';
import { DashboardComponent } from './dashboard/dashboard.component';


const routes: Routes = [
  { path: 'dashboard', component: DashboardComponent }
];

@NgModule({
  imports: [RouterModule.forRoot(routes)],
  exports: [RouterModule]
})
export class AppRoutingModule { }
```

You can now access your dashboard component from the `127.0.0.1:4200/dashboard` URL. This is a screenshot of the page at this point:

Let's remove the boilerplate content added by Angular CLI. Open the `src/app/app.component.html` file and update accordingly:

```
<router-outlet></router-outlet>
```

We only leave the router outlet where Angular router will insert the matched component(s).

Next, open the `src/app/dashboard/dashboard.component.ts` file and import `ApiService` then inject it via the component constructor:

```
import { Component, OnInit } from '@angular/core';
import { ApiService } from '../api.service';

@Component({
  selector: 'app-dashboard',
  templateUrl: './dashboard.component.html',
  styleUrls: ['./dashboard.component.css']
})
export class DashboardComponent implements OnInit {

  constructor(private apiService: ApiService) { }

  ngOnInit() {
  }
}
```

We inject `ApiService` as a private `apiService` instance.

Next, let's define the `policies` array that will hold the insurance policies once we get them from the server after we send a `GET` request and also the `selectedPolicy` variable that will hold the selected policy from the table.

```
export class DashboardComponent implements OnInit {
    policies: Policy[];
    selectedPolicy: Policy = { id : null , number:null, amount: null};
```

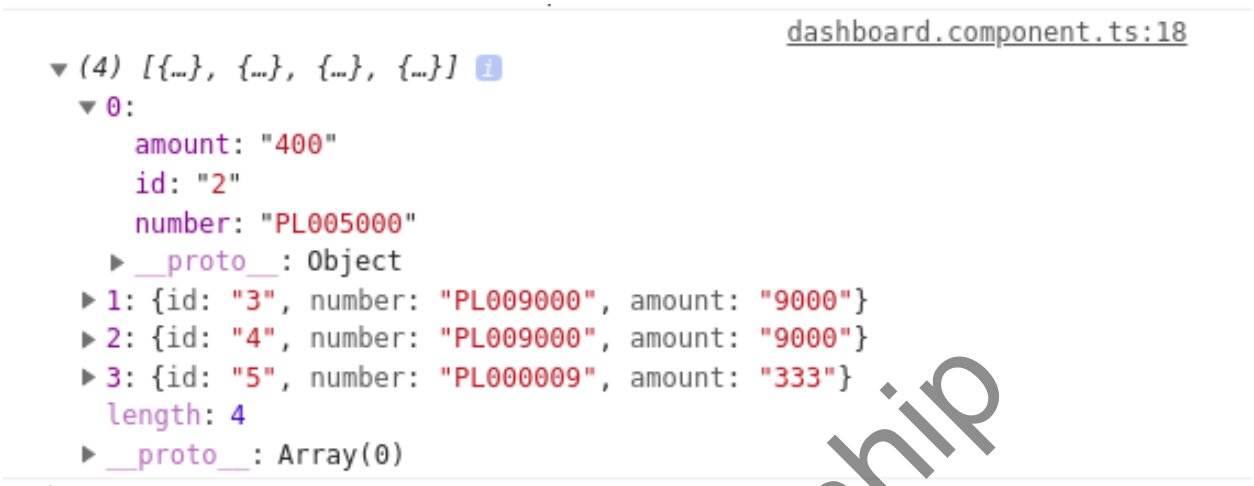
On the `ngOnInit()` method of the component, let's call the `readPolicies()` method of `ApiService` to get the policies:

```
  ngOnInit() {
    this.apiService.readPolicies().subscribe((policies: Policy[])=>{
      this.policies = policies;
      console.log(this.policies);
    })
  }
```

We call the `readPolicies()` which return an `Observable<Policy[]>` object and we subscribe to the RxJS Observable. We then assign the returned policies to the `policies` array of our component and we also log the result in the console.

> Note: The actual HTTP request is only sent to the server when you subscribe to the returned Observable.

You should see the returned policies in your browser console. This is a screenshot of the output in my case:



Angular PHP example

We'll see a bit later how to display these policies in a table in the component template.

Let's add the other methods to create, update and delete policies in our component.

```
createOrUpdatePolicy(form){
  if(this.selectedPolicy && this.selectedPolicy.id){
    form.value.id = this.selectedPolicy.id;
    this.apiService.updatePolicy(form.value).subscribe((policy: Policy)=>{
      console.log("Policy updated" , policy);
    });
  }
  else{

    this.apiService.createPolicy(form.value).subscribe((policy: Policy)=>{
      console.log("Policy created, ", policy);
    });
  }

}

selectPolicy(policy: Policy){
  this.selectedPolicy = policy;
```

```
  }

  deletePolicy(id){
    this.apiService.deletePolicy(id).subscribe((policy: Policy)=>{
      console.log("Policy deleted, ", policy);
    });
  }
```

Adding the Table and Form

Let's now add a table and form to display and create the policies in our dashboard component. Open the `src/app/dashboard/dashboard.component.html` and add the following HTML code:


```
<h1>Insurance Policy Management</h1>
<div>

  <table border='1' width='100%' style='border-collapse: collapse;'>
    <tr>
      <th>ID</th>
      <th>Policy Number</th>
      <th>Policy Amount</th>
      <th>Operations</th>

    </tr>

    <tr *ngFor="let policy of policies">
      <td>{{ policy.id }}</td>
      <td>{{ policy.number }}</td>
      <td>{{ policy.amount }}</td>
      <td>
        <button (click)="deletePolicy(policy.id)">Delete</button>
        <button (click)="selectPolicy(policy)">Update</button>
      </td>
    </tr>
  </table>
```

This is a screenshot of the page at this point:

Insurance Policy Management

ID	Policy Number	Policy Amount	Operations
2	PL005000	400	Delete Update
3	PL009000	9000	Delete Update
4	PL009000	9000	Delete Update
5	PL000009	333	Delete Update

PHP Angular Example

Next, below the table, let's add a form that will be used to create and update a policy:

```
<br>
  <form #f = "ngForm">
    <label>Number</label>
    <input type="text" name="number" [(ngModel)] = "selectedPolicy.number">
    <br>
    <label>Amount</label>
    <input type="text" name="amount" [(ngModel)] = "selectedPolicy.amount">
    <br>
    <input type="button" (click)="createOrUpdatePolicy(f)" value="Create or Update P\
olicy">
  </form>
```

This is a screenshot of the page at this point:

Insurance Policy Management

ID	Policy Number	Policy Amount	Operations
2	PL005000	400	Delete Update
3	PL009000	9000	Delete Update
4	PL009000	9000	Delete Update
5	PL000009	333	Delete Update

Number
Amount
Create or Update Policy

Angular PHP CRUD Example

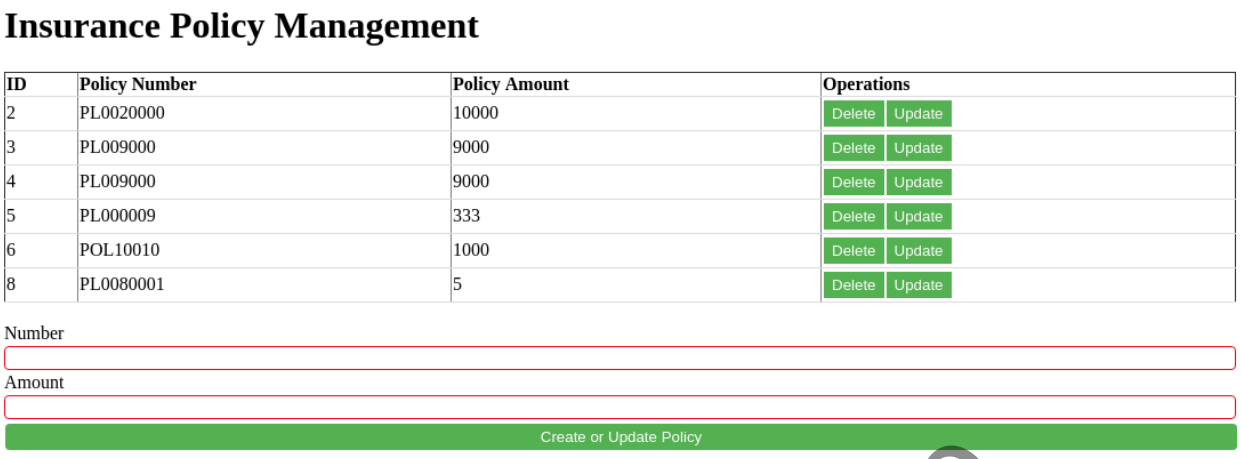
Next, open the `src/app/dashboard/dashboard.component.css` file and the following CSS styles:

```
input {
    width: 100%;
    padding: 2px 5px;
    margin: 2px 0;
    border: 1px solid red;
    border-radius: 4px;
    box-sizing: border-box;
  }

  button, input[type=button]{
    background-color: #4CAF50;
    border: none;
    color: white;
    padding: 4px 7px;
    text-decoration: none;
    margin: 2px 1px;
    cursor: pointer;
  }
  th, td {
    padding: 1px;
    text-align: left;
    border-bottom: 1px solid #ddd;

  }
  tr:hover {background-color: #f5f5f5;}
```

This is a screenshot of the page after adding some minimal styling:



Insurance Policy Management

ID	Policy Number	Policy Amount	Operations
2	PL0020000	10000	Delete Update
3	PL009000	9000	Delete Update
4	PL009000	9000	Delete Update
5	PL000009	333	Delete Update
6	POL10010	1000	Delete Update
8	PL0080001	5	Delete Update

Number

Amount

Create or Update Policy

PHP Angular REST CRUD Example

Conclusion

In this project, you learned how to create a RESTful CRUD application with PHP, MySQL, and Angular.