GitHub Copilot Interface Setup Guide 🚀

This guide provides the foundational classes and setup instructions to ensure a smooth learning experience with the GitHub Copilot Interface and Basic Usage exercises.

📋 Pre-Session Setup

Step 1: Create Base Entity Classes

First, let's create the foundational entity classes that will be referenced and extended during the exercises.

User Entity (Enhanced)

```
package com.taskmanager.app.entity;

import javax.persistence.*;
import java.time.LocalDateTime;

@Entity
@Table(name = "users")
public class User {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @Column(nullable = false)
    private String firstName;

    @Column(nullable = false)
    private String lastName;

    @Column(unique = true, nullable = false)
    private String email;

    @Column(nullable = false)
    private String password;

    @Column(name = "created_date")
    private LocalDateTime createdDate;
```

```
    @Column(name = "is_active")
    private boolean active = true;

    // Constructors
    public User() {}

    public User(String firstName, String lastName, String email) {
        this.firstName = firstName;
        this.lastName = lastName;
        this.email = email;
        this.createdDate = LocalDateTime.now();
    }

    // Getters and setters will be generated during exercises
    public Long getId() { return id; }
    public void setId(Long id) { this.id = id; }

    public String getFirstName() { return firstName; }
    public void setFirstName(String firstName) { this.firstName =
firstName; }

    public String getLastName() { return lastName; }
    public void setLastName(String lastName) { this.lastName = lastName; }

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

    public String getPassword() { return password; }
    public void setPassword(String password) { this.password = password; }

    public LocalDateTime getCreatedDate() { return createdDate; }
    public void setCreatedDate(LocalDateTime createdDate)
{ this.createdDate = createdDate; }

    public boolean isActive() { return active; }
    public void setActive(boolean active) { this.active = active; }
}
```

Task Entity (Base Template)

```
package com.taskmanager.app.entity;

import javax.persistence.*;
import java.time.LocalDateTime;

@Entity
@Table(name = "tasks")
public class Task {
```

```
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @Column(nullable = false)
    private String title;

    @Column(length = 1000)
    private String description;

    @Enumerated(EnumType.STRING)
    private TaskStatus status = TaskStatus.PENDING;

    @ManyToOne(fetch = FetchType.LAZY)
    @JoinColumn(name = "assignee_id")
    private User assignee;

    @Column(name = "created_date")
    private LocalDateTime createdDate;

    @Column(name = "due_date")
    private LocalDateTime dueDate;

    // Basic constructor
    public Task() {
        this.createdDate = LocalDateTime.now();
    }

    // Add getters/setters during exercises
    public Long getId() { return id; }
    public String getTitle() { return title; }
    public void setTitle(String title) { this.title = title; }
    public String getDescription() { return description; }
    public void setDescription(String description) { this.description =
description; }
}
```

TaskStatus Enum

```
package com.taskmanager.app.entity;

public enum TaskStatus {
    PENDING,
    IN_PROGRESS,
    COMPLETED,
```

```
    CANCELLED
}
```

Step 2: Create Repository Interfaces

UserRepository

```
package com.taskmanager.app.repository;

import com.taskmanager.app.entity.User;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.stereotype.Repository;
import java.util.Optional;

@Repository
public interface UserRepository extends JpaRepository<User, Long> {

    Optional<User> findByEmail(String email);

    // Add more methods during exercises using Copilot
}
```

TaskRepository

```
package com.taskmanager.app.repository;

import com.taskmanager.app.entity.Task;
import com.taskmanager.app.entity.TaskStatus;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.stereotype.Repository;
import java.util.List;

@Repository
public interface TaskRepository extends JpaRepository<Task, Long> {

    List<Task> findByStatus(TaskStatus status);

    // Expand this during exercises
}
```

Step 3: Create Service Layer Foundations

EmailService (Stub for Exercises)

```
package com.taskmanager.app.service;

import org.springframework.stereotype.Service;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

@Service
public class EmailService {

    private static final Logger logger =
LoggerFactory.getLogger(EmailService.class);

    public void sendWelcomeEmail(String email) {
        // Stub implementation for training
        logger.info("Welcome email would be sent to: {}", email);
    }

    public void sendTaskAssignmentEmail(String email, String taskTitle) {
        logger.info("Task assignment email for '{}' would be sent to: {}",
taskTitle, email);
    }

    // Expand this during exercises
}
```

PasswordEncoder (Simple Implementation for Training)

```
package com.taskmanager.app.service;

import org.springframework.stereotype.Component;
import java.util.Base64;

@Component
public class PasswordEncoder {

    /**
     * Simple encoding for training purposes only
     * In production, use BCryptPasswordEncoder or similar
     */
    public String encode(String rawPassword) {
        return Base64.getEncoder().encodeToString(rawPassword.getBytes());
    }
```

```
    public boolean matches(String rawPassword, String encodedPassword) {
        return encode(rawPassword).equals(encodedPassword);
    }
}
```

Step 4: Create Exception Classes

Custom Exceptions

```
package com.taskmanager.app.exception;

public class UserNotFoundException extends RuntimeException {
    public UserNotFoundException(String message) {
        super(message);
    }
}
```

```
package com.taskmanager.app.exception;

public class UserAlreadyExistsException extends RuntimeException {
    public UserAlreadyExistsException(String message) {
        super(message);
    }
}
```

```
package com.taskmanager.app.exception;

public class ValidationException extends RuntimeException {
    public ValidationException(String message) {
        super(message);
    }
}
```

Step 5: Create DTO Classes

CreateUserRequest DTO

```
package com.taskmanager.app.dto;

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

```
import javax.validation.constraints.NotBlank;
import javax.validation.constraints.Size;

public class CreateUserRequest {

    @NotBlank(message = "First name is required")
    private String firstName;

    @NotBlank(message = "Last name is required")
    private String lastName;

    @Email(message = "Valid email is required")
    @NotBlank(message = "Email is required")
    private String email;

    @Size(min = 6, message = "Password must be at least 6 characters")
    private String password;

    // Constructors
    public CreateUserRequest() {}

    public CreateUserRequest(String firstName, String lastName, String
email, String password) {
        this.firstName = firstName;
        this.lastName = lastName;
        this.email = email;
        this.password = password;
    }

    // Getters and setters
    public String getFirstName() { return firstName; }
    public void setFirstName(String firstName) { this.firstName =
firstName; }

    public String getLastName() { return lastName; }
    public void setLastName(String lastName) { this.lastName = lastName; }

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

    public String getPassword() { return password; }
    public void setPassword(String password) { this.password = password; }
}
```

🎯 Session Approach

Option 1: Complete Pre-Setup (Recommended)

When to Use: When you want to focus purely on Copilot interface learning without setup distractions.

Setup: Create all the above classes before the session starts.

Advantages:

- Immediately start with meaningful exercises
- No setup time wasted during training
- Copilot has rich context from existing classes
- Focus remains on learning Copilot features

Experience:

```
// Immediately start with exercises like:
public class UserService {

    private final UserRepository userRepository; // Already exists!
    private final EmailService emailService;     // Already exists!

    // Type this comment and see rich suggestions
    // Create a method to register a new user

}
```

Option 2: Guided Creation During Session

When to Use: When you want to demonstrate how Copilot helps with class creation from scratch.

Setup: Create only minimal stubs (empty classes).

Experience:

```
// Start with empty class
public class User {
    // Type: "Create JPA entity fields for user management"
    // Copilot suggests based on context and patterns
}
```

Option 3: Hybrid Approach (Best of Both Worlds)

Setup: Create core entities and repositories, but leave services empty.

Advantages:

- Rich context from entities for better suggestions
- Still experience class creation
- Balanced learning experience

Recommended Structure:

```
✅ Create: User.java (complete)
✅ Create: Task.java (complete)
✅ Create: TaskStatus.java (enum)
✅ Create: UserRepository.java (interface with basic methods)
✅ Create: TaskRepository.java (interface with basic methods)
🔲 Create: UserService.java (empty - fill during exercises)
🔲 Create: TaskService.java (empty - fill during exercises)
🔲 Create: EmailService.java (empty - create methods)
🔲 Create: PasswordEncoder.java (empty - implement)
```

📝 Updated Exercise Instructions

Pre-Session Checklist

- [ ] All base entity classes created and imported
- [ ] Repository interfaces created with basic methods
- [ ] Exception classes created
- [ ] DTO classes created for requests
- [ ] Maven dependencies verified (JPA, Validation)
- [ ] Project compiles and runs successfully
- [ ] VS Code with Copilot extension installed and verified

Verification Steps

Before starting exercises, verify:

```
// This should compile without errors
import com.taskmanager.app.entity.User;
```

```
import com.taskmanager.app.repository.UserRepository;
import com.taskmanager.app.service.EmailService;

public class TestSetup {
    public static void main(String[] args) {
        System.out.println("Setup verification successful!");
        // If this compiles, foundation classes are ready
    }
}
```

🚀 Enhanced Exercise Experience

With Proper Foundation

Now you get immediate, high-quality suggestions:

```
public class UserService {

    private final UserRepository userRepository;    // ✅ Exists
    private final PasswordEncoder passwordEncoder; // ✅ Exists
    private final EmailService emailService;       // ✅ Exists

    // Type: "Create constructor" - Copilot suggests perfect constructor
injection
    // Type: "Create method to register user" - Rich, contextual
suggestions
    // Type: "Add validation" - Suggestions include proper exception
handling
}
```

Quality of Suggestions Improves Dramatically

Before (without foundation):

```
// Poor suggestion due to lack of context
public void processUserRegistration(String name) {
    System.out.println("User: " + name);
}
```

After (with foundation):

```
// Rich suggestion leveraging existing context
public User processUserRegistration(CreateUserRequest request) {
    // Validate request
    if (userRepository.findByEmail(request.getEmail()).isPresent()) {
        throw new UserAlreadyExistsException("User already exists");
    }

    // Create and save user
    User user = new User(request.getFirstName(), request.getLastName(),
request.getEmail());
    user.setPassword(passwordEncoder.encode(request.getPassword()));

    User savedUser = userRepository.save(user);
    emailService.sendWelcomeEmail(savedUser.getEmail());

    return savedUser;
}
```

This approach ensures an excellent learning experience with meaningful, contextual Copilot suggestions from the very first exercise!