Copilot Java – PL2 – Pet Clinic
Guided Exercise

Contents

Chapter – 1

About the Project

1. Guided Exercise Details

Skill	GitHub Copilot
Proficiency	Level 2
Duration	16 hours
Version	1

2. About this guided exercise

Within these guided exercises, we aim to enhance the Spring development team's Pet Clinic sample project. These improvements will be facilitated through the utilization of the **AI Tool GitHub Copilot**. The Pet Clinic project, designed in various technology stacks, will specifically focus on REST and Angular implementations. The base code for these projects can be found at the following GitHub repositories:

Petclinic GitHub Repo
1. Angular - https://github.com/spring-petclinic/spring-petclinic-angular 2. REST - https://github.com/spring-petclinic/spring-petclinic-rest

Instructions on setting up the project will be furnished later in this document. All functionalities implemented in the mentioned GitHub projects are detailed below.

Existing Features related to Owners.

1. View list of Pet Owners
2. Add a new Pet Owner capturing the following details.
    a. First Name
    b. Last Name
    c. Address
    d. City
    e. Telephone

3. Find Pet Owners by Last Name
4. View Pet Owner details
    a. Name, Address and Mobile Number
    b. Pets owned by the Owner containing the following details about the Pet
        i. Pet Name
        ii. Pet Date of Birth
        iii. Pet Type (Cat, Dog, Bird, etc.)
    c. Visits made to the clinic for each Pet with following details:
        i. Visit Date
        ii. Description
        iii. Actions
5. Following operations can be performed about the Owner:
    a. Edit Owner details.
    b. Add new Pet for the Owner
6. Following operations can be performed on an Owner's pet:
    a. Edit Pet details.
    b. Delete Pet
    c. Add Visit

Existing Features related to Veterinarians.

1. View list of Veterinarians with Name, List of Specialties of the Veterinarian
2. Add new Veterinarian with First Name, Last Name and Specialty
3. Edit Veterinarian details for First Name, Last Name, and the possibility to select multiple specialties.
4. Delete Veterinarian

Existing Features related to Pet Type

1. View list of all Pet Types
2. Add a Pet Type
3. Edit a Pet Type
4. Remove a Pet Type

Existing Features related to Specialties.

1. View list of all Specialties
2. Add a Specialty
3. Edit a Specialty
4. Remove a Specialty

Features to be implemented as part of this exercise using copilot.

1. Additionally capturing email field for owner
2. Add Pet Treatment Type
3. Edit Pet Treatment Type
4. Delete Pet Treatment Type

Following topics will be covered to demonstrate capabilities of copilot.

1. Programming Fundamentals
    - Data Types, Constant, Variables, Array, Loops, Classes Objects, Object Oriented Concepts and Streams
2. Design Patterns
    - Singleton, Abstract Factory, Builder
3. Generating Join SQL Queries

Chapter – 2

Technology Stack

Technology Stack

Frontend	Backend	Database
HTML CSS TypeScript Angular Bootstrap	Java Open API Spring Boot	MySQL

3. Software Required

- Git Client Version 2.0 or above
- MySQL Workbench 8.0
- JDK 17
- MySQL 5.7.8
- Visual Studio Code version 1.60.x or later
- Chrome browser (latest version)
- GitHub account details with credentials that has Copilot license.
- Node JS version 18.10.0
- Angular version 16.2.x

Chapter – 3

GitHub Copilot Setup

4. Setting up GitHub Copilot

Requirement

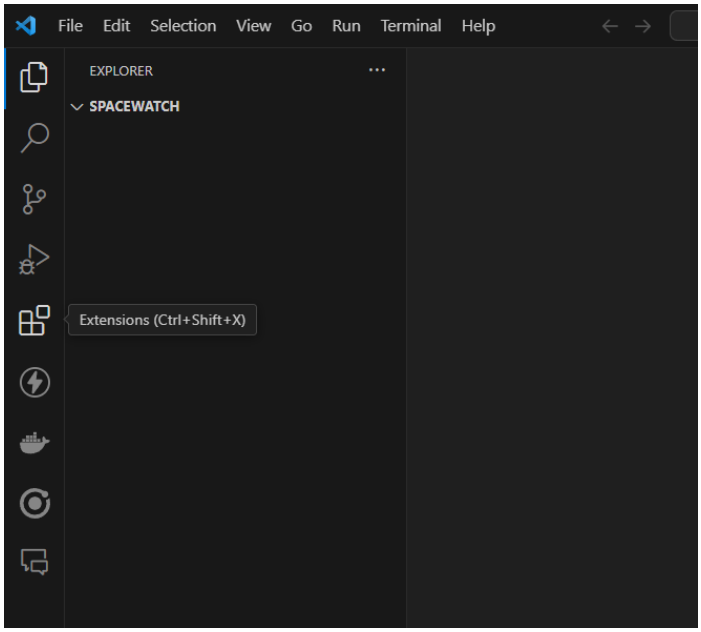
Install Copilot extension and signup into the appropriate GitHub account.

Pre-requisites

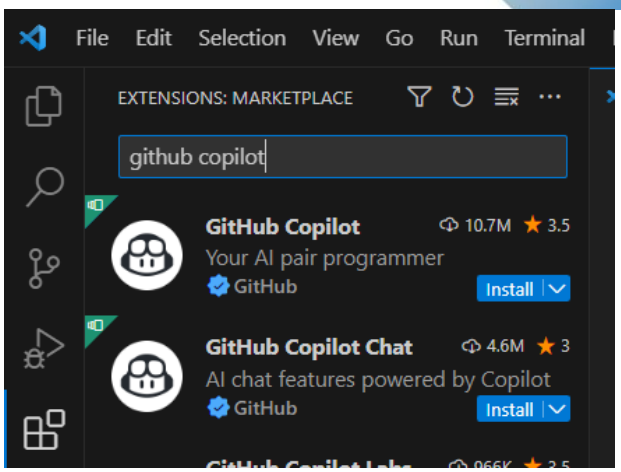
1. Visual Studio Code installed.
2. GitHub Credentials which have licensed access to Copilot Business

Activity 1: Install GitHub Copilot extension.

1. Open Visual Studio Code
2. Click on Extensions.

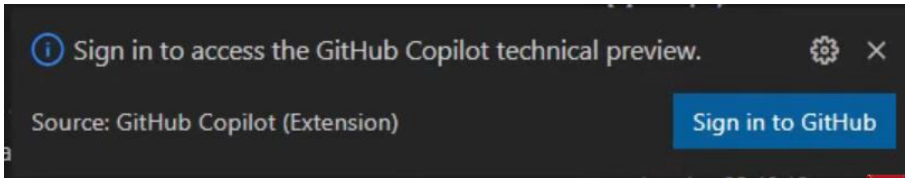


3. In the search box available in extensions section, type ‘github copilot’:



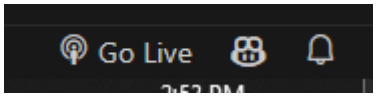
We need to install two extensions 'GitHub Copilot' and 'GitHub Copilot Chat'.

4. Click 'Install' for 'GitHub Copilot', this will install 'GitHub Copilot' and it will also install the 'GitHub Copilot Chat'.
5. After installation of the extension, a message like this will be displayed in the bottom right corner of VS Code.



Click on 'Sign in to GitHub', which will open the browser to authenticate with the GitHub account in the browser, follow the steps, which will establish the link between VS Code and the GitHub account.

6. In Visual Studio Code bottom right corner, if GitHub Copilot icon is displayed without any warning, it means that it is successfully installed.



Time Required to complete the Task

30 Minutes

Chapter – 4

Get Petclinic Code

5. Get Pet Clinic code from GitHub.

Requirement

In this section, we will get the Pet Clinic code from GitHub.

Activity 1: Get Pet Clinic REST (Backend)

1. Create a folder named 'petclinic' in a convenient location.
2. Open command prompt in 'petclinic' folder.
3. Run the below command to get the Pet Clinic backend code.

   ```
   git clone https://github.com/spring-petclinic/spring-petclinic-rest
   ```
4. Change to the project folder

   ```
   cd spring-petclinic-rest
   ```
5. Run the below command

   ```
   git checkout 65015cb02bf46fb93abc4c8a5f8587f464b44550
   ```
   This exercise was prepared based on the commit done on 26 Nov 2023. It is possible that in future there may be further commits to this project with new features. To ensure that we use the same version as what was used when preparing this exercise, the above git checkout command moves the code to the version committed on 26 Nov 2023.
6. Open `spring-petclinic-rest` folder in VS Code.

Activity 2: Get Pet Clinic Angular

1. In the command prompt run the below command to move back to 'petclinic' folder.

   ```
   cd ..
   ```
2. Run the below command to get the Pet Clinic frontend angular code.

   ```
   git clone https://github.com/spring-petclinic/spring-petclinic-angular
   ```
3. Change to the project folder

   ```
   cd spring-petclinic-angular
   ```
4. Run the below command to ensure that codebase aligns with the commit made on 11 Nov 2023 16:56, as this is the version use to implement this exercise.

   ```
   git checkout 0962c454618732fa1195692dd203fef2f9fab46f
   ```

5. Open `spring-petclinic-angular` folder in VS Code.

Time Required

30 Minutes

Chapter – 5

Setting up the Database

6. Setting up the database

Requirement

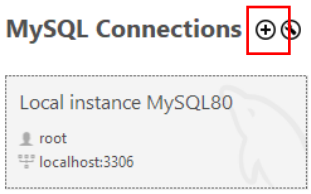
Create MySQL database for Pet Clinic application.

Pre-requisites

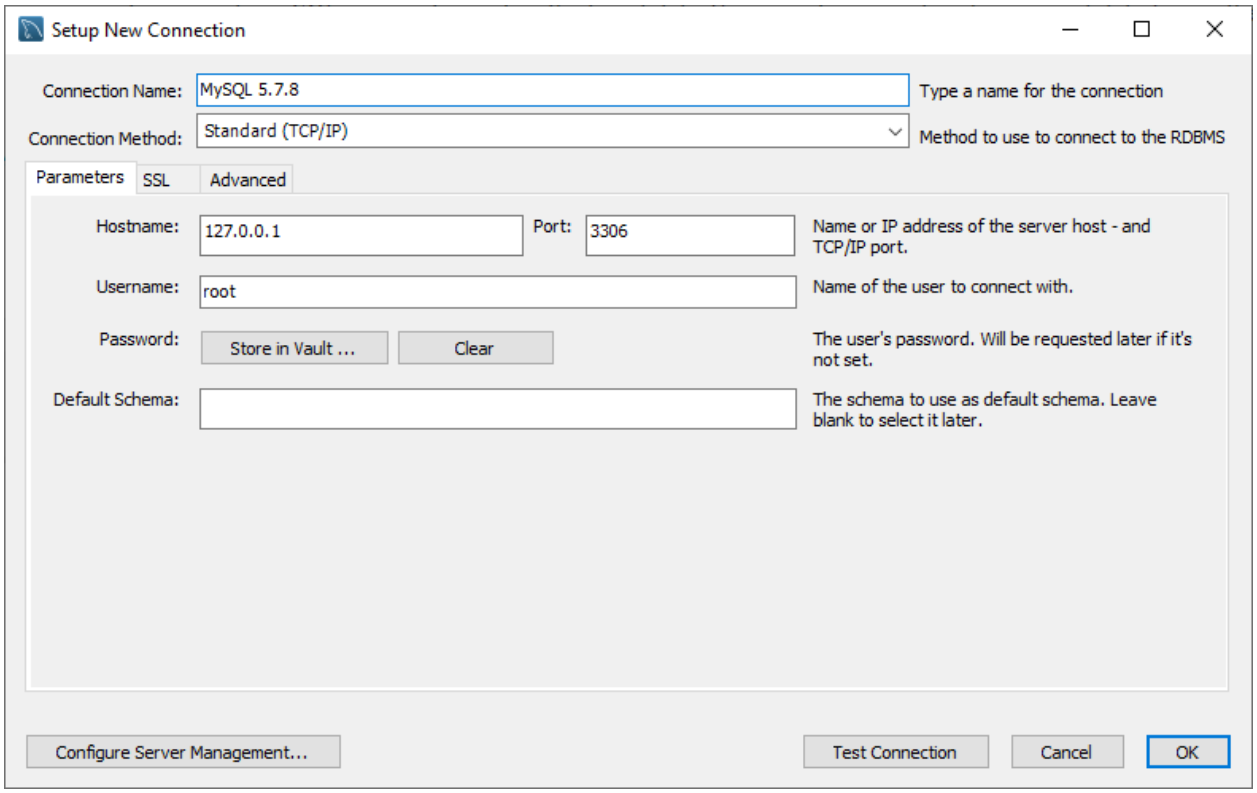
1. MySQL Community Server version 5.7.8
2. MySQL Workbench 8 installed.
3. MySQL Community Server running in the desktop.

Activity 1: Connect MySQL Workbench with MySQL Server

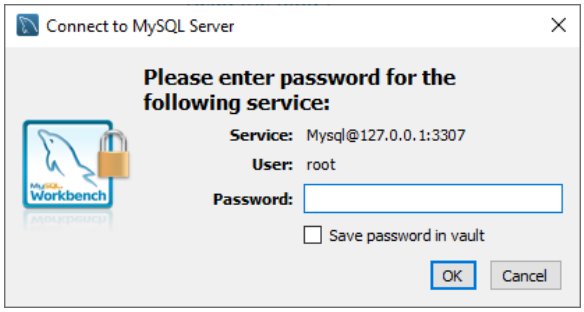
1. Open MySQL Workbench
2. Click on plus icon to add a new connection.



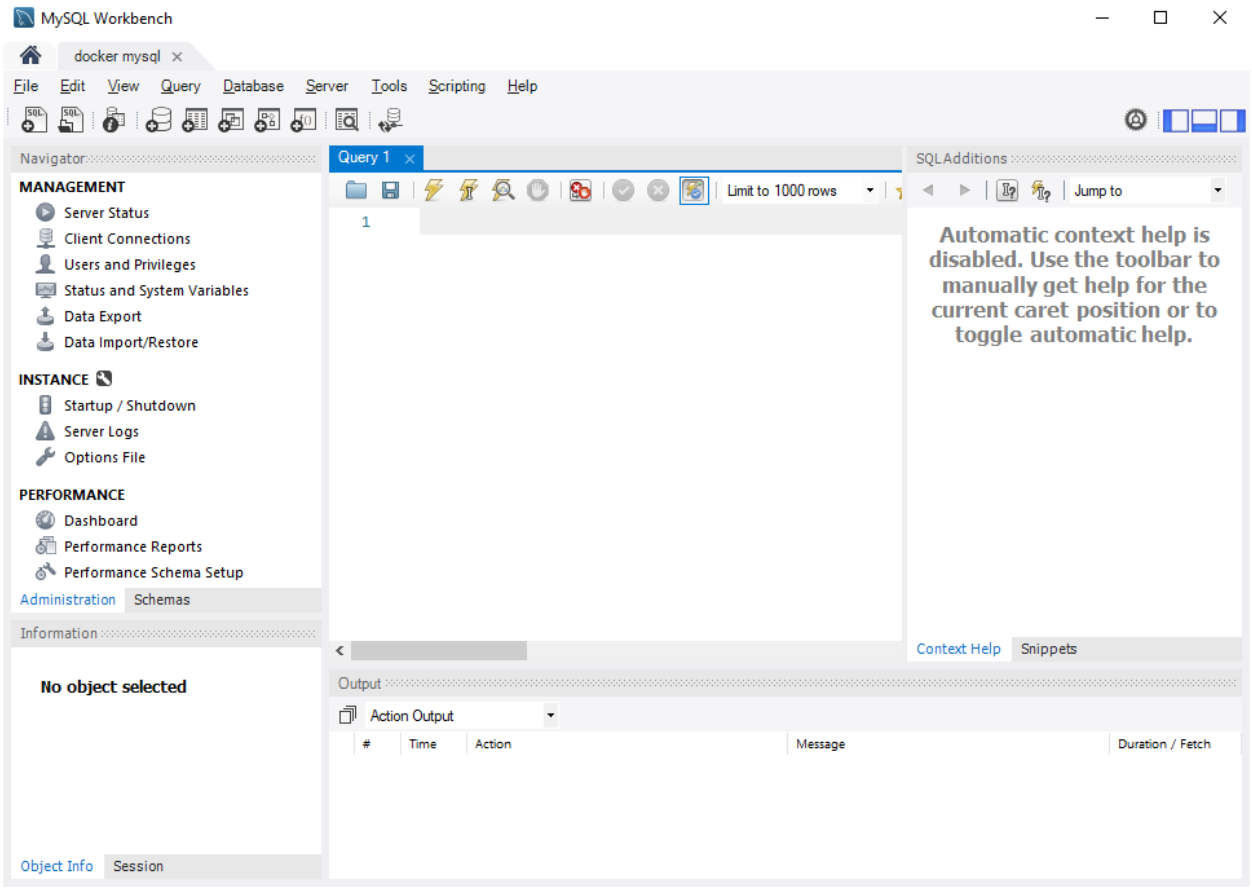
3. Enter Connection Name, Host Name, Port and Username as specified below:



4. Then click OK, which will add a new entry named 'MySQL 5.7.8' in the home screen.
5. Click on 'MySQL 5.7.8', which will ask for credentials.



6. Enter password the password and click 'OK', which will connect to the database and this is how the workbench looks like after login.

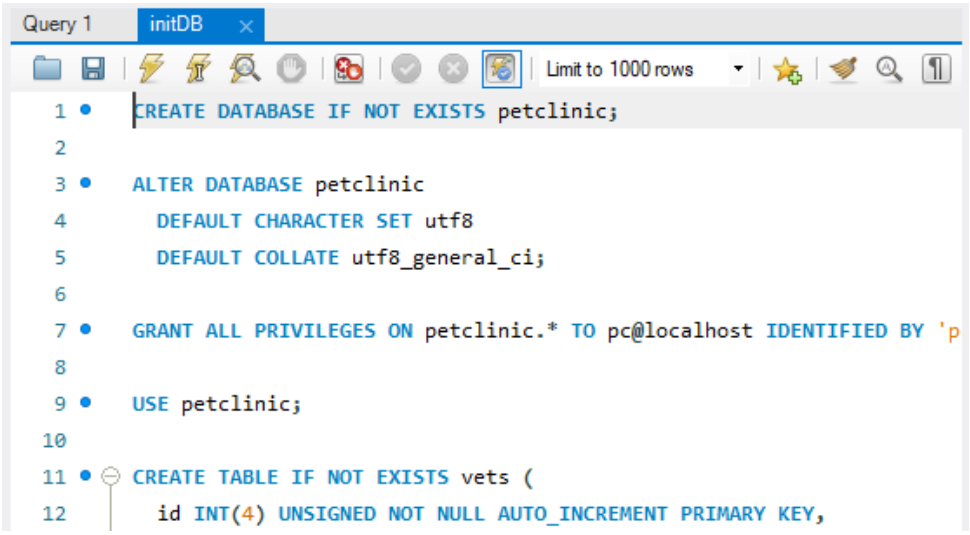


7. Under the 'Navigator' section select 'Schemas' tab.

Activity 2: Create petclinic schema

1. In MySQL Workbench, in 'File' menu select 'Open SQL Script…'.
2. Select the following SQL script file, which is available in petclinic rest project that was cloned from GitHub.

   `spring-petclinic-rest\src\main\resources\db\mysql\initDB.sql`
3. The above step will open the SQL file in an editor window.



4. In the editor window, click on the lightning icon, which will execute the script in the database server.

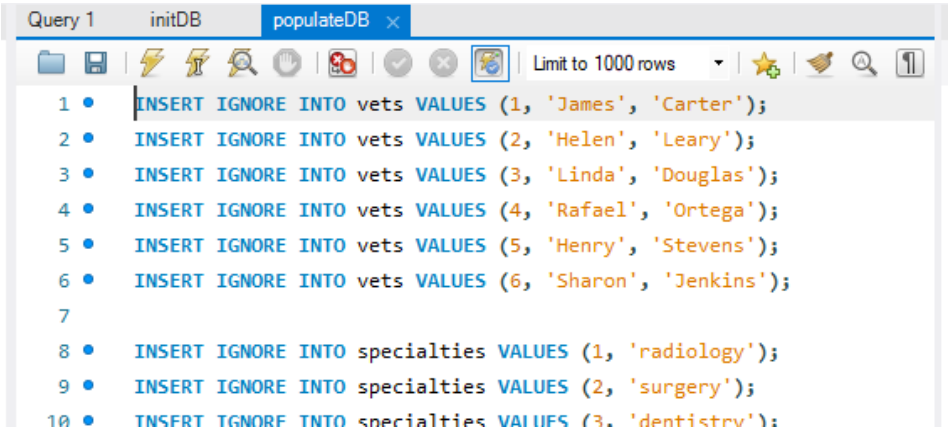
5. Check in Output section to see that there are no errors reported. If no errors are reported, then the database creation is successful.
6. The above step creates all the tables required for petclinic schema. In the next steps, we will verify if tables are created.
7. In the Navigator section click on the refresh button.

8. Then expand 'petclinic', then expand 'Tables' which should display the created tables.

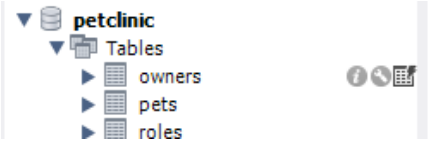
Activity 3: Populate petclinic schema with data

1. In MySQL Workbench, in 'File' menu select 'Open SQL Script…'.
2. Select the following SQL script file, which is available in petclinic rest project that was cloned from GitHub.

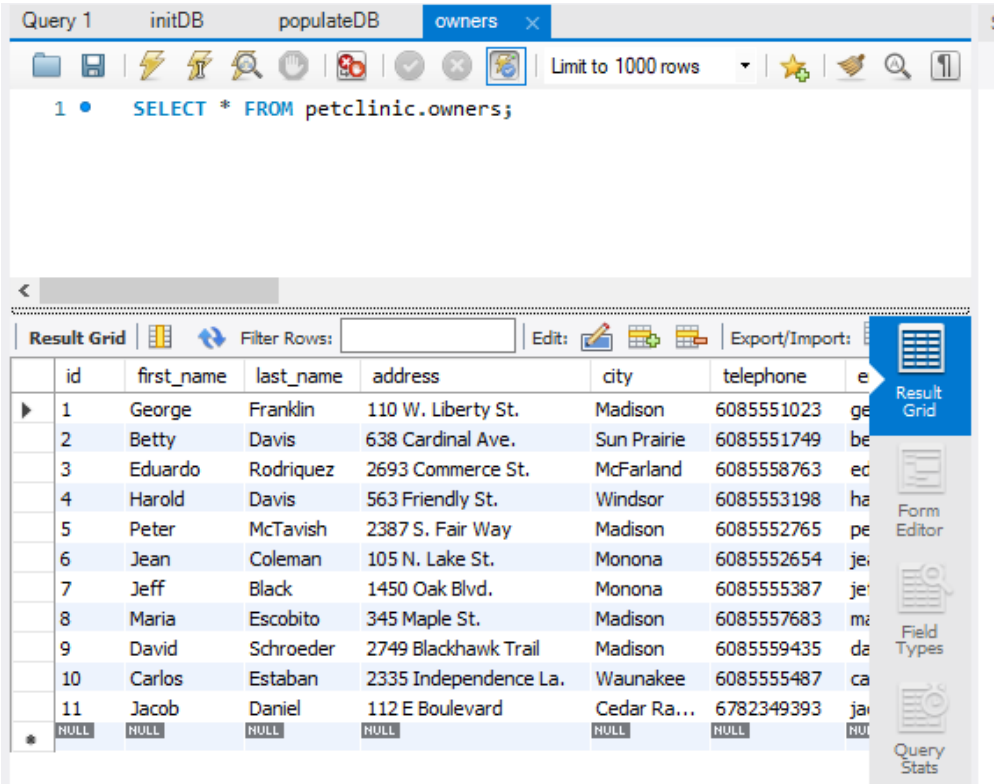
   `spring-petclinic-rest\src\main\resources\db\mysql\ populateDB.sql`
3. This will open populateDB.sql in another editor window.



4. Before running the script ensure that 'petclinic' is the current database selected. If the 'petclinic' schema listed in the navigator is bold, then it means that 'petclinic' schema is selected. If it is not selected, then double click on 'petclinic' to make it selected.
5. Using lightning icon, run the script, which will populate all the tables with appropriate data.
6. Check in bottom Output section to see if there are no errors reported.
7. Follow steps below to verify if tables are populated.
8. Hover the mouse over any one of the tables in the 'Navigator' section and click on the grid option.



9. The above action will display the table data in the right-side section.



id	first_name	last_name	address	city	telephone
1	George	Franklin	110 W. Liberty St.	Madison	6085551023
2	Betty	Davis	638 Cardinal Ave.	Sun Prairie	6085551749
3	Eduardo	Rodriquez	2693 Commerce St.	McFarland	6085558763
4	Harold	Davis	563 Friendly St.	Windsor	6085553198
5	Peter	McTavish	2387 S. Fair Way	Madison	6085552765
6	Jean	Coleman	105 N. Lake St.	Monona	6085552654
7	Jeff	Black	1450 Oak Blvd.	Monona	6085555387
8	Maria	Escobito	345 Maple St.	Madison	6085557683
9	David	Schroeder	2749 Blackhawk Trail	Madison	6085559435
10	Carlos	Estaban	2335 Independence La.	Waunakee	6085555487
11	Jacob	Daniel	112 E Boulevard	Cedar Ra...	6782349393

10. In a similar fashion, check if all tables are populated with data.
11. With this, the steps to prepare the database is completed.

Time Required

1 Hour

Chapter – 6

Launch Petclinic API

7. Start the Spring Boot Pet Clinic API

Requirement

Now that we have created the database, we can start working on the Backend REST API, which will be implemented using Spring Boot with IDE as Visual Studio Code. The first step is to run the Spring Boot Pet Clinic REST app and check if the REST API endpoints work as expected.

GitHub Copilot will be used wherever necessary to accelerate the development of REST API.

Pre-requisites

1. JDK 17 installed
2. Visual Studio Code 1.60.x or later installed
3. 'GitHub Copilot' extension installed in Visual Studio Code

Activity 1: Verify Java Environment variables

1. Check if the 'bin' folder of JDK 17 installation is in 'Path' variable.
2. Check if 'JAVA_HOME' environment variable points to the JDK 17 installed folder.

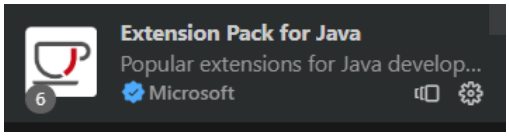
Activity 2: Open project in Visual Studio Code (vs code)

3. In window file explorer, go to 'petclinic' folder where we have cloned the project from GitHub.
4. Right click on 'spring-petclinic-rest' folder and select 'Open with Code', which will open the project in vs code.

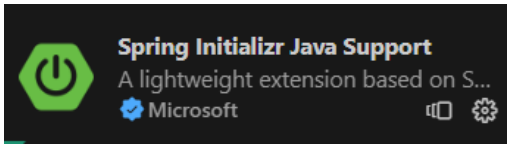
Activity 3: Install VS Code Extension required for Java development

1. In vs code, click on Extension icon available in the left side toolbar.

2. Search for 'extension pack for java', then click 'Install' to install the extension. This extension helps to build and run Java class files within Visual Studio Code.



3. Search for 'spring initializr java support' extension, then click 'Install'. This extension helps to perform Spring Boot related operations.



Activity 4: Configuring and running the Pet Clinic Spring Boot app

1. In vs code, open src\main\resources**application.properties**
2. Change profiles setting to use MySQL database.

```
spring.profiles.active=mysql,spring-data-jpa
```

3. Open src\main\resources**application-mysql.properties** and change the below specified configuration.
4. Configure database connection URL, username and password

```
spring.datasource.url = jdbc:mysql://localhost:3307/petclinic?useUnicode=true
spring.datasource.username=root
spring.datasource.password=petclinic
spring.datasource.driver-class-name=com.mysql.jdbc.Driver
```

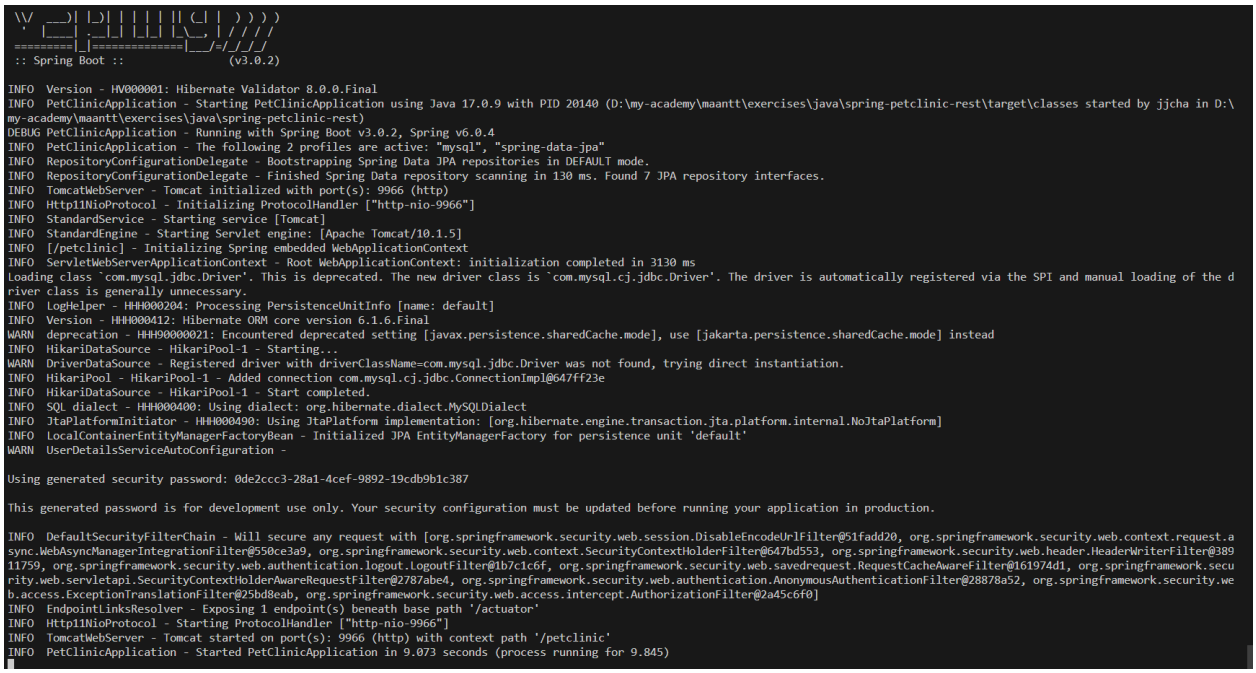
   Provide database credentials accordingly to your MySQL server installation.
5. In windows file explorer, open petclinic/spring-petclinic-rest folder.
6. Click within the windows file explorer address bar, type 'cmd' and press enter key, which will open command prompt in 'spring-petclinic-rest' folder.
7. Run the below command in the command prompt to run the app.

   `mvnw spring-boot:run`

   IMPORTANT NOTE: When preparing this exercise, the build and execution of the app was not stable, sometime just re-running the above command works. Sometimes executing a build with `mvnw clean package` command and then running the command works. One of the sample errors shown when preparing this exercise is provided below for reference.

```
***************************
APPLICATION FAILED TO START
***************************

Description:

Parameter 0 of constructor in org.springframework.samples.petclinic.service.ClinicServiceImpl required a bean of type 'org.springframework.samples.petclinic.repository.PetRe
pository' that could not be found.


Action:

Consider defining a bean of type 'org.springframework.samples.petclinic.repository.PetRepository' in your configuration.

[INFO] ------------------------------------------------------------------------
[INFO] BUILD FAILURE
[INFO] ------------------------------------------------------------------------
[INFO] Total time:  41.029 s
[INFO] Finished at: 2023-12-10T14:47:42+05:30
[INFO] ------------------------------------------------------------------------
[ERROR] Failed to execute goal org.springframework.boot:spring-boot-maven-plugin:3.0.2:run (default-cli) on project spring-petclinic-rest: Process terminated with exit code:
 1 -> [Help 1]
[ERROR]
[ERROR] To see the full stack trace of the errors, re-run Maven with the -e switch.
[ERROR] Re-run Maven using the -X switch to enable full debug logging.
[ERROR]
[ERROR] For more information about the errors and possible solutions, please read the following articles:
[ERROR] [Help 1] http://cwiki.apache.org/confluence/display/MAVEN/MojoExecutionException
```

8. This is how the log looks when the app is successfully started.



9. To verify if the REST API endpoints of Pet Clinic are working, try the below REST API Endpoint URL in the browser.

   http://localhost:9966/petclinic/api/owners

   It should display the JSON data containing owner data from owner table of petclinic database.

Time Required

1 Hour

Chapter – 7

Launch Petclinic Web App

8. Starting the Pet Clinic Angular app

Requirement

In the previous section, we started the backend REST API for Pet Clinic. In this section we will start the angular application and verify if it works with the database.

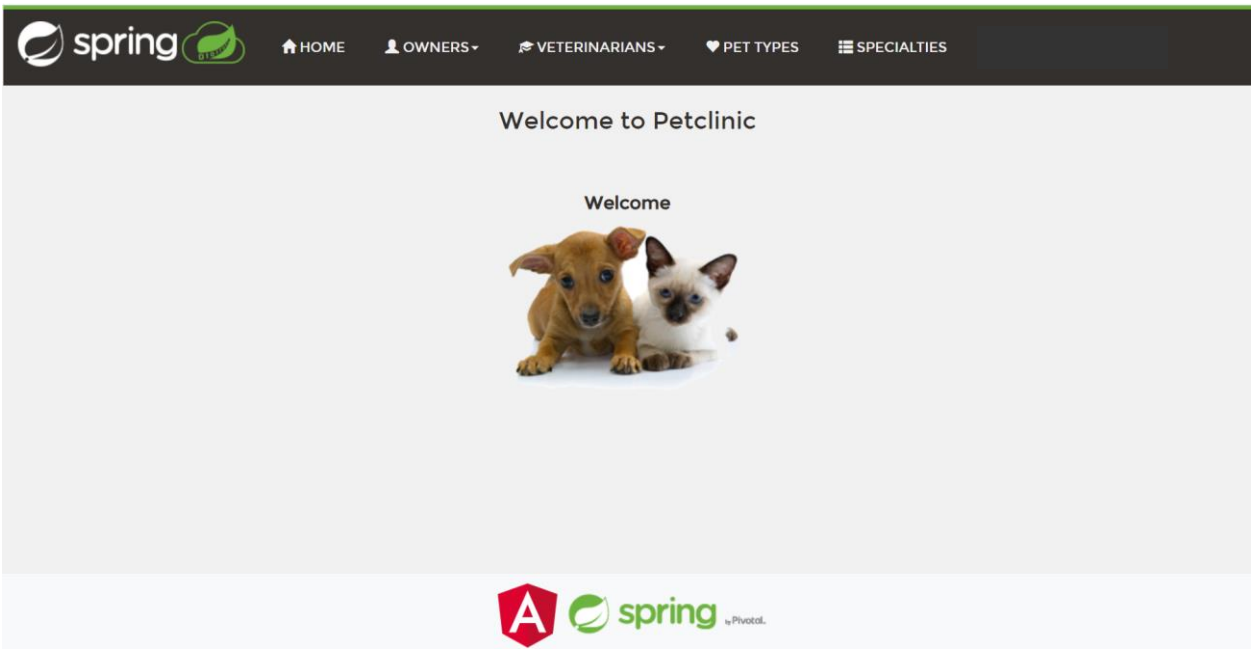
Activity 1 – Starting the Angular App

1. Open 'spring-petclinic-angular' folder in a separate vs code window.
2. Open Terminal in VS Code. [File > View > Terminal]
3. Run the below command in terminal to install dependencies

   ```
   npm install
   ```
4. Then run the below command to start the angular app:

   ```
   ng serve
   ```
5. Open the below URL in the browser.

   ```
   http://localhost:4200
   ```
6. This is how the app should look like.



7. Navigate through all the navigation links available on the top and check if all the features specified in Section 2 About this Exercise of this document is working.

Time Required

30 Minutes

Chapter -8

Include Email for Owner

9. Including email for Owner in backend

Requirement

Now that we have pet clinic app running end to end, we can start implementing the additional features one by one. Currently as owner contact, telephone number is captured for each owner. The pet clinic has planned to improve the communication with owners in the future. Hence, we must additionally capture the email address of the owner. In this section, we will see how to get this implemented.

Activity 1 – Adding email column in owners table

1. In vs code, click on this icon in the left side toolbar, which will open the copilot chat window.

2. Type the following prompt to generate the alter table script.

```
owner table needs to be modified to add email
column, generate the sql script for this requirement
```

3. This will be the script generated by copilot.

```
ALTER TABLE owners
ADD COLUMN email VARCHAR(255);
```

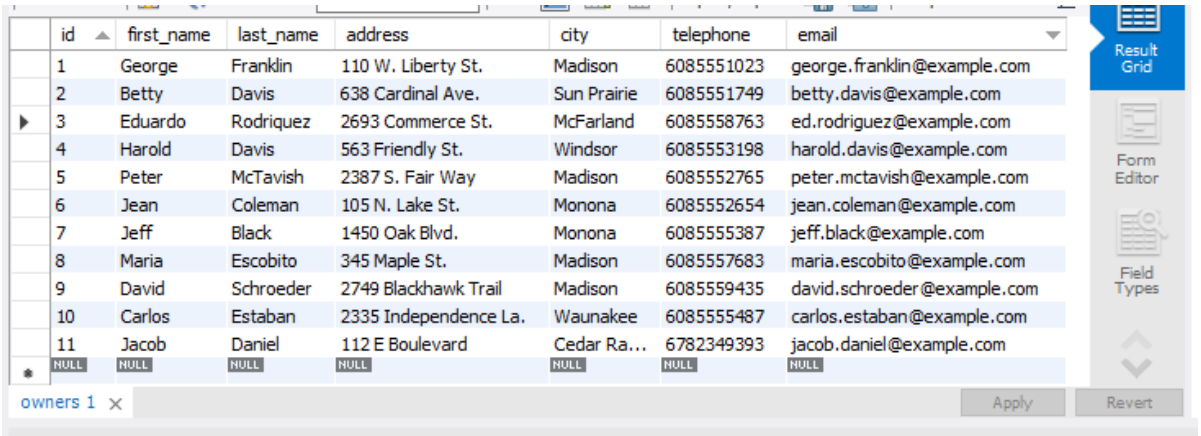
4. Copy the generated code using the copy icon.
5. Go to MySQL workbench.
6. Click on this icon below File menu to create a new SQL Script window.

7. Paste the alter table script within the editor window and click on the lightning icon to run the script, which will alter the database table.

8. Type the following prompt to generate update statement

```
Following are the list of Owners id and name, column names are 'id','first_name' which are already available in the 'Owners' table. column 'id' is the primary key. Generate sql query which will update the 'email' column with sample email id for each of these owners.
1 George
2 Betty
3 Eduardo
4 Harold
5 Peter
6 Jean
7 Jeff
8 Maria
9 David
10 Carlos
```

9. Now open a new SQL Script window, Paste the update script within the editor window and click on the lightning icon to run the script, which will update the email. This is how the table looks after entering data for email.



id	first_name	last_name	address	city	telephone	email
1	George	Franklin	110 W. Liberty St.	Madison	6085551023	george.franklin@example.com
2	Betty	Davis	638 Cardinal Ave.	Sun Prairie	6085551749	betty.davis@example.com
3	Eduardo	Rodriquez	2693 Commerce St.	McFarland	6085558763	ed.rodriguez@example.com
4	Harold	Davis	563 Friendly St.	Windsor	6085553198	harold.davis@example.com
5	Peter	McTavish	2387 S. Fair Way	Madison	6085552765	peter.mctavish@example.com
6	Jean	Coleman	105 N. Lake St.	Monona	6085552654	jean.coleman@example.com
7	Jeff	Black	1450 Oak Blvd.	Monona	6085555387	jeff.black@example.com
8	Maria	Escobito	345 Maple St.	Madison	6085557683	maria.escobito@example.com
9	David	Schroeder	2749 Blackhawk Trail	Madison	6085559435	david.schroeder@example.com
10	Carlos	Estaban	2335 Independence La.	Waunakee	6085555487	carlos.estaban@example.com
11	Jacob	Daniel	112 E Boulevard	Cedar Ra...	6782349393	jacob.daniel@example.com
NULL	NULL	NULL	NULL	NULL	NULL	NULL

10. Click on Apply button and follow the steps which will update the data in email column of owners table.

> **Copilot Feature:** From the above activity, we can understand that chat is one of the ways to generate code using copilot.

Understanding Spring Pet Clinic REST project structure

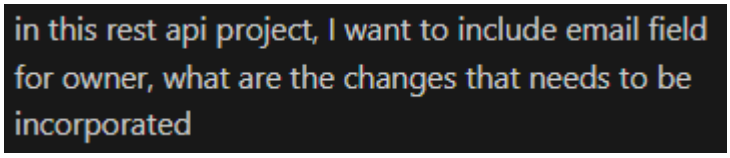
It is important to understand how the existing project code is structured and the additional technologies used. Based on the points provided below, go through the

source code references provided below to get yourself familiarized how the existing code is structured.

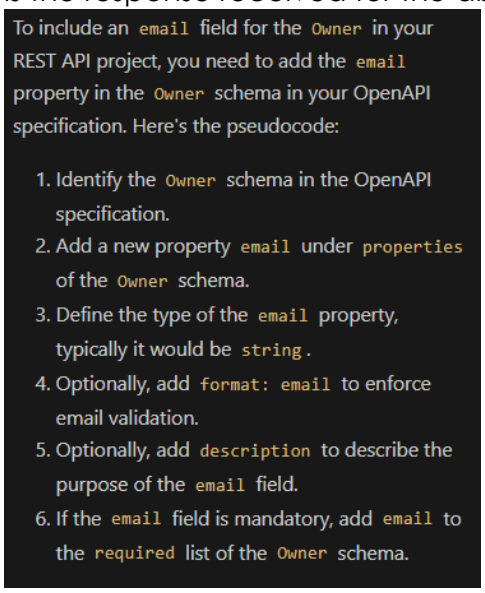
1. The controller classes are integrated with code generated by Open API.
2. Open API generates additional classes based on the configuration provided in `src\main\resources\openapi.yml`
3. Go through the code in following folders to understand if you are not familiar with Open API.
    a. `target\generated-sources\openapi\src\api`
    b. `target\generated-sources\openapi\src\dto`
    c. `src\main\java\org\springframework\samples\petclinic\rest\controller`
4. This project also uses Mapper to convert data between DTO and Entity. Refer code in `src\main\java\org\springframework\samples\petclinic\mapper`.

Activity 2 – Adding email property to entity class

5. Go to copilot chat and enter the below prompt, to know at high level what changes needs to be implemented for include including email field for owner.



The above prompt provides a detailed explanation of areas to be modified. This is the response received for the above prompt when working on this exercise.



We can see that copilot is smart enough to identify that Open API is used in this project and it suggests appropriate changes.

6. This is the next prompt provided to check if changes are required in entity class.

thank you. is there any change required in model class to include email?

7. For the above prompt, below is the code generated.

```java
import javax.validation.constraints.Email;
import javax.validation.constraints.NotEmpty;

public class Owner {
    private String username;
    @Email
    @NotEmpty
    private String email;
    private List<Role> roles;

    // existing getters and setters

    public String getEmail() {
        return email;
    }

    public void setEmail(String email) {
        this.email = email;
    }
}
```

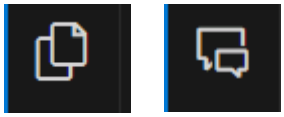
8. It has not generated the relevant Column annotations, hence provided the below prompt.

generate code to include email property for owner, include column annotation with column name as email with email validation

9. This is the code generated for the above prompt.

```java
import javax.validation.constraints.Email;
import javax.persistence.Column;

// existing code...

@Column(name = "email")
@Email
@NotEmpty
private String email;

// existing code...

public String getEmail() {
    return this.email;
}

public void setEmail(String email) {
    this.email = email;
}
```

10. The code can be copied and pasted to the appropriate location within `src\main\java\org\springframework\samples\petclinic\model\Owner.java`
11. Save Owner.java
12. In vs code, to switch between file explorer and copilot chat use the below two icons.



13. In Owner.java go to toString() function and append email. Notice that copilot generates inline suggestions as greyed out text.

```
.append("telephone", this.telephone)
.append("email", this.email)
.toString();
```

14. Just press TAB key to accept the suggestion provided.

> **Copilot Feature:** Copilot provides inline suggestions as we type code. To accept the suggestion, we can press the tab key, to ignore the suggestion press Esc key.

15. This is how the toString() function should look like after the changes.

```
@Override
public String toString() {
    return new ToStringCreator(this)

        .append("id", this.getId())
        .append("new", this.isNew())
        .append("lastName", this.getLastName())
        .append("firstName", this.getFirstName())
        .append("address", this.address)
        .append("city", this.city)
        .append("telephone", this.telephone)
        .append("email", this.email)
        .toString();
}
```

Activity 3 – Adding email property to Open API

1. The following aspects are configured in Open API.
    a. Endpoint definition
    b. Definition of request / response data model for each endpoint
    c. Definition of error response for each endpoint
    d. Definition of model for DTO classes
2. The Open API configuration is defined in openapi.yml.
3. In Open API, we need to configure the inclusion of email in owner model.
4. We can now start making changes on Open API. Following is the prompt provided. Minor correction in the below prompt, start with "in this" instead of "this".

```
this project controller is integrated with open api. from this
perspective what changes are required to include email property in
owner. the open api configuration in available in openapi.yml file of
resources folder
```

5. This is the sample change suggested by copilot in openapi.yml.

```
 telephone:
   type: string
 email:
   type: string
   format: email
   description: Owner's email address
```

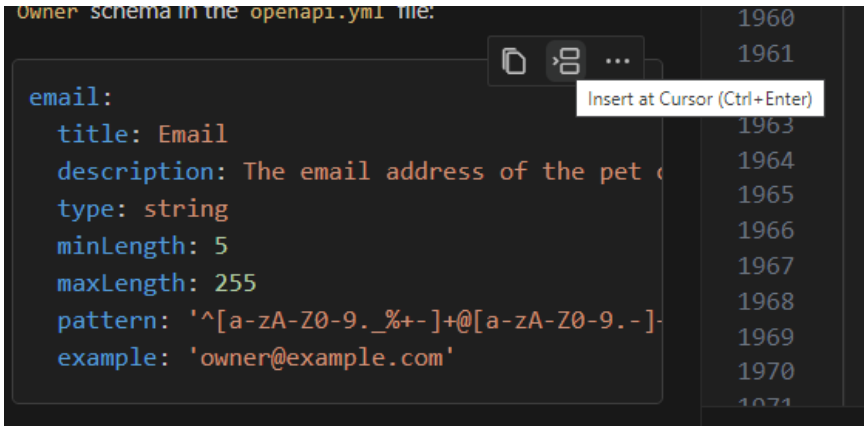
6. This response is missing additional Open API properties that needs to be provided. Hence provide the below prompt.

```
generate title, description, email regex pattern and example for email
entry in openapi.yml
```

7. Notice that it has additionally included the requested fields and additionally it has included minLength and maxLength with appropriate values.

```
email:
  title: Email
  description: The email address of the pet owner.
  type: string
  minLength: 5
  maxLength: 255
  pattern: '^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$'
  example: 'owner@example.com'
```

8. Now the above settings need to be added to openapi.yml.
9. Open src\main\resources\openapi.yml
10. Within the openapi.yml navigate to components/schemas/OwnerFields/telephone/example
11. Include a new line after 'example' and align the indent straight to telephone.
12. Then use the 'Insert at Cursor' option available in copilot chat window to get the code included in the position where the cursor is present in openapi.yml file.



13. Then check if the included code is indented properly.
14. Then include email in the required section below.

```
  email:
    title: Email
    description: Th
    type: string
    minLength: 5
    maxLength: 255
    pattern: '^[a-z
    example: 'owner
required:
  - firstName
  - lastName
  - address
  - city
  - telephone
  - email
```

15. After making above changes, we can see the dto classes generated by Open API having email field. Check it out in the file below.

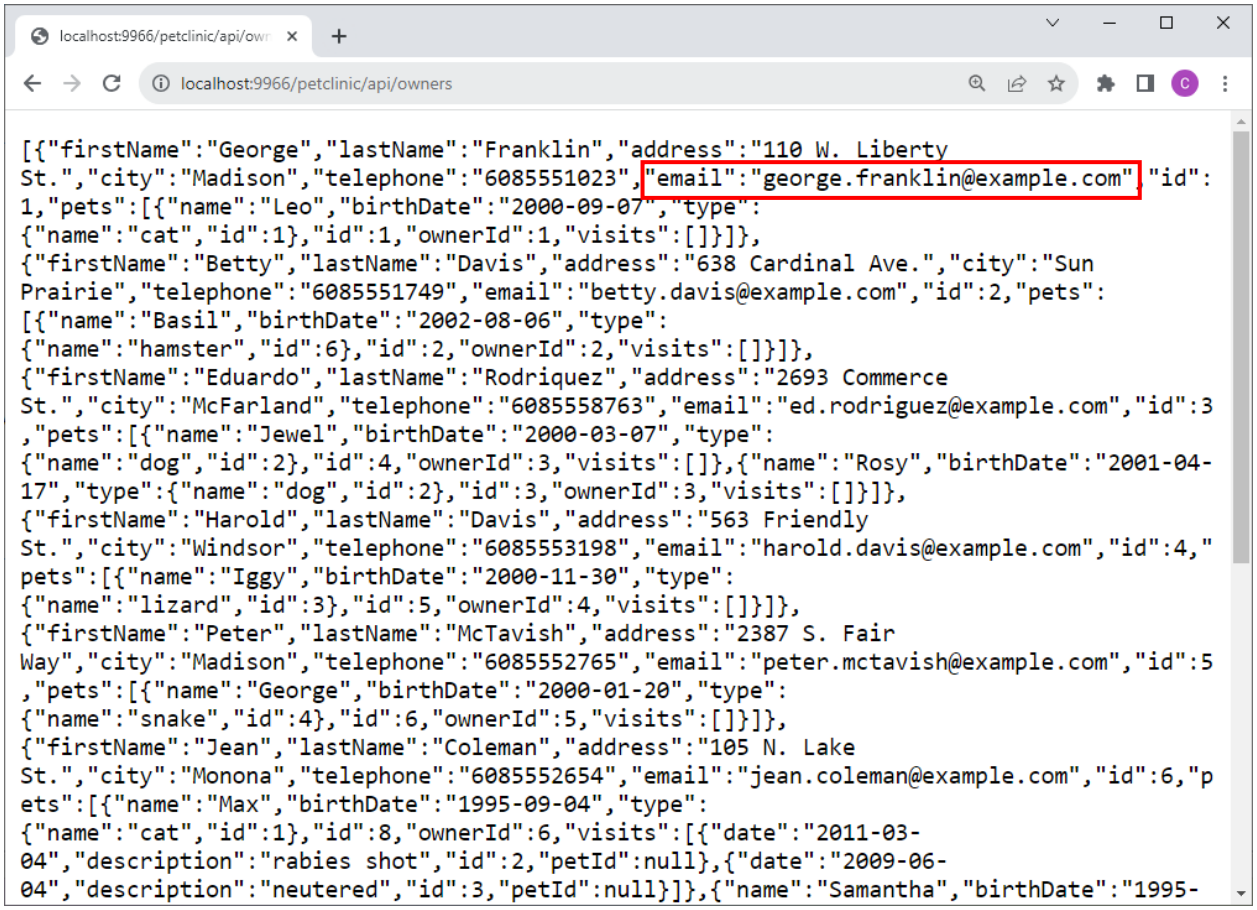
    `target\generated-sources\openapi\src\main\java\org\springframework\samples\petclinic\rest\dto\OwnerDto.java`

16. Now all necessary changes in the backend are completed to handle email. Let us now take the steps to verify if it is working

IMPORTANT NOTE: The unit test cases already implemented in pet clinic rest project was not optimized, hence even minor changes had made many test cases to fail. For example, changing the database from hsqldb to mysql results in test case failure. Hence unit testing of backend is not covered as part of this exercise.

17. Spring Boot Dev Tools is not enabled in this project due to Open API, hence we need to manually stop and start the server.
18. Go to the command prompt where the spring boot app is running.
19. Press Ctrl+C to stop the server.
20. Now start the server again with 'mvnw spring-boot:run' command. As mentioned earlier retry the command if it fails. Also try running 'mvnw clean package' once and then try running 'mvnw spring-boot:run'.
21. Once the server is started, try the below URL in the browser. This URL gets the list of all the owners in the database, we can see the email field included for each owner in the below screenshot.

    http://localhost:9966/petclinic/api/owners



Time Required

1 Hour

10. Including email for Owner in frontend

Requirement

In this section, we will make changes in the angular application to view and capture email for owner.

Activity 1: Display email when listing owners

1. Open the vs code window where the pet clinic angular project is opened.
2. Open src\app\owners\owner.ts
3. Include email property with type as string and save the file.

```
telephone: string;
email: string;
```

4. Open src\app\owners\owner-list\owner-list.component.html
5. Include Email column in the table heading

```
<th>Email</th>
```

6. Go to the body section of the table where telephone <td> tag is present.
7. Place the cursor after the closing <td> tag.
8. When enter key is pressed, we can see copilot showing suggestion.

```
<td>{{ owner.telephone }}</td>
<td>{{ owner.email }}</td>
```

NOTE: Copilot is able to show the above suggestion based on the Email heading included in the table.

9. Press TAB key to accept the suggestion. We can see how copilot helps reduce the amount code we type.
10. Save owner-list.component.html and wait till the app is reloaded in the browser
11. Click OWNERS, then click SEARCH, which will display the list of owners. We can see Email column added and email of each owner displayed.

Name	Address	City	Telephone	Email	Pets
George Franklin	110 W. Liberty St.	Madison	6085551023	george.franklin@example.com	Leo
Betty Davis	638 Cardinal Ave.	Sun Prairie	6085551749	betty.davis@example.com	Basil
Eduardo Rodriquez	2693 Commerce St.	McFarland	6085558763	ed.rodriguez@example.com	Jewel Rosy
Harold Davis	563 Friendly St.	Windsor	6085553198	harold.davis@example.com	Iggy
Peter McTavish	2387 S. Fair Way	Madison	6085552765	peter.mctavish@example.com	George
Jean Coleman	105 N. Lake St.	Monona	6085552654	jean.coleman@example.com	Max Samantha
Jeff Black	1450 Oak Blvd.	Monona	6085555387	jeff.black@example.com	Lucky
Maria Escobito	345 Maple St.	Madison	6085557683	maria.escobito@example.com	Mulligan
David Schroeder	2749 Blackhawk Trail	Madison	6085559435	david.schroeder@example.com	Freddy
Carlos Estaban	2335 Independence La.	Waunakee	6085555487	carlos.estaban@example.com	Lucky Sly

Activity 2: Display email in owner detail screen

1. Open src\app\owners\owner-detail\owner-detail.component.html
2. Press enter key after telephone </tr> tag, we can see copilot suggestion.

```
<tr>
  <th>Telephone</th>
  <td>{{ owner.telephone }}</td>
</tr>
<tr>
  <th>Email</th>
  <td>{{ owner.email }}</td>
```

   NOTE: Copilot remembers that we are working on including email field and provides suggestions accordingly.
3. Press TAB to accept suggestion and then Include closing </tr> tag and save the file.
4. In browser, in owner listing page, click on an owner name, in the owner details displayed, we can see the owner email displayed.

Owner Information

Name	George Franklin
Address	110 W. Liberty St.
City	Madison
Telephone	6085551023
Email	george.franklin@example.com

Activity 3: Capture email when editing an owner

1. Open src\app\owners\owner-edit\owner-edit.component.html
2. Type the following prompt to generate code for including email field.

```
generate code to include capture email field in owner-edit.component.html with validation for mandatory field and email pattern with regular expression
```

3. This is the response code generated by copilot.

```
<div class="form-group">
  <label class="col-sm-2 control-label" for="
  <div class="col-sm-10">
    <input
      type="email"
      id="email"
      class="form-control"
      formControlName="email"
      [ngClass]="{ 'is-invalid': email.dirty
    />
    <span
      class="help-block"
      *ngIf="email.dirty && email.hasError('r
    >
      Email is required
    </span>
    <span
      class="help-block"
      *ngIf="email.dirty && email.hasError('p
    >
      Invalid email format
    </span>
  </div>
</div>
```

```
this.ownerForm = this.fb.group({
  // other form controls...
  email: [
    '',
    [
      Validators.required,
      Validators.pattern(/^[\w-]+(\.[\w-]+)*
    ]
  ]
});
```

4. Find below the screenshot of the existing code implemented for telephone field.

```
<div
  class="form-group has-feedback"
  [class.has-success]="telephone.valid"
  [class.has-error]="!telephone.valid"
>
  <label for="telephone" class="col-sm-2 control-label">Telephone</label>
  <div class="col-sm-10">
    <input
      type="text"
      class="form-control"
      id="telephone"
      [(ngModel)]="owner.telephone"
      name="telephone"
      required
      maxlength="10"
      pattern="^[0-9]{0,10}$"
      #telephone="ngModel"
    />
    <span
      class="glyphicon form-control-feedback"
      [class.glyphicon-ok]="telephone.valid"
      [class.glyphicon-remove]="!telephone.valid"
      aria-hidden="true"
    ></span>
    <span
      class="help-block"
      *ngIf="telephone.dirty && telephone.hasError('required')"
      >Phone number is required</span
    >
    <span
      class="help-block"
      *ngIf="telephone.dirty && telephone.hasError('maxlength')"
      >Phone number cannot be more than 10 digits long</span
    >
    <span
      class="help-block"
      *ngIf="telephone.dirty && telephone.hasError('pattern')"
      >Phone number only accept digits</span
    >
  </div>
```

We can see the following deviations in the code generated by copilot:

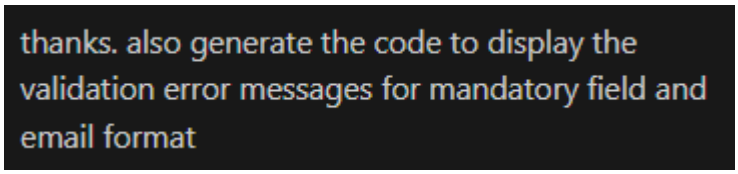
a. The existing code is implemented using Template Driven Form, whereas the code generated by copilot uses Reactive Form.

b. Style for input field with color coding is missing.

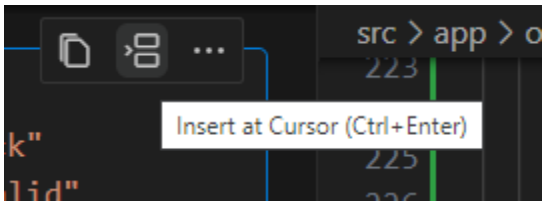
5. Hence, we provide the below prompt and requesting copilot to use the existing telephone code reference.

```
for html code use telephone field code in owner-edit.component.html as reference and freshly generate the code
```

6. Now the code is generated in alignment with what is required, we can see it is partially generated, the validation message related code is not included. Hence, we request to copilot to include validation messages.

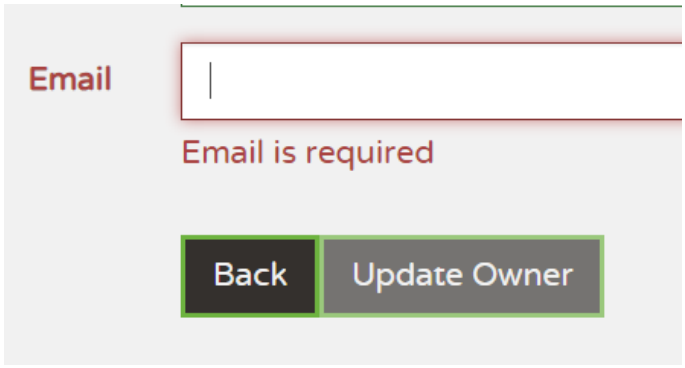


7. Now we can see the complete code generated for email.
8. In owner-edit.component.html, place the cursor after the closing telephone div tag and press enter key.
9. Then click 'Insert at Cursor' displayed above the code generated within copilot chat, which will place the code in cursor position.

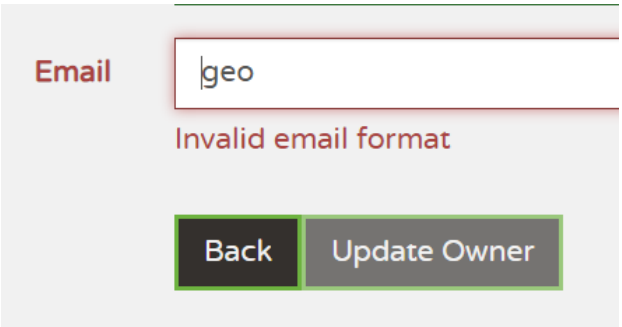


10. We can save the file.
11. Wait till app reloads in the browser
12. Click OWNER > SEARCH, then click on an owner
13. Then click 'Edit Owner' button for any one of the owners.
14. The Edit Owner page is displayed
15. Check if the email field is prepopulated with respective email.

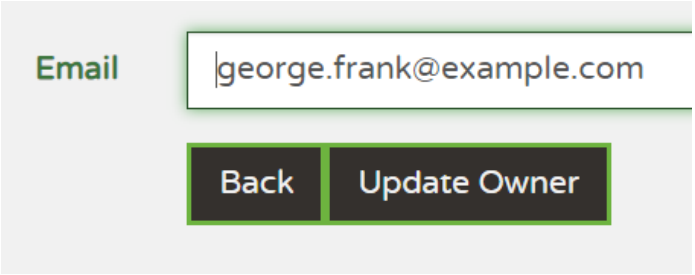
16. Clear the email populated. Now validation error message should be displayed. The border to the input field should change to red and the 'Update Owner' button should get disabled, preventing the user from submitting the form.



17. As we type the email, the invalid email format validation message should be displayed till the email is entered correctly. Notice again the input field is bordered with red color and Update Owner button is disabled.



18. Once the entered email format is correct, we can see a green border around the field, no error message displayed and Update Owner button is enabled.



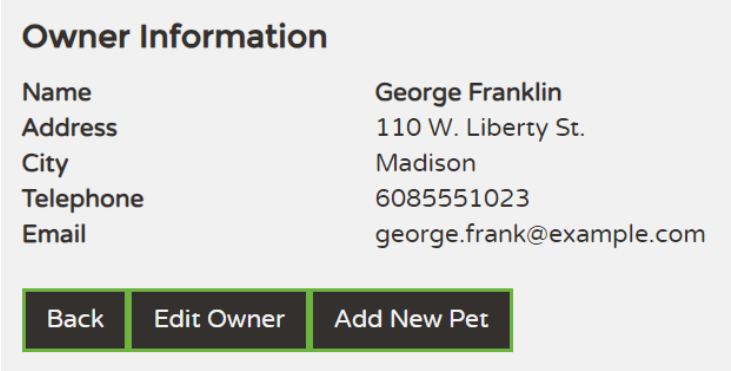
19. Now we can click on 'Update Owner' to save the changes.
20. After clicking 'Update Owner', it may not work as expected and you may notice any one of the below specified behavior.
    a. Nothing happens after clicking 'Update Owner' button.
    b. The email field for George is stores as null in the database
21. After checking the logs and code it can be found that updateOwner() function in controller\OwnerRestController.java, does not transfer email data from DTO to Entity.

```java
@PreAuthorize("hasRole(@roles.OWNER_ADMIN)")
@Override
public ResponseEntity<OwnerDto> updateOwner(Integer ownerId, OwnerFieldsDto ownerFieldsDto) {
    Owner currentOwner = this.clinicService.findOwnerById(ownerId);
    if (currentOwner == null) {
        return new ResponseEntity<>(HttpStatus.NOT_FOUND);
    }
    currentOwner.setAddress(ownerFieldsDto.getAddress());
    currentOwner.setCity(ownerFieldsDto.getCity());
    currentOwner.setFirstName(ownerFieldsDto.getFirstName());
    currentOwner.setLastName(ownerFieldsDto.getLastName());
    currentOwner.setTelephone(ownerFieldsDto.getTelephone());
    this.clinicService.saveOwner(currentOwner);
    return new ResponseEntity<>(ownerMapper.toOwnerDto(currentOwner), HttpStatus.NO_CONTENT);
}
```

22. The code to transfer email can be included; we can see copilot coming to help as suggestion as we type the code.

```java
currentOwner.setTelephone(ownerFieldsDto.getTelephone());
currentOwner.setEmail(ownerFieldsDto.getEmail());
this.clinicService.saveOwner(currentOwner);
```

23. After the above change, save the file, then stop and start the spring boot app in command prompt.
24. After server restart, perform the steps to modify email and click on 'Update Owner' button, now owner details page should be displayed with modified email.



25. With this, edit owner with email is completed.

Activity 4: Capture email when adding an owner

1. Similar steps whatever implemented in edit owner can be done in the following file to add email field.

   `src\app\owners\owner-add\owner-add.component.html`

2. In the app running in the browser, adding a new owner can be performed by following the below links:

   OWNERS > SEARCH > 'Add Owner' button displayed below the table.

Activity 5: Unit Testing after inclusion of email

1. Run the 'ng test' command in a new terminal window.
2. Failures will be reported due to the missing email field.
3. Identify the locations where telephone is included in the files ending with spec.ts and then include the test data for email, which should fix all the unit test errors.

Time Required

2 Hours

Chapter – 9

Treatment Type

11. Database modifications for Treatment Type

Requirement

Soon, the pet clinic is planning to expand its operations by bringing in more high-end instruments, which will expand its services and operations. This will enable the pet clinic to perform complex surgeries and high-quality services to Pet owners. These are services that require longer stay in the clinic and requires highly qualified veterinarians. Hence, they did not want to name these services as visits, instead they wanted to call it as Treatment. Given this context, there is a need to manage the predefined list of Treatment Types that is going to be provided in the Pet Clinic. The initial task is to provide the possibility to manage a list of Treatment Types.

In this section, we will be focusing on defining the database related changes for including Treatment Type.

Activity 1: Create table for treatment type

1. Open the vs code window where the spring boot project is opened.
2. In the copilot chat window type the following prompt to create a table for treatment type.

```
create mysql create table script for treatment type having auto generated id and name as columns
```

3. This is the code generated.

```
CREATE TABLE `treatment_type` (
  `id` INT AUTO_INCREMENT,
  `name` VARCHAR(255) NOT NULL,
  PRIMARY KEY (`id`)
);
```

4. Create a new file **treatment-type.sql** in src\main\resources\db\mysql folder.
5. Paste the above generated script into treatment-type.sql.
6. Next, we will start inserting data into this table. Following are the treatment types that needs to be added into the table.
    a. Spay & Neuter Surgery

b. Soft Tissue Surgery
c. Emergency Surgery
d. Dental Cleaning and Polishing
e. Orthopedic Evaluation & Treatment

7. The below prompt requests for generating insert script for one treatment type.

```
for treatment type table generate insert script for
treatement type 'Spay & Neuter Surgery'
```

8. The above prompt generates the following code.

```
INSERT INTO `treatment_type` (`name`) VALUES ('Spay & Neuter Surgery');
```

9. Paste the generated code into treatement-type.sql after the create table script.
10. After pasting the code press enter key. We can see copilot starting to provide suggestions for inserting the next treatment type.

```
INSERT INTO `treatment_type` (`name`) VALUES ('Dental Cleaning & Polishing');
INSERT INTO `treatment_type` (`name`) VALUES ('Dental Surgery');
UES ('Dental Cleaning & Polishing');
LUES ('Dental Surgery (Extractions)');
```

11. Likewise include the insert scripts for other treatment types.
12. Execute the create table and insert scripts in MySQL workbench and check if data is inserted into treatment_type table.

id	name
1	Spay & Neuter Surgery
2	Soft Tissue Surgery
3	Emergency Surgery
4	Dental Cleaning & Polishing
5	Orthopedic Evaluation & Treatment
NULL	NULL

Time Required

30 Minutes

12. Build CRUD – Treatment Type Entity and Repository definition.

Requirement

Now that table and data is defined for Treatment Type in the database, we can start working on the backend code.

Activity 1: Create Entity Class for treatment type.

1. By the running the below prompt in the spring boot app vs code window, we can find out all the changes that are required for implementing CRUD operation.

```
This project uses Open API and it is integrated with controller.
Recommend all the changes that needs to done in this project to expose
CRUD operations for treatment type. Use the source files available in
src\main\java\org\springframework\samples\petclinic folder as reference
to provide the suggestion.
```

2. This is the response provided for the above prompt.

To expose CRUD operations for `TreatmentType` via a REST API, you'll need to make changes in several parts of your application. Here's a step-by-step plan:

1. **Create the `TreatmentType` Entity:** Create a new `TreatmentType` class in your `model` package. This class should have `id` and `name` fields, along with their getters and setters.
2. **Update the Repository:** Create a new `TreatmentTypeRepository` interface in your `repository` package. This interface should extend `JpaRepository` or `CrudRepository` and be parameterized with `TreatmentType` and `Integer`.
3. **Create the Service:** Create a new `TreatmentTypeService` interface and its implementation (`TreatmentTypeServiceImpl`) in your `service` package. This service should use `TreatmentTypeRepository` to perform CRUD operations.
4. **Create the Controller:** Create a new `TreatmentTypeController` class in your `controller` package. This controller should use `TreatmentTypeService` to handle HTTP requests for CRUD operations.
5. **Update the OpenAPI Configuration:** Update your `openapi.yml` file to include the `TreatmentType` schema and the paths for the CRUD operations.
6. **Update the Tests:** Create new tests for `TreatmentTypeController`, `TreatmentTypeService`, and `TreatmentTypeRepository` in your `test` package.

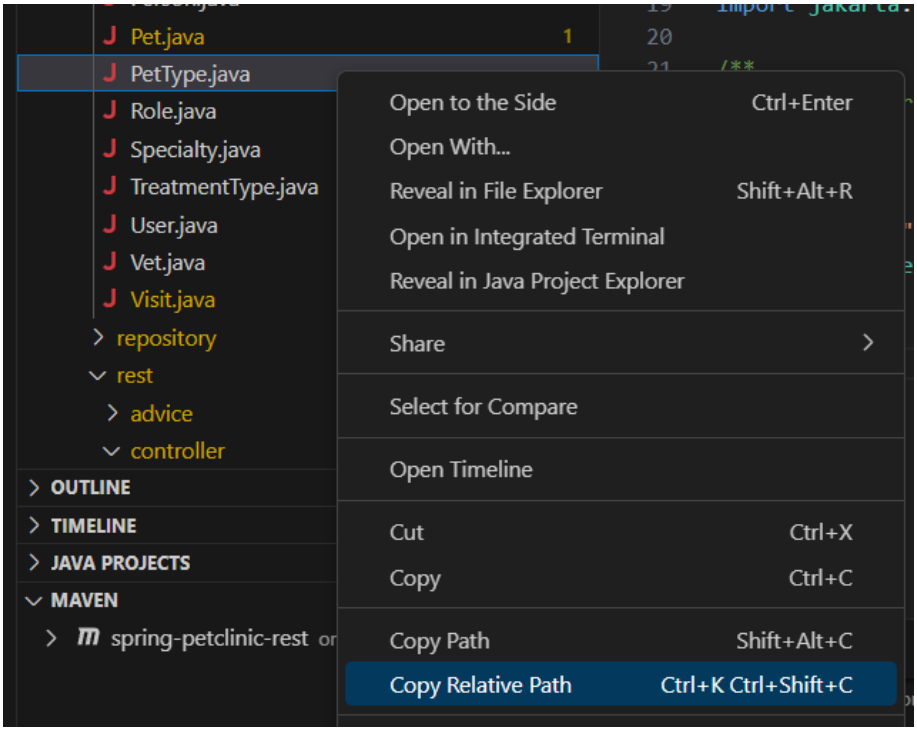
3. The first step is to create the entity class.
4. Before starting this activity, let us look into the existing entity code.

5. Go through the files in src\main\java\org\springframework\samples\petclinic \model folder.
6. Go through BaseEntity and NamedEntity to understand how inheritance-based definition of entity classes are implemented.
7. We can understand that any entity having only id and name can extend NamedEntity. For generating treatment type as named entity, we can use the below prompt to generate the entity class for treatment type.

```
use src\main\java\org\springframework\samples\petclinic\model\PetType.java as reference. this class uses inheritance from NamedEntity to generate named entity class. in a similar fashion generate entity code for treatment type.
```

NOTES:

- In the above prompt the complete relative path of PetType.java is provided. This can be obtained using VS Code. right click on the file or folder in the file explorer window and then select 'Copy relative path'



- Notice that even though the prompt mentions it as 'treatment type' it identifies the respective entity.

8. This is the output of the above prompt.

```java
package org.springframework.samples.petclinic.model;

import jakarta.persistence.Entity;
import jakarta.persistence.Table;

/**
 * Simple JavaBean domain object representing a treatment type.
 */
@Entity
@Table(name = "treatment_type")
public class TreatmentType extends NamedEntity {

}
```

9. Create a new file named TreatmentType.java in 'model' folder and the above generated code can be included as it is into this file.
10. In the generated code, verify if the table name generated matches with the treatment type table created earlier.

Activity 2: Define repository interface for treatment type.

The 'repository' folder contains all the interfaces required for interacting with the database. The folders 'jdbc', 'jpa', 'springdatajpa' contains different implementations based on the technology we want to use. The profile definition in application.properties helps to define which technology should be used. Currently 'springdatajpa' is what is configured; hence we will be implementing the repository under 'springdatajpa' folder and we will not implement 'jdbc' and 'jpa'.

First, we will implement the generation of interface for repository.

1. We are going to use PetTypeRepository.java file as reference to generate the code to treatment type.
2. Use the below prompt to generate the interface for repository.

```
use
src\main\java\org\springframework\samples\petclinic\repository\PetTypeRepository.java as reference to create repository interface for treatment type
```

3. This is the code generated.

```java
package org.springframework.samples.petclinic.repository;

import java.util.Collection;

import org.springframework.dao.DataAccessException;
import org.springframework.samples.petclinic.model.TreatmentType;

public interface TreatmentTypeRepository {

    TreatmentType findById(int id) throws DataAccessException;

    Collection<TreatmentType> findAll() throws DataAccessException;

    void save(TreatmentType treatmentType) throws DataAccessException;

    void delete(TreatmentType treatmentType) throws DataAccessException;

}
```

NOTE: We can see copilot generating precise code that we want. It has identified the right package. It has replaced the code with TreatmentType wherever required based on the reference code provided.

4. Create a file named TreatmentTypeRepository.java in 'repository' folder, then copy and paste the above code into the file created.

Activity 3: Spring Data JPA implementation for Treatment Type Repository

1. Use the below prompt to generate the Spring Data JPA implementation.

```
using
src\main\java\org\springframework\samples\petclinic\repository\springdatajpa\SpringDataSpecialtyRepository.java as reference, create the repository interface for treatment type
```

2. This is the code generated.

```java
package org.springframework.samples.petclinic.repository.springdatajpa;

import org.springframework.context.annotation.Profile;
import org.springframework.data.repository.Repository;
import org.springframework.samples.petclinic.model.TreatmentType;
import org.springframework.samples.petclinic.repository.TreatmentTypeRepository;

@Profile("spring-data-jpa")
public interface SpringDataTreatmentTypeRepository extends TreatmentTypeRepository, Repository<TreatmentType, Integer> {

}
```

3. Create a file name SpringDataTreatmentTypeRepository.java in 'repository/springdatajpa' folder. Then copy the above generated code and save the file.

Time Required

45 Minutes

13. Build CRUD – Treatment Type Service

Requirement

Repository implementation is done for treatment type, next is the implementation of service. Service layer is implemented in ClinicService.java and ClinicServiceImpl.java. These classes need to be modified to include treatment type related function.

Activity 1 - Modify ClinicService interface for Treatment Type

1. Open ClinicService.java from src\main\java\org\springframework\samples \petclinic\service folder.
2. Define a new function in the end by typing TreatmentType,

```
PetType findPetTypeByName(String name) throws DataAccessExce

TreatmentType findTreatmentTypeById(int treatmentTypeId);
```

   we can see copilot providing suggestion based on the existing methods defined in that file.
3. After accepting the suggestion and pressing enter, we can see the next suggestion.

```
TreatmentType findTreatmentTypeById(int treatmentTypeId);
Collection<TreatmentType> findAllTreatmentTypes() throws DataAccessException;
```

   Always go through the code to understand if copilot is generating relevant code and confirm the suggestion.
4. In a similar fashion, the rest of the service methods required can be quickly included.

```
TreatmentType findTreatmentTypeById(int treatmentTypeId);
Collection<TreatmentType> findAllTreatmentTypes() throws DataAccessException;
void saveTreatmentType(TreatmentType treatmentType) throws DataAccessException;
void deleteTreatmentType(TreatmentType treatmentType) throws DataAccessException;
```

Activity 2 - Modify ClinicService implementation for Treatment Type

1. Open ClinicServiceImpl.java.
2. Include TreatmentTypeRepository as instance variable.

```
@Service
public class ClinicServiceImpl implements ClinicService {

    private PetRepository petRepository;
    private VetRepository vetRepository;
    private OwnerRepository ownerRepository;
    private VisitRepository visitRepository;
    private SpecialtyRepository specialtyRepository;
    private PetTypeRepository petTypeRepository;
    private TreatmentTypeRepository treatmentTypeRepository;
```

3. Place the cursor after the last parameter of the constructor.

```
public ClinicServiceImpl(
        PetRepository petRepository,
        VetRepository vetRepository,
        OwnerRepository ownerRepository,
        VisitRepository visitRepository,
        SpecialtyRepository specialtyRepository,
        PetTypeRepository petTypeRepository) {
```

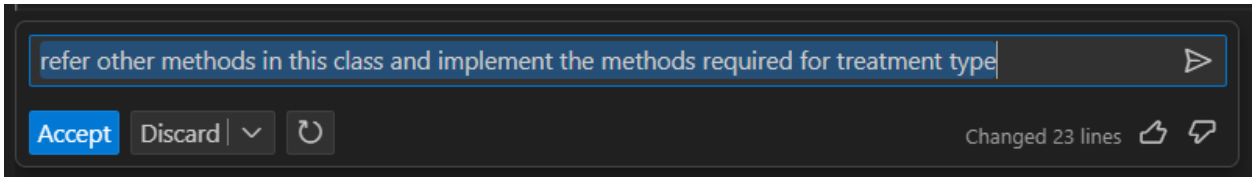
4. Press comma and then enter key.
5. We can see the copilot automatically suggesting inclusion of TreatmentTypeRepository.

```
public ClinicServiceImpl(
        PetRepository petRepository,
        VetRepository vetRepository,
        OwnerRepository ownerRepository,
        VisitRepository visitRepository,
        SpecialtyRepository specialtyRepository,
        PetTypeRepository petTypeRepository,
        TreatmentTypeRepository treatmentTypeRepository) {
```

6. Include the assignment of TreatementTypeRepository in the constructor implementation.

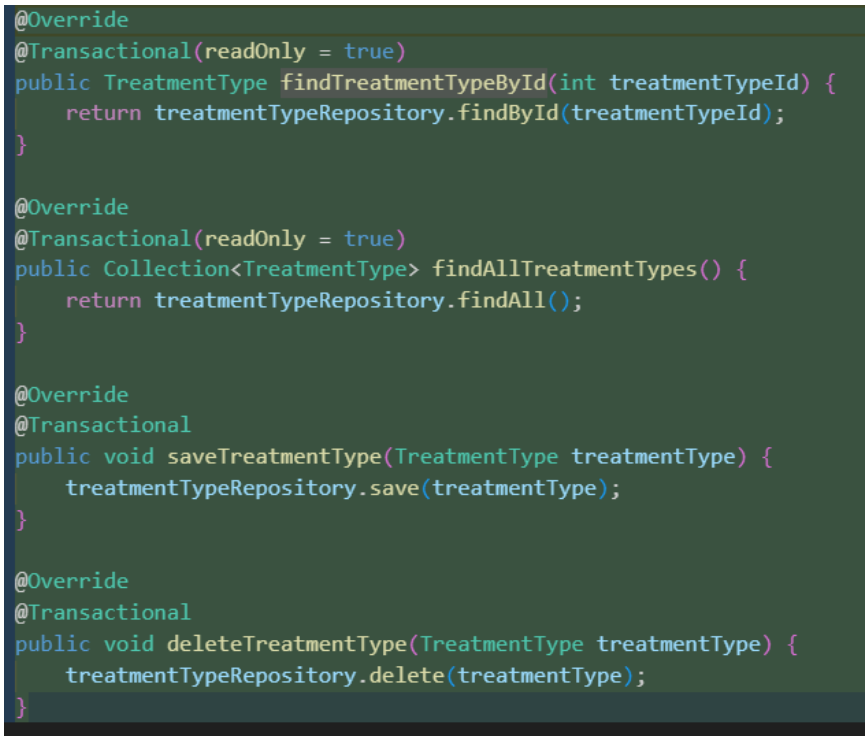
```
this.specialtyRepository = specialtyRepository;
this.petTypeRepository = petTypeRepository;
this.treatmentTypeRepository = treatmentTypeRepository;
```

7. After accepting the above suggestion, place the cursor after the last function implemented.
8. Press Ctrl+i, which will open a chat panel within the editor window.
9. Type the following prompt in the chat window.



10. Pressing enter key after typing the above prompt displays the generated code.
11. It displays the current state of the code in the left-hand side and the generated modified code in the right-hand side.

12. This is the code generated for the above prompt, the screenshot of the right-hand side is provided here.



```
@Override
@Transactional(readOnly = true)
public TreatmentType findTreatmentTypeById(int treatmentTypeId) {
    return treatmentTypeRepository.findById(treatmentTypeId);
}

@Override
@Transactional(readOnly = true)
public Collection<TreatmentType> findAllTreatmentTypes() {
    return treatmentTypeRepository.findAll();
}

@Override
@Transactional
public void saveTreatmentType(TreatmentType treatmentType) {
    treatmentTypeRepository.save(treatmentType);
}

@Override
@Transactional
public void deleteTreatmentType(TreatmentType treatmentType) {
    treatmentTypeRepository.delete(treatmentType);
}
```

13. We can see all CRUD operations for Treatment Type is implemented.
14. Click Accept button on the in-editor chat window which will include the code.
15. At times the code inserted contains some error, it may include an additional closing bracket or remove an existing closing bracket, which can be corrected after the code is inserted.

> **Copilot Feature:** In-editor chat can be triggered within the code editor window by pressing Ctrl + i shortcut key. We can directly type what we want to do and the code gets generated and inserted in the cursor location.

16. Save ClinicServiceImpl.java and check if it is free of compilation errors.

Time Required

30 Minutes

14. Build CRUD – Treatment Type Open API

Requirement

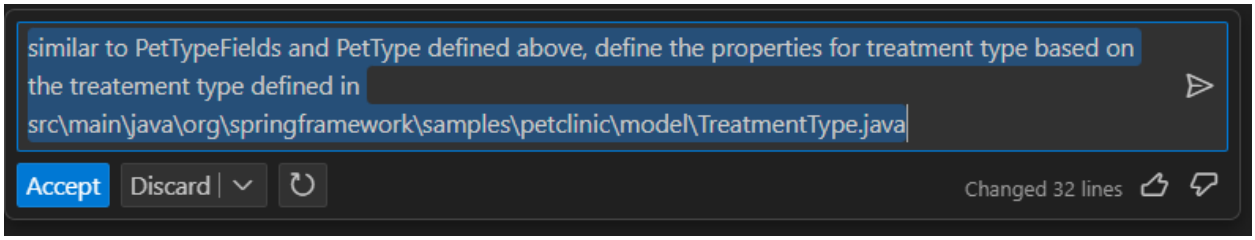
In this section we will implement the Open API related changes, so that the CRUD operations can be exposed as REST API.

Activity 1 – Configure TreatmentType fields in openapi.yml

1. Open openapi.yml
2. Place the cursor after PetType definition.

```
PetTypeFields: ···
PetType:
  title: Pet type
  description: A pet type.
  allOf:
    - $ref: '#/components/schemas/PetTypeFields'
    - type: object
      properties:
        id:
          title: ID
          description: The ID of the pet type.
          type: integer
          format: int32
          minimum: 0
          example: 1
          readOnly: true
      required:
        - id
```

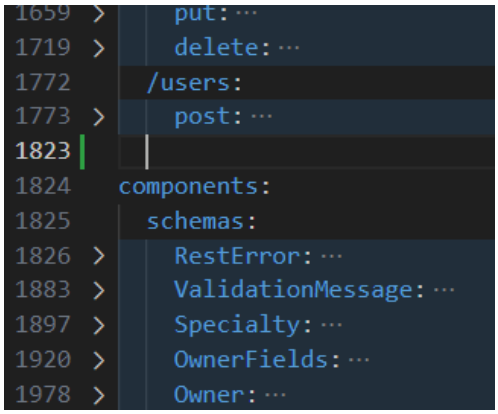
3. Enter the below prompt.



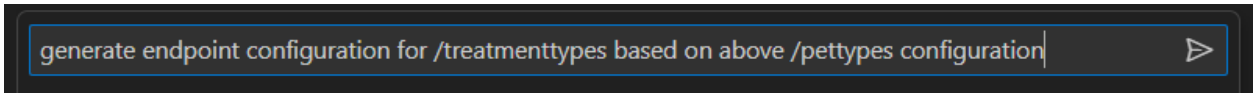
4. Compare the generated configuration, with PetTypeFields and PetType defined above, if the generated configuration looks good, then the code can be accepted.
5. Check if indentation is applied correctly and make corrections if required.
6. The above configuration helps to define the Treatment Type fields and this configuration helps in generating the DTO classes for Treatment Type.

Activity 2 – Configure TreatmentType endpoints in openapi.yml

1. In openapi.yml, place the cursor above the 'components' definition.



2. Press Ctrl+i and enter the below prompt to generate the endpoints for TreatmentType. We are asking to generate the endpoint configuration using whatever is defined for PetType. The /treatmenttypes endpoint will address the below to HTTP methods.
    a. GET method for getting all treatment types.
    b. POST method for adding a treatment type.



3. Accept the generated code
4. Correct if there are errors in indentation
5. Compare the generated code with the existing definition of /pettypes
6. If any sections of generated code are missing, it can be copied and pasted or a new prompt can be provided to generate that part of code.
7. When preparing this exercise, following code snippets were not generated.
    a. ETag configuration was missing for Response code 200. It was manually copied and pasted.
    b. Response configuration for 304 was missing, which was manually copied and pasted
8. Once /treatmenttypes configuration is completed as we can start /treatmenttypes/{treatmentTypeId}.
9. Use the below prompt to generate the GET endpoint configuration to get a specific treatment type based on treatment type id.

```
using /pettypes/{petTypeId} configuration in openapi.yaml file, generate configuration for http 'get' method code for treatment types as /treatmenttypes/{treatmentTypeId}
```

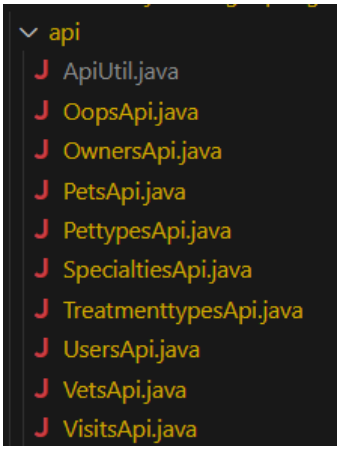
10. The above prompt does not generate the HTTP response code handling configuration. The following prompt will help get full configuration.

```
similar to pettypes generate complete configuration for responses 200, 304, 400, 404 and 500
```

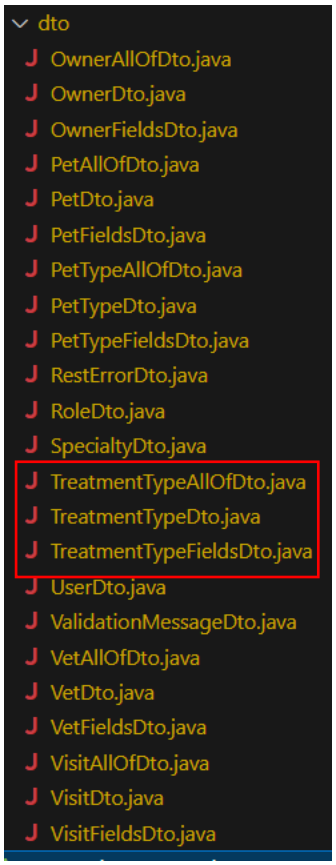
11. Use the previous-to-previous prompt and change ‘get’ to ‘put’ or ‘delete’ to generate the respective endpoint configuration for treatment type update and delete.
12. Find below a part of the configuration generated by copilot for PUT method. This is for your reference.

```
/treatmenttypes/{treatmentTypeId}:
  put:
    tags:
      - treatmenttype
    summary: Update an existing treatment type
    description: Update a treatment type by ID
    operationId: updateTreatmentTypeById
    parameters:
      - name: treatmentTypeId
        in: path
        description: ID of treatment type to update
        required: true
        schema:
          type: integer
          format: int64
    requestBody:
      description: Treatment type object that needs to be updated
      content:
        application/json:
          schema:
            $ref: '#/components/schemas/TreatmentType'
      required: true
  responses:
    '200':
      description: Successful operation
      content:
        application/json:
          schema:
            $ref: '#/components/schemas/TreatmentType'
```

13. After saving openapi.yml, in `target\generated-sources\openapi\src\main\java\org\springframework\samples\petclinic\rest\api` folder, we can see the file `TreatmenttypeApi.java` generated. This file will contain the generated methods, which are called for the Treatment Type REST API endpoints.



14. In `target\generated-sources\openapi\src\main\java\org\springframework\samples\petclinic\rest\dto` folder, we can see multiple files generated for treatment type.



15. With this the Open API configuration is concluded.

Time Required

1 Hour

15. Build CRUD – Treatment Type Mapper

Requirement

In this section we will generate the code for Treatment Type Mapper.

Activity 1 – Generate code for Mapper data transformation

1. Use the below prompt to generate the code for treatment type mapper.

```
using src\main\java\org\springframework\samples\petclinic\mapper\SpecialtyMapper.java generate mapper code for TreatmentType entity
```

2. Create a new file named TreatmentTypeMapper.java in 'mapper' folder.
3. Copy the code generated into this file.
4. In the generated code, change all references from List to Collection.
5. Remove the import of List if present.
6. This is how the final mapper code looks like.

```java
package org.springframework.samples.petclinic.mapper;

import org.mapstruct.Mapper;
import org.springframework.samples.petclinic.rest.dto.TreatmentTypeDto;
import org.springframework.samples.petclinic.model.TreatmentType;

import java.util.Collection;

/**
 * Map TreatmentType & TreatmentTypeDto using mapstruct
 */
@Mapper
public interface TreatmentTypeMapper {
    TreatmentType toTreatmentType(TreatmentTypeDto treatmentTypeDto);

    TreatmentTypeDto toTreatmentTypeDto(TreatmentType treatmentType);

    Collection<TreatmentTypeDto> toTreatmentTypeDtos(Collection<TreatmentType> treatmentTypes);

    Collection<TreatmentType> toTreatmentTypes(Collection<TreatmentTypeDto> treatmentTypeDtos);
}
```

Time Required
30 Minutes

16. Build CRUD – Treatment Type Controller

Requirement

In this section we will generate the code for Treatment Type controller.

Activity 1 – Generate Controller class.

1. Create a new file named TreatmentTypeRestController.java in src\main\java\org\springframework\samples\petclinic\rest\controller folder.
2. Use the below prompt to generate the base level code for the controller.

```
using
src\main\java\org\springframework\samples\petclin
ic\rest\controller\SpecialtyRestController.java
generate code package, imports, class definition
code with annotations for treatment type controller
```

3. Copy and paste the generated code into the file created in Step 1.
4. There may few compilation errors, which can be easily resolved.
5. This is how the final code looks like after resolving the compilation errors. For brevity, the imports are not provided in the screenshot.

```
@RestController
@CrossOrigin(exposedHeaders = "errors, content-type")
@RequestMapping("api")
public class TreatmentTypeRestController implements TreatmenttypesApi {

    private final ClinicService clinicService;

    private final TreatmentTypeMapper treatmentTypeMapper;

    public TreatmentTypeRestController(ClinicService clinicService, TreatmentTypeMapper treatmentTypeMapper) {
        this.clinicService = clinicService;
        this.treatmentTypeMapper = treatmentTypeMapper;
    }

    // CRUD methods will go here...
}
```

We can notice the following aspects taken care by copilot:

- @CrossOrigin @RequestMapping code generated
- The code generated implements TreatmenttypeApi which was generated by Open API.
- It has included appropriate service and mapper classes with initialization using constructor.

Activity 2 – Method to get all treatment types

1. As a reference go through the code in SpecialtyRestController.java to understand how controller class and its functions are implemented.

2. We are going to provide SpecialtyRestController.java as reference to copilot to implement the coding for TreatmentTypeRestController, which can be achieved using the below prompt. In this prompt, we are clearly specifying the source file to use, the function in the source file and the destination file.

```
using listSpecialties() function in
src\main\java\org\springframework\samples\petclinic\rest\controller\S
pecialtyRestController.java generate the controller method for
src\main\java\org\springframework\samples\petclinic\rest\controller\T
reatmentTypeRestController.java
```

3. Cross verify the code generated by the above prompt with the listSpecialties() function defined in SpecialtyRestController.java.
4. Check if the function name generated matches with the function name generated in TreatmenttypesApi, if not it can be manually corrected.
5. The final generated code and compilation error fixed code should look like this.

```
@PreAuthorize("hasRole(@roles.VET_ADMIN)")
@Override
public ResponseEntity<List<TreatmentTypeDto>> getTreatmentTypes() {
    List<TreatmentTypeDto> treatmentTypes = new ArrayList<>();
    treatmentTypes.addAll(treatmentTypeMapper.toTreatmentTypeDtos(this.clinicService.findAllTreatmentTypes()));
    if (treatmentTypes.isEmpty()) {
        return new ResponseEntity<>(HttpStatus.NOT_FOUND);
    }
    return new ResponseEntity<>(treatmentTypes, HttpStatus.OK);
}
```

6. With the above code the endpoint to get all treatment types is implemented.

Activity 3 – Method to get a specific treatment type

1. Use the below prompt to generate the code to get a specific treatment type.

```
using getSpecialty() function as reference in
src\main\java\org\springframework\samples\petclinic\rest\controller\S
pecialtyRestController.java, generate the code to get a specfic
treatment type for
src\main\java\org\springframework\samples\petclinic\rest\controller\T
reatmentTypeRestController.java
```

2. This is the code generated.

```
@PreAuthorize("hasRole(@roles.VET_ADMIN)")
@Override
public ResponseEntity<TreatmentTypeDto> getTreatmentType(@PathVariable("treatmentTypeId") int treatmentTypeId) {
    TreatmentType treatmentType = this.clinicService.findTreatmentTypeById(treatmentTypeId);
    if (treatmentType == null) {
        return new ResponseEntity<>(HttpStatus.NOT_FOUND);
    }
    return new ResponseEntity<>(treatmentTypeMapper.toTreatmentTypeDto(treatmentType), HttpStatus.OK);
}
```

3. Fix the compilation error by changing 'int' to 'Integer'.

```
@PreAuthorize("hasRole(@roles.VET_ADMIN)")
@Override
public ResponseEntity<TreatmentTypeDto> getTreatmentTypeById(@PathVariable("treatmentTypeId") Integer treatmentTypeId) {
    TreatmentType treatmentType = this.clinicService.findTreatmentTypeById(treatmentTypeId);
    if (treatmentType == null) {
        return new ResponseEntity<>(HttpStatus.NOT_FOUND);
    }
    return new ResponseEntity<>(treatmentTypeMapper.toTreatmentTypeDto(treatmentType), HttpStatus.OK);
}
```

Activity 4 – Method to add a treatment type

1. Use the below prompt for adding a treatment type.

   using addSpecialty() function as reference in src\main\java\org\springframework\samples\petclinic\rest\controller\SpecialtyRestController.java, generate the code to add a new treatment type for src\main\java\org\springframework\samples\petclinic\rest\controller\TreatmentTypeRestController.java

2. In the code generated when implementing this exercise, additional parameters like bindingResult and ucBuilder were included as arguments of the generated function. To fix this the following modified prompt can be used.

   modify the code to not include bindingResult and ucBuilder as parameter, then similar addSpeciality() function of src\main\java\org\springframework\samples\petclinic\rest\controller\SpecialtyRestController.java create the UriComponentsBuilder internally.

3. In the generated code, the error handling part is not required. The below prompt would fix it.

   thank you, exclude the bindingResult and error handling part of the code, as it is already handled in openapi configuration

   This is the code generated.

```
@PreAuthorize("hasRole(@roles.VET_ADMIN)")
@Override
public ResponseEntity<TreatmentTypeDto> addTreatmentType(@RequestBody @Valid TreatmentTypeDto treatmentTypeDto) {
    TreatmentType treatmentType = this.clinicService.saveTreatmentType(treatmentTypeMapper.toTreatmentType(treatmentTypeDto));
    UriComponentsBuilder ucBuilder = UriComponentsBuilder.newInstance();
    HttpHeaders headers = new HttpHeaders();
    headers.setLocation(ucBuilder.path("/api/treatmenttypes/{id}").buildAndExpand(treatmentType.getId()).toUri());
    return new ResponseEntity<>(treatmentTypeMapper.toTreatmentTypeDto(treatmentType), headers, HttpStatus.CREATED);
}
```

4. This is the final version of code after fixing the compilation errors.

```java
@PreAuthorize("hasRole(@roles.VET_ADMIN)")
@Override
public ResponseEntity<TreatmentTypeDto> addTreatmentType(@RequestBody @Valid TreatmentTypeDto treatmentTypeDto) {
    TreatmentType treatmentType = treatmentTypeMapper.toTreatmentType(treatmentTypeDto);
    this.clinicService.saveTreatmentType(treatmentType);
    UriComponentsBuilder ucBuilder = UriComponentsBuilder.newInstance();
    HttpHeaders headers = new HttpHeaders();
    headers.setLocation(ucBuilder.path("/api/treatmenttypes/{id}").buildAndExpand(treatmentType.getId()).toUri());
    return new ResponseEntity<>(treatmentTypeMapper.toTreatmentTypeDto(treatmentType), headers, HttpStatus.CREATED);
}
```

Activity 5 – Method to update a treatment type

1. In the earlier activities, we can see the code generated is not in alignment with the reference code provided, hence the below prompt is used for generating update method, it provides emphasis on strictly sticking to the reference code provided, which is highlighted below.

   using updateSpecialty() function as reference in src\main\java\org\springframework\samples\petclinic\rest\controller\SpecialtyRestController.java, generate the code to update a treatment type for src\main\java\org\springframework\samples\petclinic\rest\controller\TreatmentTypeRestController.java, follow previous suggestions and ensure that the code is generated similar to how it is defined in updateSpeciality() function

2. The above prompt makes it possible to get the right version of code in the first instance itself. This is the final version of code after fixing compilation errors.

```java
@PreAuthorize("hasRole(@roles.VET_ADMIN)")
@Override
public ResponseEntity<TreatmentTypeDto> updateTreatmentTypeById(
    @PathVariable("treatmentTypeId") Integer treatmentTypeId,
    @RequestBody @Valid TreatmentTypeDto treatmentTypeDto
) {
    TreatmentType currentTreatmentType = this.clinicService.findTreatmentTypeById(treatmentTypeId);
    if (currentTreatmentType == null) {
        return new ResponseEntity<>(HttpStatus.NOT_FOUND);
    }
    currentTreatmentType.setName(treatmentTypeDto.getName());
    this.clinicService.saveTreatmentType(currentTreatmentType);
    return new ResponseEntity<>(treatmentTypeMapper.toTreatmentTypeDto(currentTreatmentType), HttpStatus.NO_CONTENT);
}
```

Activity 6 – Method to delete a treatment type

1. Similar to update code generate, delete function can also be generated using the below prompt.

using deleteSpecialty() function as reference in src\main\java\org\springframework\samples\petclinic\rest\controller\SpecialtyRestController.java, generate the code to delete a treatment type for src\main\java\org\springframework\samples\petclinic\rest\controller\TreatmentTypeRestController.java, follow previous suggestions and ensure that the code is generated similar to how it is defined in deleteSpecialty() function

2. This is the final version of the code.

```
@PreAuthorize("hasRole(@roles.VET_ADMIN)")
@Override
public ResponseEntity<TreatmentTypeDto> deleteTreatmentType(@PathVariable("treatmentTypeId") Integer treatmentTypeId) {
    TreatmentType treatmentType = this.clinicService.findTreatmentTypeById(treatmentTypeId);
    if (treatmentType == null) {
        return new ResponseEntity<>(HttpStatus.NOT_FOUND);
    }
    this.clinicService.deleteTreatmentType(treatmentType);
    return new ResponseEntity<>(HttpStatus.NO_CONTENT);
}
```

Activity 7 – Restart the server

To reflect the changes made to expose the treatment type CRUD operations, the build needs to be freshly executed.

1. Go to the command prompt where the spring boot app is running.
2. Press Ctrl+C to stop the server.
3. Run the below command to freshly create a build

   `mvnw clean package`
4. Run the below command to run the app

   `mvnw spring-boot:run`
5. Run the below URL in the browser to check if existing REST API calls are working as expected.

   `http://localhost:9966/petclinic/api/owners`

After completion of the above 7 activities. The implementation of CRUD operation for Treatment Type is completed. In the next section, we will verify if the CRUD operations work as expected.

Time Required

1 Hour 30 Minutes

Chapter – 10

Treatment Type - Testing

17. Test CRUD of Treatment Type backend REST API

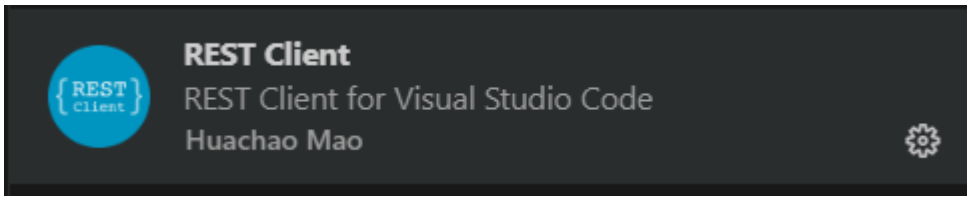
Requirement

In this section we will test the REST API code for Treatment Type controller.

Activity 1 – Install REST Client extension.

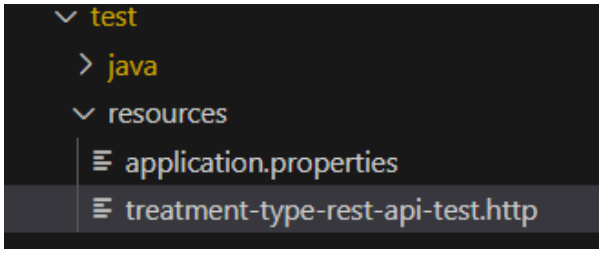
1. Open VS Code window where the pet clinic spring boot app is opened.
2. Go to extensions.

3. Search for 'rest client' and install the extension. This extension helps to run REST API calls from vs code.

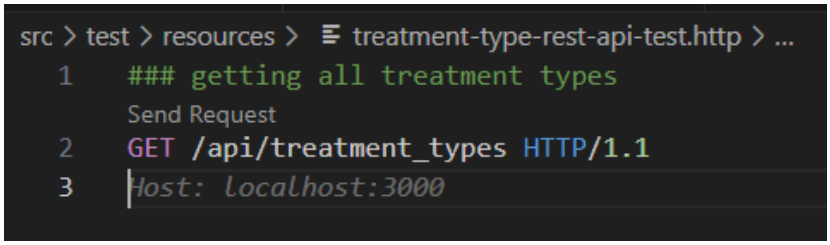


Activity 2 – Test get all treatment types

4. Create file named 'treatment-type-rest-api-test.http' in 'resources' folder.



5. Type the following in the above created file. Line 2 and 3 represents the raw HTTP request sent to the server. Notice that, here as well copilot suggests code.



6. Complete the HTTP request with updated port number.

```
### getting all treatment types
Send Request
GET /petclinic/api/treatmenttypes HTTP/1.1
Host: localhost:9966
```

7. Then click on 'Send Request' option displayed by vs code above the HTTP request, which will send the request to the server.
8. The HTTP raw response from the server will be displayed in the right-side window.
9. Below is the output, that should be received. If any error is received, the respective troubleshooting needs to be done.

Response(2303ms)

```
HTTP/1.1 200
Vary: Origin, Access-Control-Request-Method, Access-Control-Request-Headers
X-Content-Type-Options: nosniff
X-XSS-Protection: 0
Cache-Control: no-cache, no-store, max-age=0, must-revalidate
Pragma: no-cache
Expires: 0
X-Frame-Options: DENY
Content-Type: application/json
Transfer-Encoding: chunked
Date: Sun, 10 Dec 2023 09:44:43 GMT
Connection: close

[
  {
    "name": "Spay & Neuter Surgery",
    "id": 1
  },
  {
    "name": "Soft Tissue Surgery",
    "id": 2
  },
  {
    "name": "Emergency Surgery",
    "id": 3
  },
  {
    "name": "Dental Cleaning & Polishing",
    "id": 4
  },
  {
    "name": "Orthopedic Evaluation & Treatment",
    "id": 5
  }
]
```

10. Verify the response output received in the following aspects, which means that the request is working as expected:
    a. Header response status is 200
    b. Response body contains the array of treatment types that were present in the database.

Activity 3 – Test getting treatment type base on id

1. Open treatment-type-rest-api-test.http
2. After the existing request create a couple of blank lines and enter the following comment.

```
### getting treatment type based on id
GET /petclinic/api/treatmenttypes/1 HTTP/1.1
```

3. After pressing enter key, we can see the first line of the HTTP request suggested by copilot. This is suggested by copilot based on the comment we typed.
4. Subsequently, we can see copilot generating the second line of request as well.

```
### getting treatment type based on id
Send Request
GET /petclinic/api/treatmenttypes/1 HTTP/1.1
Host: localhost:9966
```

5. This time it has automatically identified the port number as 9966.
6. This HTTP request gets the treatment type that has 1 as its id.
7. Clicking on 'Send Request' should provide the below response, which means that this endpoint is working as expected.

```
HTTP/1.1 200
Vary: Origin, Access-Control-Request-Method, Access-Control-Request-Headers
X-Content-Type-Options: nosniff
X-XSS-Protection: 0
Cache-Control: no-cache, no-store, max-age=0, must-revalidate
Pragma: no-cache
Expires: 0
X-Frame-Options: DENY
Content-Type: application/json
Transfer-Encoding: chunked
Date: Sun, 10 Dec 2023 10:06:02 GMT
Connection: close

{
  "name": "Spay & Neuter Surgery",
  "id": 1
}
```

Activity 4 – Test adding a new treatment type

1. Enter the POST HTTP request as specified below in 'treatment-type-rest-api-test.http'. Here as well copilot helps to quickly complete this request.

```
### adding new treatment type
Send Request
POST /petclinic/api/treatmenttypes HTTP/1.1
Host: localhost:9966
Content-Type: application/json

{
  "name": "Complete Body Cleanse"
}
```

2. The above request should give back the below response.

```
HTTP/1.1 201
Vary: Origin, Access-Control-Request-Method, Access-Control-Request-Headers
Location: /api/treatmenttypes/7
X-Content-Type-Options: nosniff
X-XSS-Protection: 0
Cache-Control: no-cache, no-store, max-age=0, must-revalidate
Pragma: no-cache
Expires: 0
X-Frame-Options: DENY
Content-Type: application/json
Transfer-Encoding: chunked
Date: Sun, 10 Dec 2023 10:06:41 GMT
Connection: close

{
  "name": "Complete Body Cleanse",
  "id": 7
}
```

3. Check in treatment type table if a new entry is added.

id	name
1	Spay & Neuter Surgery
2	Soft Tissue Surgery
3	Emergency Surgery
4	Dental Cleaning & Polishing
5	Orthopedic Evaluation & Treatment
7	Complete Body Cleanse
NULL	NULL

Activity 5 – Test updating a treatment type

1. Open 'treatment-type-rest-api-test.http'
2. Key in this HTTP request for update as specified below

```
### updating treatment type
Send Request
PUT /petclinic/api/treatmenttypes/7 HTTP/1.1
Host: localhost:9966
Content-Type: application/json

{
  "name": "Complete Body Sanitization"
}
```

3. Send the above request.
4. Refer expected response.

```
HTTP/1.1 204
Vary: Origin, Access-Control-Request-Method, Access-Control-Request-Headers
X-Content-Type-Options: nosniff
X-XSS-Protection: 0
Cache-Control: no-cache, no-store, max-age=0, must-revalidate
Pragma: no-cache
Expires: 0
X-Frame-Options: DENY
Content-Type: application/json
Date: Sun, 10 Dec 2023 10:10:33 GMT
Connection: close
```

5. Check in table if it is updated.

id	name
1	Spay & Neuter Surgery
2	Soft Tissue Surgery
3	Emergency Surgery
4	Dental Cleaning & Polishing
5	Orthopedic Evaluation & Treatment
7	Complete Body Sanitization
NULL	NULL

Activity 6 – Test deletion of treatment type

1. Open 'treatment-type-rest-api-test.http'

2. Key in this HTTP request for update as specified below.

```
### deleting treatment type
Send Request
DELETE /petclinic/api/treatmenttypes/7 HTTP/1.1
Host: localhost:9966
```

3. Check if the below response is received.

```
HTTP/1.1 204
Vary: Origin, Access-Control-Request-Method, Access-Control-Request-Headers
X-Content-Type-Options: nosniff
X-XSS-Protection: 0
Cache-Control: no-cache, no-store, max-age=0, must-revalidate
Pragma: no-cache
Expires: 0
X-Frame-Options: DENY
Date: Sun, 10 Dec 2023 10:13:05 GMT
Connection: close
```

4. Check in table if the respective treatment type is removed.

id	name
1	Spay & Neuter Surgery
2	Soft Tissue Surgery
3	Emergency Surgery
4	Dental Cleaning & Polishing
5	Orthopedic Evaluation & Treatment
NULL	NULL

Time Required

45 Minutes

18. Build CRUD – Preparing the angular app.

Now that all the REST API for Treatment Type is built, we can start implementing the frontend part in angular. In this section we will perform activities in the angular app, that will prepare us to start the development of the CRUD operations of treatment type.

Activity 1 – Create Module and Listing Component

1. Open the vs code window of the angular app.
2. Open a new terminal window (View > Terminal)
3. Run the below command to create a new module for treatment type

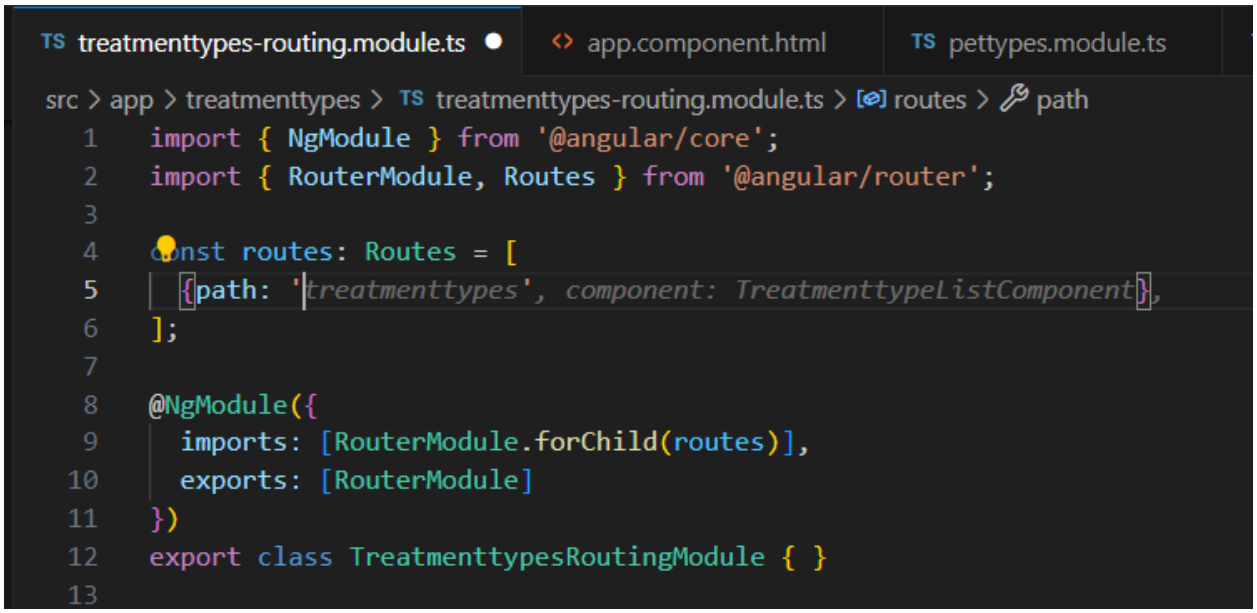
```
ng g m treatmenttypes --routing
```

The –routing generates routing for the new module.

4. Create a component within the new module to display the list of treatment types

```
ng g c treatmenttypes/treatmenttype-list
```

5. Open treatmenttypes-routing.module.ts and define the route for the new component



Activity 2 – Include navigation link for treatment type

1. Open app.component.html
2. Enter the following prompt in copilot chat.

```
in app.component.html include a new link for treatment type and name the link as 'Treatment Types'
```

3. This is the code generated.

```
<li>
  <a routerLink="/treatment-types/list" routerLinkActive="active" title="treatment-types">
    <span class="glyphicon glyphicon-plus-sign" aria-hidden="true"></span>
    <span> Treatment Types</span>
  </a>
</li>
```

Note that copilot has used the existing navigation link code and generated the HTML code in alignment with the existing classes used. Notice that it has identified its own icon for Treatment Types.

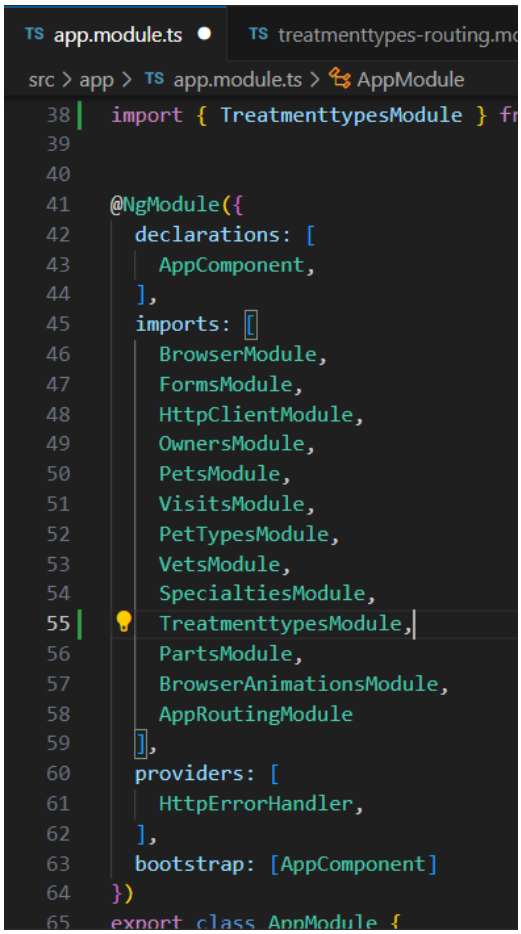
4. We would like to have the navigation link as 'treatmenttypes' instead of 'treatment-types', which can be modified using the below prompt.

   ```
   name the routerlink as "treatmenttypes"
   ```

5. Include the generated code into app.component.html after the closing <li> tag of specialties.

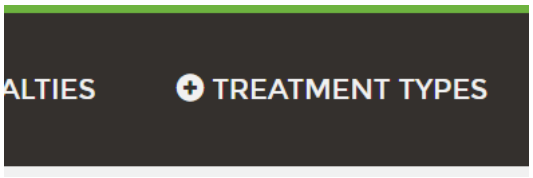
Activity 3 – Add treatment type module to the main app module

1. Open app.module.ts and include TreatmentTypeModule in the imports section.

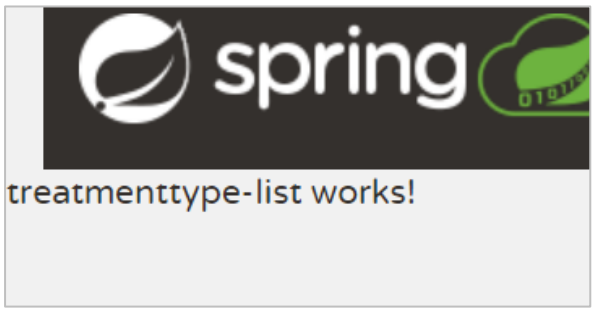


2. After saving all the files, check if there are no compilation errors in the terminal window where the angular app is running.

   Open browser and check if navigation link for treatment type is displayed after specialties link.



3. Hovering over 'TREATMENT TYPES' navigation link should change the background color to green.
4. Clicking on 'TREATMENT TYPES' navigation link should display the default message from the newly created component.



5. After clicking on 'TREATMENT TYPES', the background color should retain the green color to indicate that it is the currently selected item.

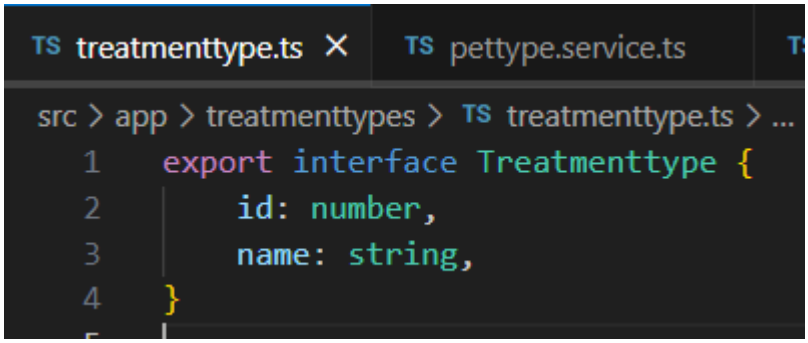
Time Required
30 Minutes

19. Build CRUD – Defining the service for treatment type.

In this section we will define the service class for treatment type which will contain the method to get all the treatment types.

Activity 1 – Define model.

1. Create TreatmentType interface in src\app\treatmenttypes folder.



Activity 2 – Create service

1. Run the below command in terminal, which will create the service class.

```
ng g s treatmenttypes/treatmenttype
```

2. Use the below prompt which will prepare the service class to have appropriate instance variables and endpoint URLs.

```
Generate properties and constructors for
src\app\treatmenttypes\treatmenttype.service.ts based on
src\app\pettypes\pettype.service.ts
```

3. This is the code generated.

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

@Injectable({
  providedIn: 'root'
})
export class TreatmenttypeService {
  private apiServerUrl: string;

  constructor(private http: HttpClient) {
    this.apiServerUrl = 'http://localhost:8
  }

  // Add methods for CRUD operations here
}
```

4. If the above code is compared with existing service class we can see various deviations, which can be addressed by using the below prompt.

similar to PetTypeService include url based on environment variable, include handleError property and constructor with error handler

5. Then this is the code generated, which can be pasted into treatmenttype.service.ts.

```
import { Injectable } from '@angular/core';
import { HttpClient } from '@angular/common/http';
import { Observable } from 'rxjs';
import { environment } from '../../environments/environment';
import { HandleError, HttpErrorHandler } from '../error.service';

@Injectable()
export class TreatmenttypeService {
  entityUrl = environment.REST_API_URL + 'treatmenttypes';

  private readonly handlerError: HandleError;

  constructor(private http: HttpClient, private httpErrorHandler: HttpErrorHandler) {
    this.handlerError = httpErrorHandler.createHandleError('TreatmenttypeService');
  }

  // Add methods for CRUD operations here
}
```

Activity 3 – Create service method to get all treatment types

1. Open treatmenttype.service.ts
2. Place the cursor after the closing curly braces of constructor body
3. Press enter key
4. Copilot provides suggestion code to get all treatment types from backend.

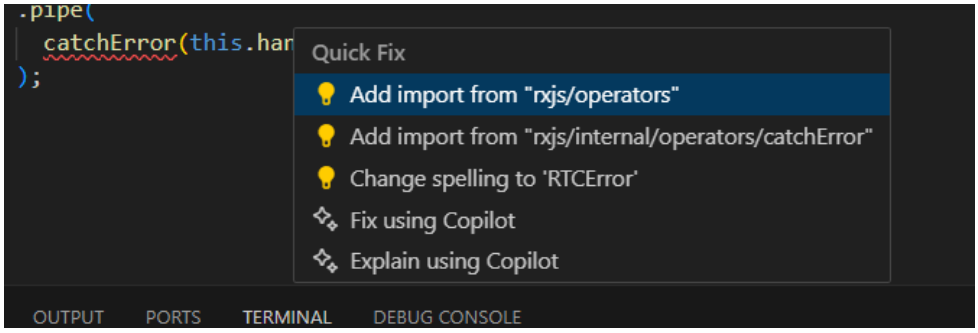
```
    this.handlerError = httpErrorHandler.createHandleError('Treatme
  }

  getTreatmenttypes(): Observable<Treatmenttype[]> {
    return this.http.get<Treatmenttype[]>(this.entityUrl)
      .pipe(
        catchError(this.handlerError('getTreatmenttypes', []))
      );
  }
```

5. Copilot gets this reference code from the PetTypeService reference code that we have already provide.
6. Once the above suggested code is accepted by pressing TAB, we can see few compilation errors, which can be fixed by including appropriate imports.

7. The imports can be quickly done by hovering the mouse over the respective error indicator and then clicking on Quick Fix, which will provide a drop down list from which the respective import can be included.

```
getTreatmenttypes(): Observable<Treatmenttype[]> {
  return this.http.get<Treatmenttype[]>(this.entityUrl)
    .pipe(
      catchError(this.handlerError('getTreatmenttypes', []))
    );
}
```

8. The catchError can be imported from rxjs/operators.



Activity 4 – Unit testing for treatment type service

1. Generate test cases using the below prompt.

```
using src\app\pettypes\pettype.service.spec.ts generate the
code for making changes in
src\app\treatmenttypes\treatmenttype.service.spec.ts
```

2. This is the code generated.

```
import { inject, TestBed, waitForAsync } from '@angular/core/testing';
import { TreatmenttypeService } from './treatmenttype.service';
import { HttpClient } from '@angular/common/http';
import { HttpClientTestingModule, HttpTestingController } from '@angular/common/http/testing';

describe('TreatmenttypeService', () => {
  beforeEach(() => {
    TestBed.configureTestingModule({
      // Import the HttpClient mocking services
      imports: [HttpClientTestingModule],
      providers: [TreatmenttypeService]
    });
  });

  it('should ...', waitForAsync(inject([HttpTestingController], (treatmenttypeService: TreatmenttypeService, http: HttpClient) => {
    console.log('TreatmenttypeService should ...');
    expect(treatmenttypeService).toBeTruthy();
  })));
});
```

3. Copy and paste the above code into treatmenttype.service.spec.ts.
4. If 'ng test' is already running in the terminal, stop it using Ctrl+C.
5. Start the 'ng test' command again, this ensure that newly added test cases are also executed.
6. The above command reports the test results in a Chrome browser window, in which we can check if the TreatmentTypeService test case is passed.

```
TreatmenttypeService
  • should ...
```

Activity 5 – Implementing rest of the service method

1. In treatmentype.service.ts, place the cursor after the closing curly braces of the getTreatmentTypes() function.
2. Press enter key.

```
getTreatmenttypes(): Observable<Treatmenttype[]> {
  return this.http.get<Treatmenttype[]>(this.entityUrl)
    .pipe(
      catchError(this.handlerError('getTreatmenttypes', []))
    );
}

getTreatmenttypeById(treatmenttypeId: string): Observable<Treatmenttype> {
  return this.http.get<Treatmenttype>(this.entityUrl + '/' + treatmenttypeId)
    .pipe(
      catchError(this.handlerError('getTreatmenttypeById', {} as Treatmenttype))
    );
}
```

3. In a similar fashion, methods for add, modify and delete functions for TreatmentType can be generated. Verify if the generated code matches with how the coding is defined in other service classes.

Time Required

1 Hour

20. Build CRUD – Implement component for listing treatment types.

Now that the service class is ready, we can start implementing the components. As a first step, in this section, we will implement the component that displays the list of all treatment types.

Activity 1 – Generate component TypeScript code.

1. We are going to implement the layout and features exactly like pet type.
2. Hence let us use the below prompt which generates the code for treatment type listing component based on the pet type code base.

```
using src\app\pettypes\pettype-list\pettype-list.component.ts generate the property definitions for src\app\treatmenttypes\treatmenttype-list\treatmenttype-list.component.ts
```

3. This is the code generated.

```typescript
import { Component, OnInit } from '@angular/core';
import { TreatmenttypeService } from '../treatmenttype.service';
import { TreatmentType } from '../treatmenttype';

@Component({
  selector: 'app-treatmenttype-list',
  templateUrl: './treatmenttype-list.component.html',
  styleUrls: ['./treatmenttype-list.component.css']
})
export class TreatmenttypeListComponent implements OnInit {
  treatmentTypes: TreatmentType[];

  constructor(private treatmenttypeService: TreatmenttypeService) { }

  ngOnInit() {
    this.treatmenttypeService.getTreatmentTypes().subscribe(
      (data: TreatmentType[]) => this.treatmentTypes = data,
      (error: any) => console.log(error)
    );
  }
}
```

4. If we compare this code with pet type component code, we can see lot of gaps, the following prompt will address those gaps.

```
include all properties from pettype-list.component.ts
```

NOTE: Notice we have not defined any context and we have provided very less words, copilot relates to whatever was asked before and provided the revised code.

5. Based on the above prompt, following is the code generated.

```
import { Component, OnInit } from '@angular/core';
import { TreatmenttypeService } from '../treatmenttype.service';
import { TreatmentType } from '../treatmenttype';
import { Router } from '@angular/router';
import { finalize } from 'rxjs/operators';

@Component({
  selector: 'app-treatmenttype-list',
  templateUrl: './treatmenttype-list.component.html',
  styleUrls: ['./treatmenttype-list.component.css']
})
export class TreatmenttypeListComponent implements OnInit {
  treatmentTypes: TreatmentType[];
  errorMessage: string;
  responseStatus: number;
  isTreatmentTypesDataReceived: boolean = false;
  isInsert = false;

  constructor(private treatmenttypeService: TreatmenttypeService, private router: Router) {
    this.treatmentTypes = [] as TreatmentType[];
  }

  ngOnInit() {
    this.treatmenttypeService.getTreatmentTypes().pipe(
      finalize(() => {
        this.isTreatmentTypesDataReceived = true;
      })
    ).subscribe(
      treatmentTypes => this.treatmentTypes = treatmentTypes,
      error => this.errorMessage = error as any
    );
  }
```

6. There may be few compilation errors because of the difference between TreatmentType and Treatmenttype, which can be manually fixed.
7. Once the compilation errors are fixed, we can implement the other functions required in this component.
8. Place the cursor after the closing bracket of ngOnInit() function.
9. Then press enter key two times
10. We can see the function for deleting a treatment type generated based on the Pet Type component code provided as reference earlier, we can accept the suggestion and generate the code.

```
      treatmentTypes => this.treatmentTypes = treatme
      error => this.errorMessage = error as any
    );
  }

  deleteTreatmenttype(treatmenttype: Treatmenttype) {
    this.treatmenttypeService.deleteTreatmenttype(tre
      response => {
        this.responseStatus = response;
        this.treatmentTypes = this.treatmentTypes.fil
      },
      error => this.errorMessage = error as any);
  }
}
```

11. In a similar fashion, all the other methods as well can be implemented.

```
onNewTreatmenttype(newTreatmentType: Treatmenttype) {
  this.treatmentTypes.push(newTreatmentType);
  this.showAddTreatmenttypeComponent();
}

showAddTreatmenttypeComponent() {
  this.isInsert = !this.isInsert;
}

showEditTreatmenttypeComponent(updatedTreatmentType: Treatmenttype) {
  this.router.navigate(['/treatmenttypes', updatedTreatmentType.id.toString(), 'edit']);
}

gotoHome() {
  this.router.navigate(['/welcome']);
}
```

Activity 2 – Generate component HTML code

1. Use the below prompt to generate the HTML code.

```
using src\app\pettypes\pettype-list\pettype-list.component.html generate html code for listing treatment types in src\app\treatmenttypes\treatmenttype-list\treatmenttype-list.component.html
```

2. The above prompt generates the below code.

```
<div *ngIf="errorMessage" class="alert alert-danger">
  {{errorMessage}}
</div>

<div *ngIf="isTreatmentTypesDataReceived">
  <h2>Treatment Types</h2>
  <div *ngIf="treatmentTypes && treatmentTypes.length">
    <table class="table table-striped">
      <thead>
        <tr>
          <th>Name</th>
          <th>Actions</th>
        </tr>
      </thead>
      <tbody>
        <tr *ngFor="let treatmentType of treatmentTypes">
          <td>{{treatmentType.name}}</td>
          <td>
            <button (click)="editTreatmentType(treatmentType.id)" class="btn btn-prima
            <button (click)="deleteTreatmentType(treatmentType.id)" class="btn btn-dar
          </td>
        </tr>
      </tbody>
    </table>
  </div>
  <div *ngIf="!treatmentTypes || !treatmentTypes.length">
    <p>No treatment types found.</p>
  </div>
</div>
```

3. The code is more or less the same but the Home and Add buttons are missing, which can be generated using the below prompt.

```
similar to pettype-list.component.html include buttons for Home and Add. Also enclose the html within container-fluid and xd-container
```

4. The above prompt will generate all the necessary code required.

```
<div class="container-fluid">
    <div class="container xd-container">
        <h2>Treatment Types</h2>
        <button class="btn btn-default" routerLink="/">Home</button>
        <button class="btn btn-default" routerLink="/treatmenttypes/add">Add</button>
        <table id="treatmenttypes" class="table table-striped">
            <thead>
                <tr>
                    <th>Name</th>
                    <th></th>
                    <th></th>
                </tr>
            </thead>
            <tbody>
                <tr *ngFor="let treatmentType of treatmentTypes; let element_id=index">
                    <td>
                        <input [id]=element_id [readonly]="true" type="text" class="form-control" [(ngModel)]="treatmentType.name" name="treatmentType_name" />
                    </td>
                    <td>
                        <button class="btn btn-default" (click)="showEditTreatmentTypeComponent(treatmentType)">Edit</button>
                        <button class="btn btn-default" (click)="deleteTreatmentType(treatmentType)">Delete</button>
                    </td>
                </tr>
            </tbody>
        </table>
        <div *ngIf="isInsert">
            <!-- <app-pettype-add (newPetType)="onNewTreatmenttype($event)">...</app-pettype-add> -->
        </div>
        <div>
            <button *ngIf="isTreatmentTypesDataReceived" class="btn btn-default" (click)="gotoHome()">Home</button>
            <button *ngIf="isTreatmentTypesDataReceived" class="btn btn-default" (click)="showAddTreatmenttypeComponent()"> Add </button>
        </div>
    </div>
</div>
```

5. Comment the code that includes the add component, which will be used later.

```
            </tbody>
        </table>
        <div *ngIf="isInsert">
            <!-- <app-pettype-add (newPetType)="onNewTreatmenttype($event)">...</app-pettype-add> -->
        </div>
        <div>
```

6. Include this entire code into treatmenttype-list.component.html and save the file.
7. In the terminal, the code compilation may fail with following errors.

```
Error: src/app/treatmenttypes/treatmenttype-list/treatmenttype-list.component.html:17:99 - error NG8002: Can't bind to 'ngModel' since it isn't a
known property of 'input'.

17                        <input [id]=element_id [readonly]="true" type="text" class="form-control" [(ngModel)]="treatmentType.name" name="treatm
entType_name" />
                                                                                                    ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

  src/app/treatmenttypes/treatmenttype-list/treatmenttype-list.component.ts:9:16
    9   templateUrl: './treatmenttype-list.component.html',
                     ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    Error occurs in the template of component TreatmenttypeListComponent.


Error: src/app/treatmenttypes/treatmenttype-list/treatmenttype-list.component.html:17:112 - error TS2322: Type 'Event' is not assignable to type '
string'.

17                        <input [id]=element_id [readonly]="true" type="text" class="form-control" [(ngModel)]="treatmentType.name" name="treatm
entType_name" />
                                                                                                               ~~~~~~~~~~~~~~~~~~~~~~~~~~
```

8. This can be fixed by including the FormsModule in src\app\owners\owners.module.ts, under 'imports' section.
9. Now the compilation errors would have gone but clicking on TREATMENT TYPES in the browser will display the following error message in the browser console.

```
2 ▶ERROR Error: Uncaught (in promise):                core.mjs:10920
  NullInjectorError: R3InjectorError(AppModule)[TreatmenttypeService ->
  TreatmenttypeService]:
    NullInjectorError: No provider for TreatmenttypeService!
  NullInjectorError: R3InjectorError(AppModule)[TreatmenttypeService ->
  TreatmenttypeService]:
    NullInjectorError: No provider for TreatmenttypeService!
      at NullInjector.get (core.mjs:8888:27)
      at R3Injector.get (core.mjs:9327:33)
      at R3Injector.get (core.mjs:9327:33)
      at ChainedInjector.get (core.mjs:13418:36)
      at lookupTokenUsingModuleInjector (core.mjs:4524:39)
      at getOrCreateInjectable (core.mjs:4572:12)
      at Module.ɵdirectiveInject (core.mjs:11214:19)
      at NodeInjectorFactory.TreatmenttypeListComponent_Factory [as
  factory] (treatmenttype-list.component.ts:12:40)
      at getNodeInjectable (core.mjs:4778:44)
      at createRootComponent (core.mjs:13673:35)
      at resolvePromise (zone.js:1193:31)
      at resolvePromise (zone.js:1147:17)
      at zone.js:1260:17
      at _ZoneDelegate.invokeTask (zone.js:402:31)
      at core.mjs:27127:55
      at AsyncStackTaggingZoneSpec.onInvokeTask (core.mjs:27127:36)
      at _ZoneDelegate.invokeTask (zone.js:401:60)
      at Object.onInvokeTask (core.mjs:27437:33)
      at _ZoneDelegate.invokeTask (zone.js:401:60)
      at Zone.runTask (zone.js:173:47)
```

10. To find out the reason for this issue, the following prompt was included, which is nothing but the main error message displayed above.

```
No provider for TreatmenttypeService!
```

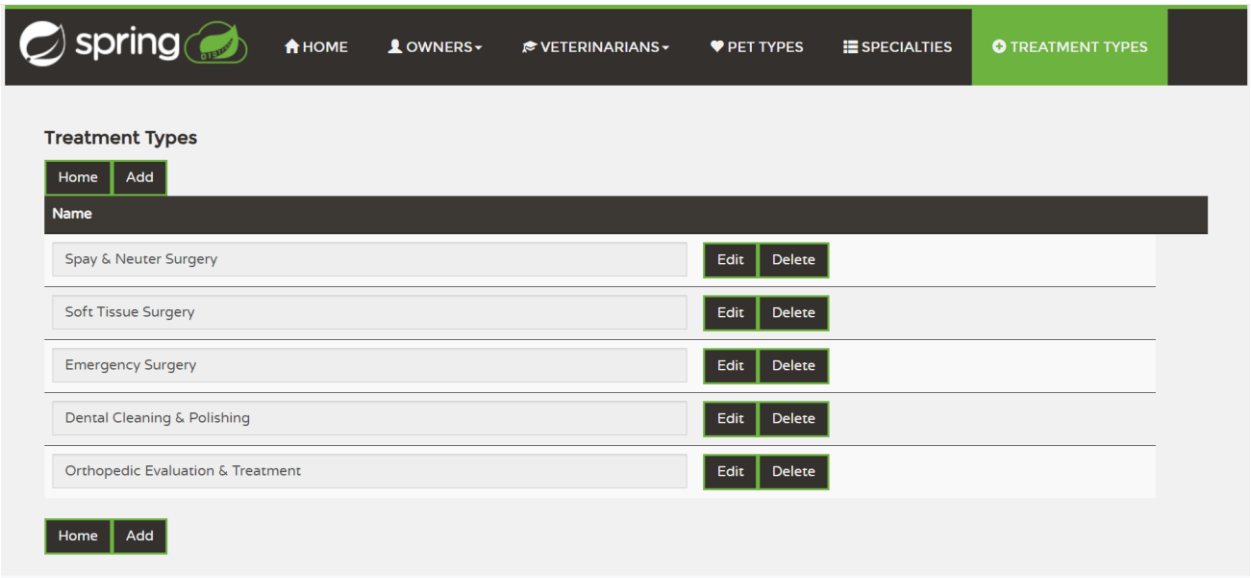
11. Copilot correctly identified that the service is not provided in the module, and it shows the code snippet where changes needs to be made.

```
import { NgModule } from '@angular/core
import { CommonModule } from '@angular/
import { TreatmenttypeService } from '.

@NgModule({
  imports: [
    CommonModule
    // other imports here
  ],
  providers: [TreatmenttypeService],
  // other module configurations here
})
export class YourModule { }
```

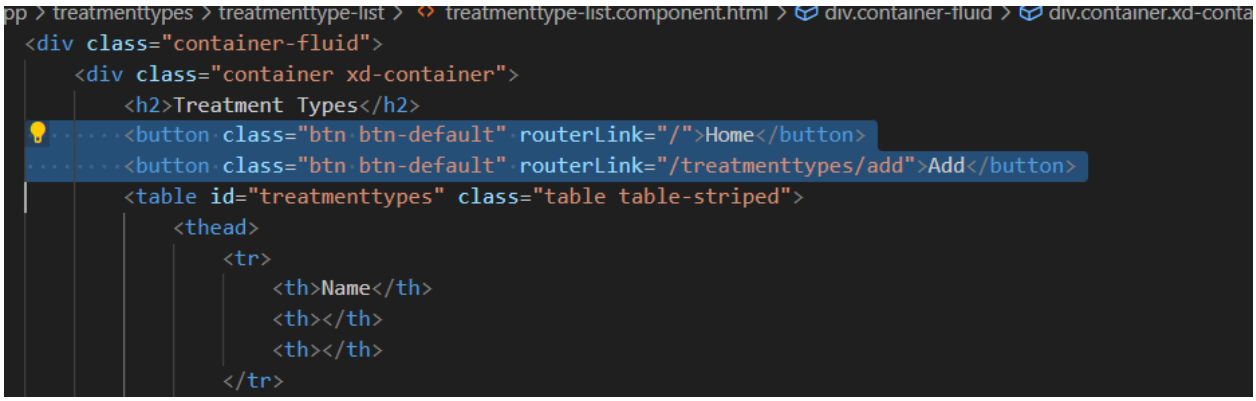
12. Based on the above suggestion the following line was included in src\app\treatmenttypes\treatmenttypes.module.ts.

```
 ],
 providers: [TreatmenttypeService]
)
```

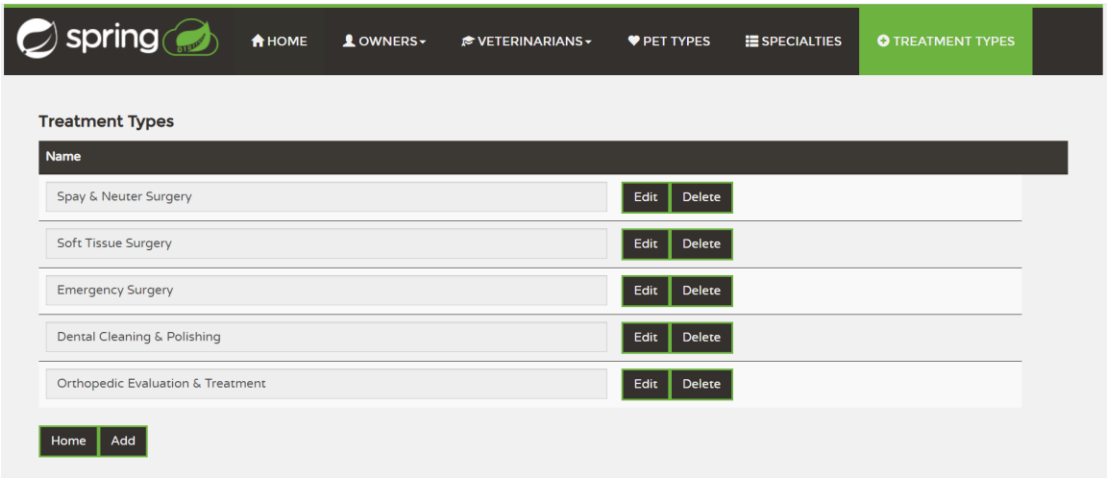
13. After the above changes the page should work displaying the treatment types.



14. When implementing this exercise, additional buttons were generated below the heading, which were removed.



15. This is how it looks like after the correction.



Time Required

1 Hour

21. Build CRUD – Implementing Add Treatment type.

In the section, we will implement the component for adding a new treatment type.

Activity 1 – Add Treatment Type component.

1. Create a new component for adding a treatment type.

```
ng g c treatmenttypes/treatmenttype-add
```

2. Use the below two prompts to generate the TypeScript code for add component.

```
using src\app\pettypes\pettype-add\pettype-add.component.ts generate code for treatment type add component src\app\treatmenttypes\treatmenttype-add\treatmenttype-add.component.ts
```

```
similar to pettype-add.component.ts include properties for error message and output event emitter, also include the emit event call on submission
```

3. Use the below two prompts to generate the HTML code for add component.

```
using src\app\pettypes\pettype-add\pettype-add.component.html generate html code for adding treatment type in src\app\treatmenttypes\treatmenttype-add\treatmenttype-add.component.html
```

```
apply similar container tags, styling and formatting from pettype-add.component.html
```

4. Uncomment the following line in treatmenttype-list.component.html, to enable the display of add treatment type component.

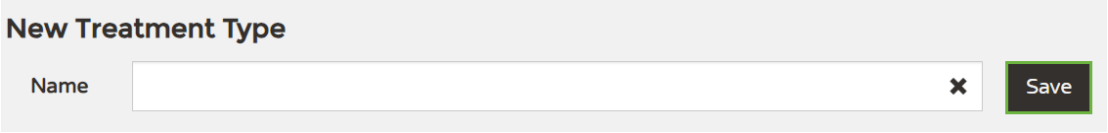
```
<!-- <app-pettype-add (newPetType)="onNewTreatmenttype($event)">...</app-pettype-add> -->
```

5. Modify the component name as specified below and the event name as well.

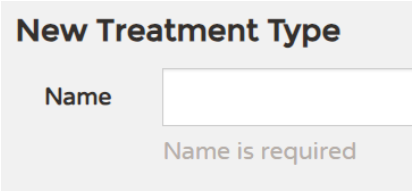
```
<app-treatmenttype-add (newTreatmentType)="onNewTreatmenttype($event)">...</app-treatmenttype-add>
```

6. Save the files.
7. The next step is to check if add feature works.

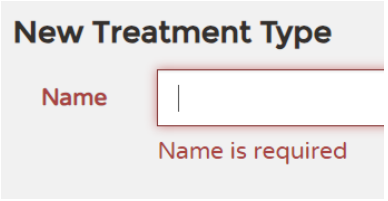
8. Follow steps below to test if add treatment type works:
    a. Click TREATMENT TYPES in navigation.
    b. Click Add button below the listed treatment types, which will display a form below.



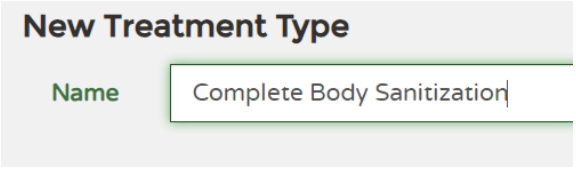
    c. Click on Save button which should not submit the form and it should display a message to enter the name.



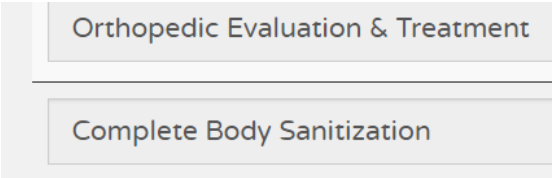
    d. Typing something and clearing the name input field should display required field validation error message.



    e. Type a new treatment type



    f. Click on Save button, which should add the new Treatment Type in the listing table above.



    g. If these features work as expected, then implementation of add treatment type feature is completed.

Activity 2 – Edit Treatment Type

1. Create component for editing

```
ng g c treatmenttypes/treatmenttype-edit
```

2. Define route to edit

```
const routes: Routes = [
  {path: 'treatmenttypes', component: TreatmenttypeListComponent},
  {path: 'treatmenttypes/:id/edit', component: TreatmenttypeEditComponent}
];
```

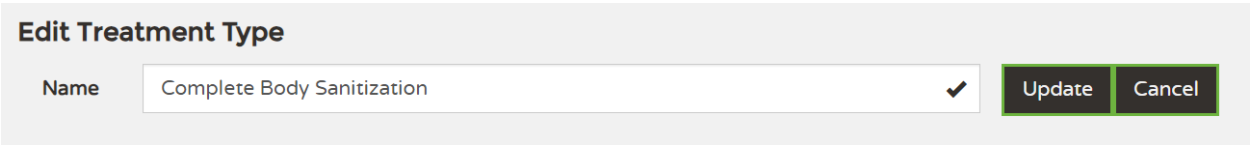
3. Use the below prompt to generate the TypeScript code for the component.

```
using src\app\pettypes\pettype-edit\pettype-edit.component.ts generate edit component for treatment type in src\app\treatmenttypes\treatmenttype-edit\treatmenttype-edit.component.ts
```

4. Use the below prompt to generate HTML template code.

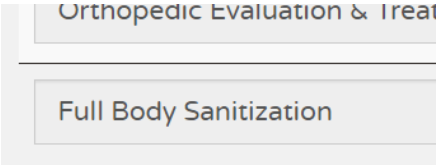
```
using src\app\pettypes\pettype-edit\pettype-edit.component.html generate edit treatment type html template for src\app\treatmenttypes\treatmenttype-edit\treatmenttype-edit.component.html, generate similar containers and style formatting
```

5. Implement following steps to check if edit treatment type works.
    a. Click 'TREATMENT TYPES'
    b. Click on Edit button of 'Complete Body Sanitization', which should display treatment type edit component, with the respective treatment type populated in the input element.



    c. Verify if Cancel button navigates the screen back to list screen and then come back to this screen.
    d. Clear the treatment type in the input element and check if validation error message is displayed.

e. Enter treatment type as 'Full Body Sanitization' and click on Update button, which should navigate back to listing screen, with the last entry in the table modified as 'Full Body Sanitization'.



Activity 3 – Delete Treatment Type

1. The coding for deleting treatment type is already in place. We just need to verify if it works.
2. Click 'TREATMENT TYPES'
3. Click on Delete button of 'Full Body Sanitization' and check if the respective entry is removed from the table.

With the above implementation, enhancements to pet clinic app is completed.

In the subsequent sections, we will cover various other aspects where copilot can be used.

Time Required

2 Hours

Chapter – 11

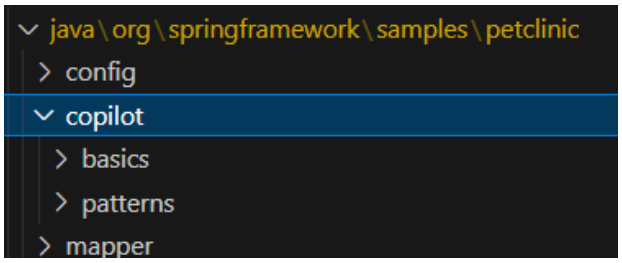
GitHub Copilot for Programming Basics

22. Using copilot for programming language basics

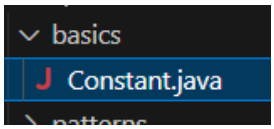
In this section, we will see how copilot can be used to generate basic programming constructs. We will be implementing this on Java programming language. The coding will be implemented in a separate folder in the Spring Pet Clinic REST project.

Activity 1 – Constant

1. Open the vs code window of the spring pet clinic rest project.
2. Create a new folder named 'copilot' under 'petclinic' folder.



3. Under 'copilot' folder create two folders 'basics' and 'patterns'.
4. In 'basics' folder, create a new file named 'Constant.java'



5. Using below prompt generate the code for constant.

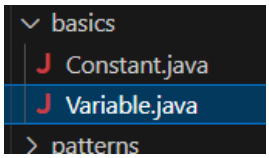
```
include a main method that demonstrates how to define a constant in java and print the constant
```

6. Include the generated code into Constant.java
7. Run the main method using the Run option displayed about the main() method.

```
Run | Debug
public static void main(String[] args) {
    System.out.println(MY_CONSTANT);
}
```

Activity 2 – Variable

1. Create a new file named 'Variable.java'



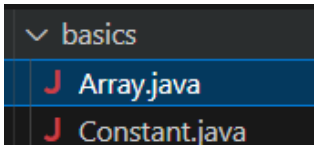
2. Using below prompt generate the code for variable.

```
include main method that demonstrates how to define each primitive data type and print the data
```

3. Include the code in Variable.java, run the program and verify if it works as expected.

Activity 3 – Array

1. Create a new file named Array.java



2. Generate array example code using below prompt, include the code in Array.java and run the code.

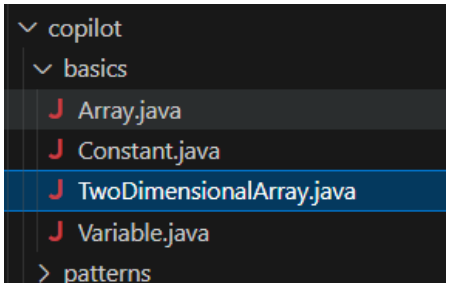
```
include a main method that demonstrates definition of an array of weekdays, then print the weekdays using while loop
```

3. Use below prompt to generate variations in the already implemented code.

```
modify the code to print only the weekdays
```

Activity 4 – Two Dimensional Array

1. Create a new file



2. Generate using below prompt.

include main method to demonstrate defining a 4 x 4 matrix of numbers, then print it in matrix format

3. In the resulting code of the above prompt, the numbers displayed are not aligned. Use the below prompt to align the numbers displayed in 4 x 4 matrix.

   modify code to display the numbers in aligned format

Activity 5 – Class and Object

1. Create file name Employee.java
2. Generate code using below prompt.

   create employee class with appropriate attributes

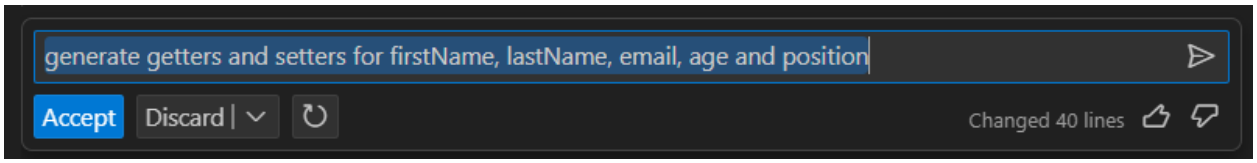
3. This is the code generated.

```
public class Employee {
    private String firstName;
    private String lastName;
    private String email;
    private int age;
    private String position;

    // getters and setters
}
```

4. The package line is not generated, when included in the file and when typing package, we can see the code generated by copilot.
5. Setters and getters can be generated as we type

```
// getters and setters
public String getFirstName() {
    return firstName;
}

public void setFirstName(String firstName) {
    if (firstName == null || firstName.isEmpty()) {
        throw new IllegalArgumentException("First name cannot be empty");
    }
    this.firstName = firstName;
}
```

6. We can remove unwanted code from the above setter method and then the subsequent code in setter methods does not include the code we don't want to have.
7. The in-editor chat can also be used to generate the getters and setters.



8. The other miscellaneous methods can be generated using the below prompt.

```
in employee.java include hashcode, tostring and equals functions
```

9. We can also generate the main method code to demonstrate how to use the employee class.

```
include main method in Employee.java to demonstrate the usage of Employee class and print the employee data
```

Activity 6 – Complex data model representation

1. Provide high level requirement as prompt.

```
java beans to be created to represent coffee shop, it should have classes to store coffee shop details, menu and bill
```

2. This is the high-level code generated.

```
public class CoffeeShop {
    private String name;
    private String location;
    private Menu menu;

    // getters and setters
}

public class Menu {
    private List<Coffee> coffeeList;

    // getters and setters
}

public class Coffee {
    private String name;
    private double price;

    // getters and setters
}

public class Bill {
    private CoffeeShop coffeeShop;
    private List<Coffee> orderedCoffees;

    // getters and setters
}
```

3. The generated code can be optimized using improved prompts.

```
regenerate the classes keeping in mind that the coffee shop may contain other menu items as well, primarily avoid a class named coffee
```

4. Code generated.

```java
public class CoffeeShop {
    private String name;
    private String location;
    private Menu menu;

    // getters and setters
}

public class Menu {
    private List<MenuItem> menuItems;

    // getters and setters
}

public class MenuItem {
    private String name;
    private double price;

    // getters and setters
}

public class Bill {
    private CoffeeShop coffeeShop;
    private List<MenuItem> orderedItems;

    // getters and setters
}
```

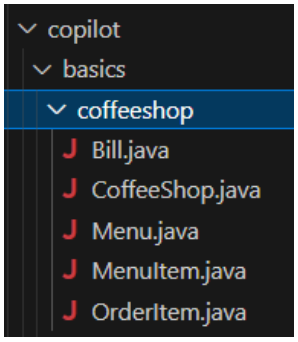
5. More details required for billing can be generated using the below prompt.

```
in Bill class, the orderedItems does not contain the information about quantity, line total and grand total, regenerate respective classes to store these details
```

6. The above prompt will improve the following three class.

```java
public class MenuItem {
    private String name;
    private double price;

    // getters and setters
}

public class OrderItem {
    private MenuItem menuItem;
    private int quantity;
    private double lineTotal;

    // getters and setters
}

public class Bill {
    private CoffeeShop coffeeShop;
    private List<OrderItem> orderedItems;
    private double grandTotal;

    // getters and setters
}
```

7. Create a new folder named 'coffeeshop' under 'basics' folder and generate separate Java files for each one of the above generated classes.



8. Copy and paste the respective class code into each file.
9. Using inline suggestions provided by copilot constructor, getters, setters and toString() functions can be generated.
10. In OrderItem.java, don't generate setLineTotal() function. Request copilot to generate the code to calculate line total.

```
include a function to calculate line total
```

11. This is the code generated.

```java
public void setMenuItem(MenuItem menuItem) {
    this.menuItem = menuItem;
    calculateLineTotal();
}

public void setQuantity(int quantity) {
    this.quantity = quantity;
    calculateLineTotal();
}

private void calculateLineTotal() {
    if (menuItem != null) {
        this.lineTotal = this.quantity * this.menuItem.getPrice();
    }
}
```

12. Using similar prompts as above, generate grand total calculation code in Bill.java.

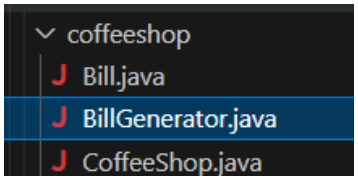
```java
private double calculateGrandTotal() {
    double grandTotal = 0.0;
    for (OrderItem orderItem : orderedItems) {
        grandTotal += orderItem.getLineTotal();
    }
    return grandTotal;
}
```

13. Copilot helps generating appropriate code in constructor and other functions in Bill.java

```java
public Bill(CoffeeShop coffeeShop, List<OrderItem> orderedItems) {
    this.coffeeShop = coffeeShop;
    this.orderedItems = orderedItems;
    this.grandTotal = calculateGrandTotal();
}
```

```java
public void setOrderedItems(List<OrderItem> orderedItems) {
    this.orderedItems = orderedItems;
    this.grandTotal = calculateGrandTotal();
}
```

14. Create a file name BillGenerator.java, which will use all the created classes.



15. The main method for BillGenerator.java can be generated using the below prompt.

include main function in BillGenerator class to demonstrate the usage of CoffeeShop, Menu, MenuItem, OrderItem and Bill classes

16. In the generated code, fix minor compilation errors. This is the output when running the program.

```
Bill [coffeeShop=CoffeeShop [name=The Coffee Shop, location=123 Main St, menu=Menu [menuItems=[MenuItem [name=Coffee, price=2.5], MenuItem [name=Tea, price=2.0]]]], orderedItems=[OrderItem [menuItem=MenuItem [name=Coffee, price=2.5], quantity=2, lineTotal=5.0], OrderItem [menuItem=MenuItem [name=Tea, price=2.0], quantity=1, lineTotal=2.0]], grandTotal=7.0]
```

17. The bill is not printed in a readable format, which can be corrected using the below prompt.

in Bill.java, modify toString() function to display the data in a proper bill format

18. The above prompt provide code gives the below result.

```
Location: 123 Main St
Ordered Items:
2 x Coffee = 5.0
1 x Tea = 2.0
Grand Total: 7.0
```

19. The above result can be further improved using the below prompt.

thanks, make further corrections, the labels "Coffee Shop", "Location" and "Ordered Items" are not required, in ordered items, include menu item price, then properly align menu item name, quantity, price and line total

NOTE: The prompt does not mention about bill display, yet it understands.

20. This is the generated code.

```java
@Override
public String toString() {
    StringBuilder sb = new StringBuilder();
    sb.append(coffeeShop.getName()).append("\n");
    sb.append(coffeeShop.getLocation()).append("\n\n");
    for (OrderItem orderItem : orderedItems) {
        sb.append(String.format("%-20s %5d x %6.2f = %6.2f\n",
            orderItem.getMenuItem().getName(),
            orderItem.getQuantity(),
            orderItem.getMenuItem().getPrice(),
            orderItem.getLineTotal()));
    }
    sb.append("\nGrand Total: ").append(String.format("%.2f", grandTotal));
    return sb.toString();
}
```

21. The above code provides the below result.

```
The Coffee Shop
123 Main St

Coffee                  2 x   2.50 =   5.00
Tea                     1 x   2.00 =   2.00

Grand Total: 7.00
```

22. The generated toString() code uses for loop to iterate through the ordered items. The following prompt can be used to convert the for loop to use streams.

```
in toString() function of Bill.java, convert the for loop to use streams
```

23. Following is the generated code for the above prompt.

```java
@Override
public String toString() {
    StringBuilder sb = new StringBuilder();
    sb.append(coffeeShop.getName()).append("\n");
    sb.append(coffeeShop.getLocation()).append("\n\n");
    orderedItems.stream().forEach(orderItem ->
        sb.append(String.format("%-20s %5d x %6.2f = %6.2f\n",
            orderItem.getMenuItem().getName(),
            orderItem.getQuantity(),
            orderItem.getMenuItem().getPrice(),
            orderItem.getLineTotal()))
    );
    sb.append("\nGrand Total: ").append(String.format("%.2f", grandTotal));
    return sb.toString();
}
```

24. Unit test cases can be quickly generated using the below prompt for OrderItem.java

```
create unit test cases for testing OrderItem.java
```

25. Following is the code generated.

```java
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.assertEquals;

public class OrderItemTest {

    @Test
    public void testCalculateLineTotal() {
        // Arrange
        MenuItem coffee = new MenuItem("Coffee", 2.5);
        OrderItem orderItem = new OrderItem();
        orderItem.setMenuItem(coffee);
        orderItem.setQuantity(2);

        // Act
        double lineTotal = orderItem.getLineTotal();

        // Assert
        assertEquals(5.0, lineTotal, "Line total should be 5.
    }

    @Test
    public void testToString() {
        // Arrange
        MenuItem coffee = new MenuItem("Coffee", 2.5);
        OrderItem orderItem = new OrderItem();
        orderItem.setMenuItem(coffee);
        orderItem.setQuantity(2);

        // Act
        String str = orderItem.toString();

        // Assert
        assertEquals("OrderItem [menuItem=Coffee, quantity=2,
    }
```

Time Required

2 Hours 30 Minutes

Chapter – 12

GitHub Copilot for Implementing Design Patterns

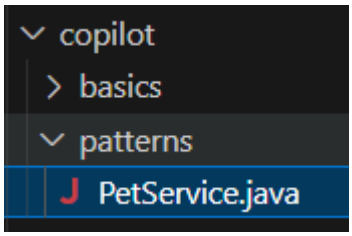
23. Using copilot to implement design patterns.

In this section, we will see how to implement the following design patterns.

1. Singleton
2. Abstract Factory
3. Builder Pattern

Activity 1 – Singleton

1. Create a new file named PetService.java in 'patterns' folder.



2. Use following code to implement PetService class as a singleton.

```
Only one instance of PetService class must be created, suggest changes to convert this class into a singleton
```

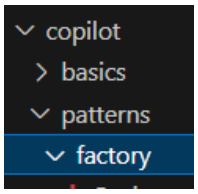
3. Paste the generated code, run and check if it works as expected.

Activity 2 – Abstract Factory

1. Create a new file named PetService.java in 'patterns' folder.
2. Use below prompt to create necessary classes.

```
define pet class with abstract factory pattern
```

3. Based on the code provided by the above prompt, create appropriate class files in the following folder:



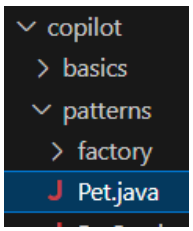
4. Copy and paste the generated code into the appropriate files.
5. Create a file named 'FactoryDemo.java' in 'factory' folder and include the code generated by the below prompt.

```
include a main method in FactoryDemo.java to demonstrate using the pet factory classes
```

6. Run the main() method and check if the abstract factory pattern works as expected.

Activity 3 – Builder Pattern for Pet

1. Create file



2. Run the below prompt

```
create a class to represent a Pet
```

3. This is the generated code

```java
public class Pet {
    private String name;
    private String type;

    public Pet(String name, String typ
        this.name = name;
        this.type = type;
    }

    public String getName() {
        return name;
    }

    public String getType() {
        return type;
    }

    // other methods
}
```

4. Copy and paste the generate code into Pet.java.

5. Use the below prompt to convert the code in Builder pattern.

```
convert pet class to use builder pattern
```

6. Use the below prompt to generate main() method that uses the Pet class with builder pattern.

```
include main method that demonstrates how to use
the pet class with builder pattern
```

Time Required

45 Minutes

24. Comment on existing methods

1. The below prompt generates documentation comments for the existing listOwners() function.

```
generate comments for listOwners function
```

2. This is the code generated

```
/**
 * This function is used to list all the owners.
 * It retrieves all the owner records from the databa
 *
 * @return List<Owner> This returns a list of all the
 */
public List<Owner> listOwners() {
    // function implementation
}
```

3. Following prompt will generate comments explaining the steps.

```
also generate comments for the operations performed within the function
```

```
public List<Owner> listOwners() {
    // Create an empty list to store the owners
    List<Owner> owners = new ArrayList<>();

    // Retrieve all the owner records from the database
    // This is done by calling the findAll() method of t
    Iterable<Owner> ownerRecords = ownerRepository.findA

    // Iterate over the owner records and add each one t
    for (Owner owner : ownerRecords) {
        owners.add(owner);
    }

    // Return the list of owners
    return owners;
}
```

Time Required

15 Minutes

Chapter – 13

GitHub Copilot

for

Generating Complex Join Queries

25. Generating complex join queries.

1. A prompt like this can be provided to generate a complex query.

```
write sql query to get all the visits happened on a
specific date, it should display the date, owner
name, the pet name and the reason for the visit
```

2. Corrections were required in the actual column names, which are provided in the below prompt.

```
visits table has visit date column as visit_date, the
owners name is defined in first_name and last_name
columns
```

3. Pet type id was missing, which can be provided in the below prompt.

```
also include the pet type name using type_id
column in pet table
```

4. This is the final query generated.

```
SELECT v.visit_date,
        CONCAT(o.first_name, ' ', o.last_name) AS owner_name,
        p.name AS pet_name,
        t.name AS pet_type,
        v.description AS visit_reason
FROM visits v
JOIN pets p ON v.pet_id = p.id
JOIN owners o ON p.owner_id = o.id
JOIN types t ON p.type_id = t.id
WHERE v.visit_date = '2010-03-04';
```

5. This is the output of the above query.

visit_date	owner_name	pet_name	pet_type	visit_reason
2010-03-04	Jean Coleman	Samantha	cat	rabies shot

Time Required

30 Minutes

Chapter – 14

GitHub Copilot for Generating CSS Style

26. Generate CSS style definitions

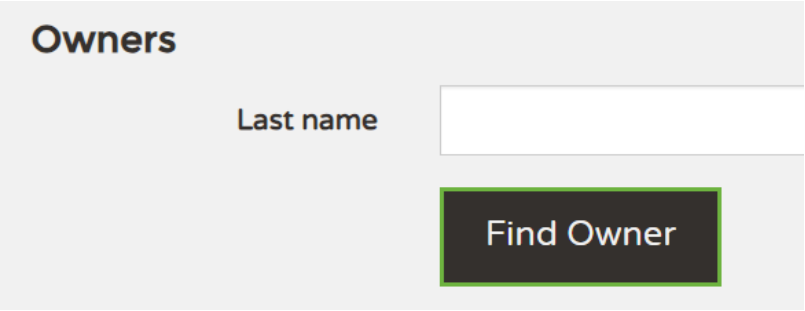
Let us say that we want to display the buttons in the pet clinic app with slightly larger size buttons. Then we can use the below prompt.

```
generate styles for btn-default class to have a slightly larger button with slightly larger font size
```

The above prompt generates the below style definition.

```
.btn-default {
    font-size: 1.2em; /* Increase font size */
    padding: 10px 20px; /* Increase button size */
}
```

Inclusion of the above style definition in styles.css of the pet clinic angular app will show the buttons in slightly larger size.



Time Required

15 Minutes

Chapter – 15

Migrating from Angular to React

27. Migrating from Angular to React

Migrating from Angular to React involves several steps. Below is a detailed guide on how to approach this migration process.

Migration is required only for the below specified requirements. Adjust the GitHub Copilot prompts provided within these requirements to align with the migration process.

10.	Including email for Owner in frontend
12.	Build CRUD – Treatment Type Entity and Repository definition
18.	Build CRUD – Preparing the angular app
19.	Build CRUD – Defining the service for treatment type
20.	Build CRUD – Implement component for listing treatment types
21.	Build CRUD – Implementing Add Treatment type

Step 1: Understand the Existing Application

1. Codebase Analysis
   - Identify structure of Angular app
   - Components, services, modules, routing
2. Dependencies
   - List third-party dependencies

Step 2: Set Up React Environment

1. Create a New React Project
   - Use create-react-app or preferred method
2. Install Dependencies

Step 3: Component Mapping

1. Identify Components
   - List Angular components
   - Map to React components
2. Recreate Components
   - Copy logic, templates, styles

Step 4: Routing

1. Configure React Router
   - Set up React Router for navigation
2. Navigation
   - Adjust navigation components.

Step 5: State Management

1. Identify State Management
   - Services, NgRx, or other
2. Choose React State Management

Step 6: Styling

1. Stylesheets
   - Convert Angular styles to React styles
2. CSS Frameworks

Step 7: API Calls

1. HTTP Requests

- Replace Angular HTTP services

==========End of the document==========