Analyzing Web Frameworks through an Object-Oriented Lens: A Comprehensive research on Django and Spring Boot

Jayant Duneja Jayant.Duneja@colorado.edu

Tilak Singh Tilak.Singh@colorado.edu

Overview

1. **Introduction :** The research aims to examine two popular web frameworks, Django (Python) and Spring Boot (Java), from an object-oriented perspective, shedding light on their design, architecture, and adherence to object-oriented principles.
2. **Object-Oriented Principles :** The study delves into how Django and Spring Boot utilize key object-oriented principles such as inheritance, encapsulation, and polymorphism in the context of web application development.
3. **Design Patterns :** The research explores the design patterns employed by each framework, highlighting Django's use of the Model-View-Controller (MVC) pattern and Spring Boot's combination of MVC and Dependency Injection patterns.
4. **Modularity and Extensibility:** It investigates how both frameworks encourage modularity and the creation of reusable components. Additionally, it explores how developers can customize and extend these components through inheritance and polymorphism.
5. **Practical Considerations:** The research also factors in practical considerations, including the communities and ecosystems surrounding Django and Spring Boot, performance, scalability, and the learning curve, which can influence the choice of framework for specific development projects.

Introduction to Spring Boot [1]

- Spring Boot simplifies the development of Java applications by providing opinionated defaults, auto-configuration, and a range of pre-built templates. This reduces the amount of boilerplate code, allowing developers to focus more on application logic rather than infrastructure setup.
- Spring Boot includes support for embedded web servers, enabling developers to package their applications as standalone JAR files. This eliminates the need for external web server configurations and facilitates easy deployment, making it well-suited for building microservices and modern, cloud-native applications.

Introduction to Django [2]

- Django is a high-level Python web framework that encourages rapid development and clean, pragmatic design. It follows the "Don't Repeat Yourself" (DRY) and "Convention over Configuration" principles, providing a set of conventions and best practices to streamline the development process. Django's design promotes modular and reusable code, making it an efficient choice for building web applications.

- Django follows a "batteries-included" philosophy, offering a rich set of built-in features such as an Object-Relational Mapping (ORM) system for database interaction, a templating engine for dynamic content rendering, and an admin interface for easy content management.

- This comprehensive set of tools reduces the need for developers to integrate third-party libraries for common tasks, allowing them to focus on building application-specific features.

Object Oriented Principles in Web Development [3]

1. ***Definition:*** OOD is a paradigm that models software systems as a collection of objects that interact with each other through well-defined interfaces.

2. For Web Development, Object-Oriented Design (OOD) can help you achieve systems that can handle various user scenarios, requests, data, and devices by providing advantages like:
    - Modularity: Divide systems into smaller components for independent development and deployment.
    - Reusability: Enable code to be reused across projects or system components.
    - Extensibility: Adapt easily to new requirements or features.
    - Maintainability: Ensure code is easy to maintain and update.
    - Security: Use encapsulation to protect data and functionality.

3. Make use of Design Principles like MVC, SOLID, YAGNI, DRY, KISS.

4. Examples of OOAD Principles in Web Development:
    - ***WordPress:*** Uses OOD to create a modular and extensible content management system.
    - ***Django:*** A web framework that employs OOD to handle common tasks like authentication and database access.
    - ***React:*** A web library that uses OOD to create a component-based system for UI elements.
    - ***Spring Boot:*** An open-source Java-based framework that leverages OOD principles to simplify the development of production-ready, stand-alone, and web-based applications.

Abstraction in Spring Boot [4]

One of the most important examples of where Spring Boot uses abstraction is in the context of **Database connectivity**. This is done by 2 ways:

1. **Abstraction of Configuration:**
   In Spring Boot, abstraction is used to encapsulate the configuration details required for database connectivity. Spring Boot allows you to define database-related configuration settings in a configuration file. These configuration properties include database URL, username, password, and other settings. For example, we can see an example of one such configuration file here:

```
app.datasource.url=jdbc:mysql://localhost/test
app.datasource.username=dbuser
app.datasource.password=dbpass
app.datasource.pool-size=30
```

Abstraction in Spring Boot

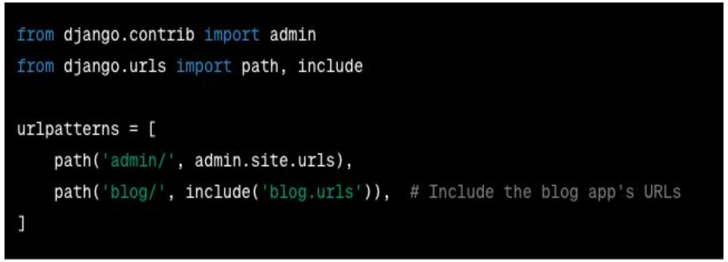
2. **DataSource Abstraction:**

- Spring Boot abstracts the creation and management of the database connection pool through a DataSource. Spring Boot defines a DataSource interface, which abstracts the database connection. You can inject a DataSource into your application components without knowing the specific implementation details like the type of Datasource connection you have created.
- Connection Pool Management: Spring Boot manages the connection pool under the hood, including details like connection pooling, connection acquisition, and releasing. Developers can work with the DataSource without concerning themselves with these low-level operations.
- In the example shown below, we do not have to worry about internal details like connection pooling or which datasource is being created. Spring does this for us.

```java
@Bean
@ConfigurationProperties("app.datasource")
public DataSource dataSource() {
        return DataSourceBuilder.create().build();
}
```

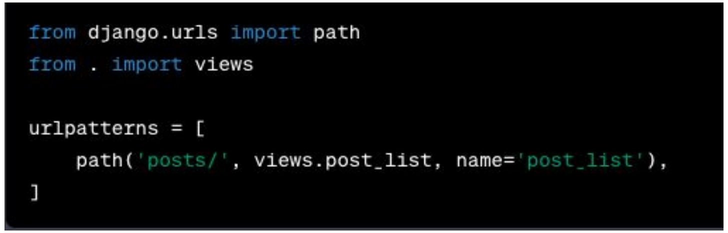
Abstraction in Django

- Django achieves abstraction through various components, enabling developers to focus on application logic rather than low-level database operations. Two key aspects of abstraction in Django are:
  - **Database abstraction:** Django provides a high-level database abstraction layer through its Object-Relational Mapping (ORM) system. This abstraction simplifies database interactions, allowing developers to work with objects and classes instead of raw SQL queries.
  - **URL routing and view abstraction:** Django abstracts URL routing and view handling through its URL patterns and view functions or classes. This abstraction simplifies the management of routes and views, ensuring clean separation of concerns



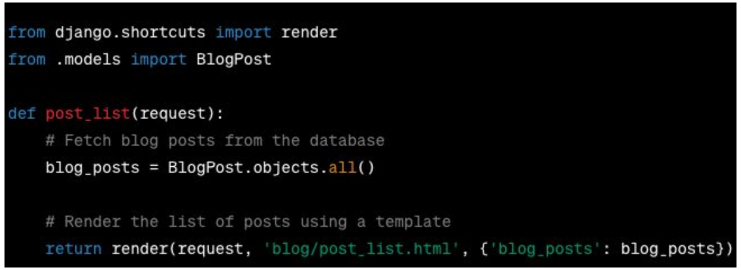
```
from django.contrib import admin
from django.urls import path, include

urlpatterns = [
    path('admin/', admin.site.urls),
    path('blog/', include('blog.urls')),  # Include the blog app's URLs
]
```

Figure 1



```
from django.urls import path
from . import views

urlpatterns = [
    path('posts/', views.post_list, name='post_list'),
]
```

Figure 2



```
from django.shortcuts import render
from .models import BlogPost

def post_list(request):
    # Fetch blog posts from the database
    blog_posts = BlogPost.objects.all()

    # Render the list of posts using a template
    return render(request, 'blog/post_list.html', {'blog_posts': blog_posts})
```

Figure 3

For an example of URL Routing and view abstraction in Django: [5]

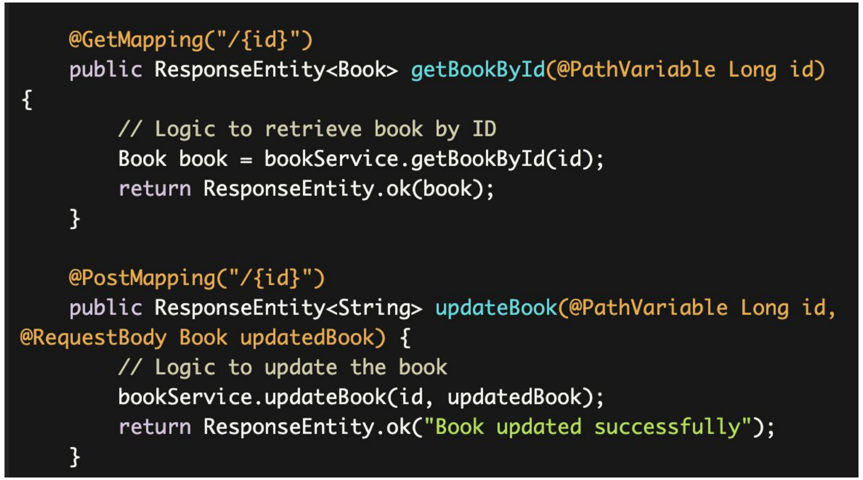
- In the code block 1, we define a URL pattern in your Django project urls.py file. This is where you specify the URL route and associate it with a view function or class.
    - In this code block, we've created a URL pattern that maps any URL starting with "blog/" to the blog app URLs.
- In the code block 2, in the blog app’s urls.py, we define the URL patterns for the blog related views.
    - In this pattern, we have defined a pattern for listing all the blog posts using that maps the "posts/" URL to a view function called post_list.
- Finally, in the third figure, we define post_list, which is a view function that fetches all the blog posts from the database and then renders a template called 'blog/post_list.html'. This template can display the list of blog posts.

Polymorphism in Spring Boot [6]

- One example of how we can use polymorphism in Spring Boot is through Handler Mapping and Method Overriding.
- Spring Boot provides us various ways of performing polymorphism using Handlers. We can define different methods and mappings for the same URL, multiple mappings for the same method or different URL's matched to the same method.
- We can define multiple request mappings for a single URL endpoint using different HTTP methods. For example, you can have a GET mapping and a POST mapping for the same endpoint. This allows you to have multiple methods in a controller class with the same endpoint URL but different HTTP methods, providing polymorphic behavior based on the incoming request.

Polymorphism in Spring Boot [7]

- For example, in the code shown on the right, the getBookById() method is annotated with @GetMapping and handles GET requests to the /books/{id} URL pattern.
- The updateBook() method is annotated with @PostMapping and handles POST requests to the same /books/{id} URL pattern.
- This approach enables us in grouping related functionality under the same endpoint URL while using polymorphism.
- This approach is particularly useful when you want to handle different types of requests for the same resource or when you want to provide different actions based on the HTTP method.



```
    @GetMapping("/{id}")
    public ResponseEntity<Book> getBookById(@PathVariable Long id)
{
        // Logic to retrieve book by ID
        Book book = bookService.getBookById(id);
        return ResponseEntity.ok(book);
    }

    @PostMapping("/{id}")
    public ResponseEntity<String> updateBook(@PathVariable Long id,
@RequestBody Book updatedBook) {
        // Logic to update the book
        bookService.updateBook(id, updatedBook);
        return ResponseEntity.ok("Book updated successfully");
    }
```

Polymorphism in Django [8]

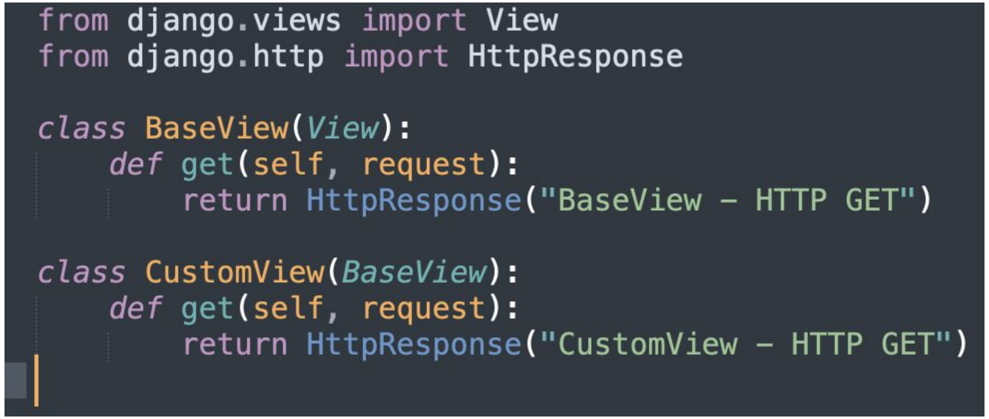
- Polymorphism in Django is realized through class-based views, which are Python classes that can be extended to customize behavior for specific HTTP request methods. The ability to extend and customize class-based views and have Django's URL dispatcher dynamically invoke the appropriate methods based on the HTTP request method is a powerful demonstration of polymorphism in action.
- Polymorphism in Django allows developers to create flexible and modular applications by utilizing Python's object-oriented features. In Django's Object-Relational Mapping (ORM), polymorphism enables the use of inheritance, where you can define a base model class with shared attributes and methods, and then create subclasses that can extend or override those features. This promotes code reuse and simplifies the development of complex data models.
- Additionally, in Django's view system, polymorphism can be leveraged by using class-based views, where different views can inherit from a common base view, enabling the reuse of common behavior and customization when needed.

Polymorphism in Django

Class-Based Views

1. Class-based views in Django allow developers to create extensible and reusable view components. They are defined as Python classes and can be extended to customize behavior for specific HTTP request methods.

2. In the code example, we have two views: **BaseView** and **CustomView**. BaseView defines a *get* method to handle HTTP GET requests, while CustomView extends BaseView and overrides the *get* method to provide a customized response.

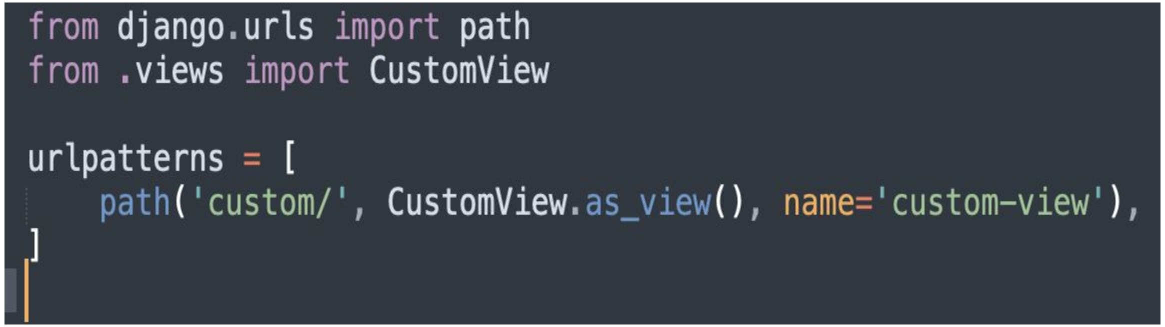


```python
from django.views import View
from django.http import HttpResponse

class BaseView(View):
    def get(self, request):
        return HttpResponse("BaseView - HTTP GET")

class CustomView(BaseView):
    def get(self, request):
        return HttpResponse("CustomView - HTTP GET")
```

Polymorphism in Django

Dynamic Method Dispatch

- In the code example, we define a URL pattern that associates the /custom/ URL with the CustomView class using CustomView.as_view(). When a GET request is made to this URL, the get method of CustomView is executed, showcasing dynamic method dispatch in action.

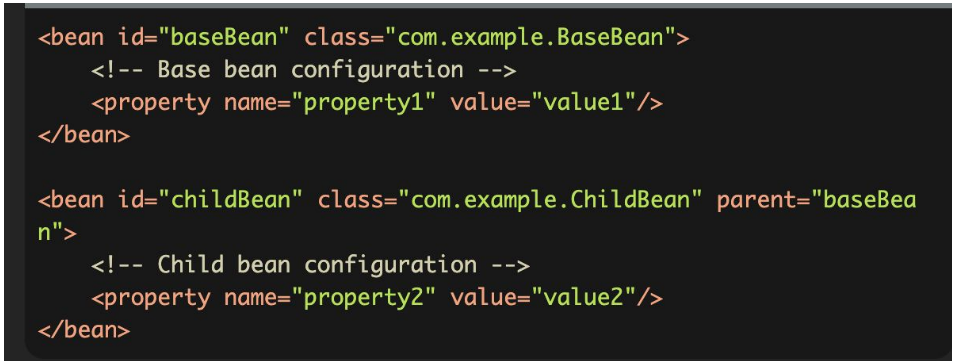


```python
from django.urls import path
from .views import CustomView

urlpatterns = [
    path('custom/', CustomView.as_view(), name='custom-view'),
]
```

Inheritance in Spring Boot [9]

- A bean in Spring Boot is a managed and configurable object that is instantiated and managed by the Spring framework, representing a reusable component within a Spring application. You can define beans and their configurations using either XML-based configuration or Java-based configuration. In both cases, inheritance plays a significant role, allowing you to create a hierarchy of bean definitions and configurations.

- For the case when we define using XML notation, this is an example of bean inheritance:

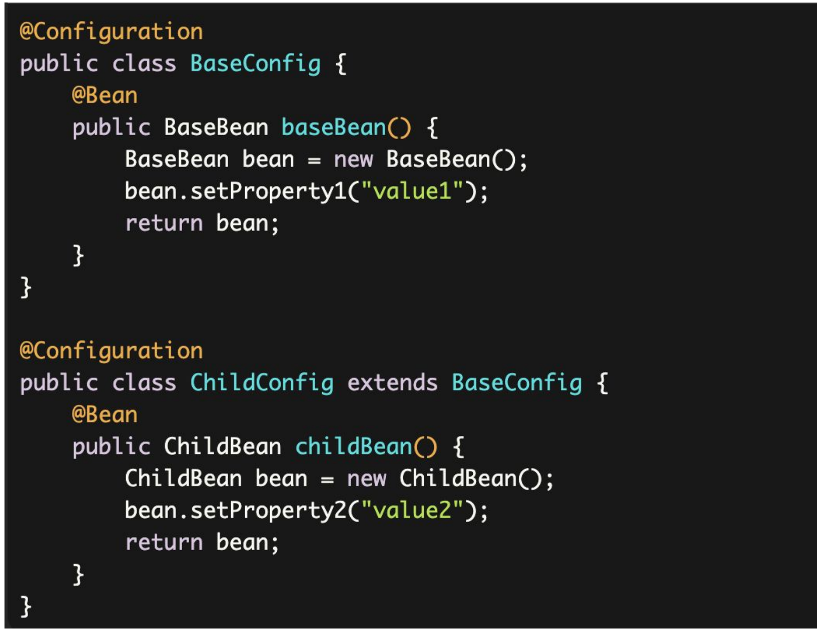


```
<bean id="baseBean" class="com.example.BaseBean">
    <!-- Base bean configuration -->
    <property name="property1" value="value1"/>
</bean>

<bean id="childBean" class="com.example.ChildBean" parent="baseBea
n">
    <!-- Child bean configuration -->
    <property name="property2" value="value2"/>
</bean>
```

Inheritance in Spring Boot

- For the case when we define Beans using Java-based configuration, this is an example of Bean inheritance:

- Through these methods, we get the benefits of Inheritance like code reuse,modularity etc. while defining Spring Beans.

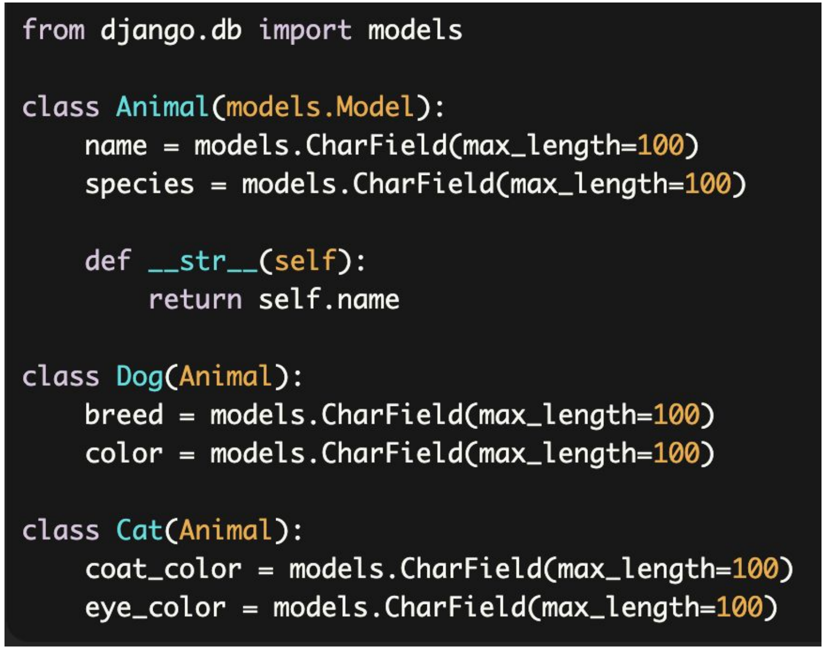


```
@Configuration
public class BaseConfig {
    @Bean
    public BaseBean baseBean() {
        BaseBean bean = new BaseBean();
        bean.setProperty1("value1");
        return bean;
    }
}

@Configuration
public class ChildConfig extends BaseConfig {
    @Bean
    public ChildBean childBean() {
        ChildBean bean = new ChildBean();
        bean.setProperty2("value2");
        return bean;
    }
}
```

Inheritance in Django [10]

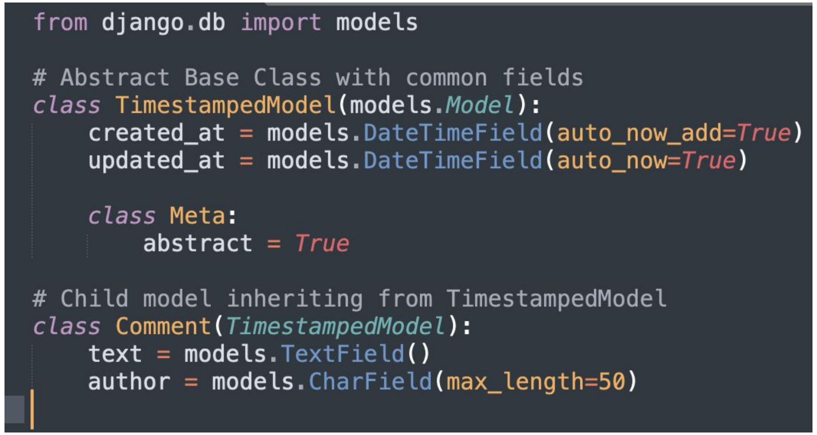
- In Django, a "Model" is a concept related to the Object-Relational Mapping (ORM) system. A model in Django represents a database table and its associated fields, defining the structure and behavior of the data stored in the database.

- Similar to how we use inheritance for Bean definitions in Java, we can use inheritance for model definitions in Django. This will help us in re-using code and implementing a hierarchy. An example of this is shown:



```python
from django.db import models

class Animal(models.Model):
    name = models.CharField(max_length=100)
    species = models.CharField(max_length=100)

    def __str__(self):
        return self.name

class Dog(Animal):
    breed = models.CharField(max_length=100)
    color = models.CharField(max_length=100)

class Cat(Animal):
    coat_color = models.CharField(max_length=100)
    eye_color = models.CharField(max_length=100)
```

Inheritance in Django

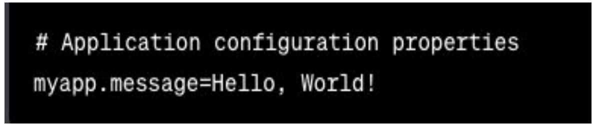


```
from django.db import models

# Abstract Base Class with common fields
class TimestampedModel(models.Model):
    created_at = models.DateTimeField(auto_now_add=True)
    updated_at = models.DateTimeField(auto_now=True)

    class Meta:
        abstract = True

# Child model inheriting from TimestampedModel
class Comment(TimestampedModel):
    text = models.TextField()
    author = models.CharField(max_length=50)
```

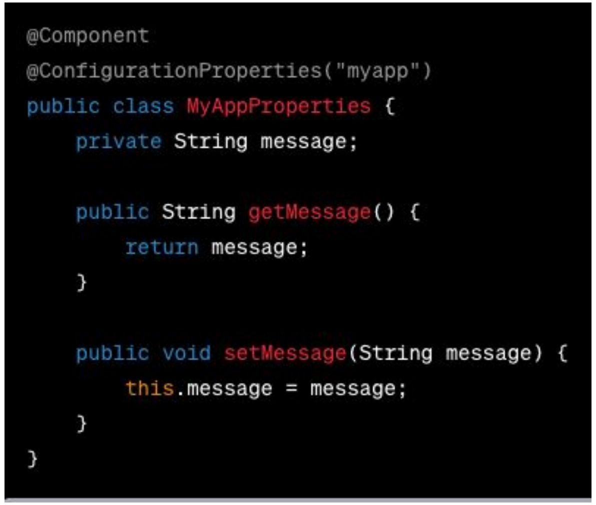
- In this code, Django leverages the concept of inheritance to create a relationship between models.
- The `TimestampedModel` serves as an abstract base class with common fields, including `created_at` and `updated_at`, which are used to track creation and modification times.
- The child model, `Comment`, inherits from `TimestampedModel`, which means it inherits the fields and behaviors of the parent class while adding its own fields like `text` and `author`.
- This approach promotes code reuse and maintains a structured database schema.

Encapsulation in Spring Boot [11]

- A simplified example within Spring Boot's internal workings can be found in how Spring Boot handles properties and configuration.
- SpringBoot uses property files configure application properties. The details of property loading, resolution, and configuration are encapsulated within Spring Boot.
- In the code example given on the right, we have defined the properties for a specific myapp, and we can use this configuration simply with the help of Annotation @ConfigurationProperties, as shown in the example. We have specified ‘myapp’ in the annotation, which will pull all the properties of that particular app.



```
# Application configuration properties
myapp.message=Hello, World!
```

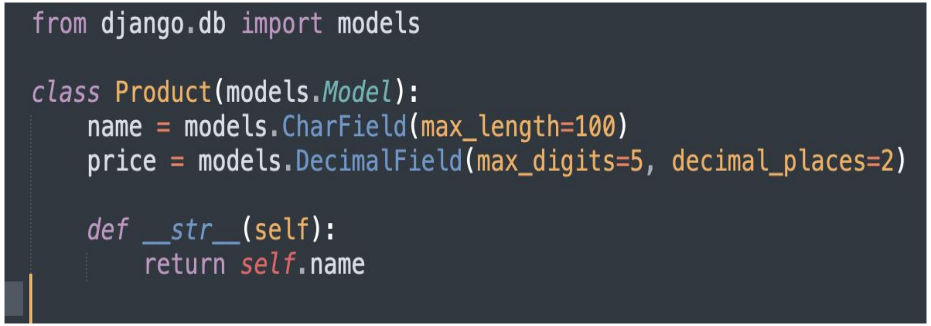


```
@Component
@ConfigurationProperties("myapp")
public class MyAppProperties {
    private String message;

    public String getMessage() {
        return message;
    }

    public void setMessage(String message) {
        this.message = message;
    }
}
```

Encapsulation in Django [12]

- Django, as a high-level web framework, achieves encapsulation primarily through its Object-Relational Mapping (ORM) system, which abstracts database interactions.
- Django models, which represent database tables, encapsulate data attributes and behaviors. The data attributes are defined as model fields (e.g., CharField, DecimalField).
- Django model methods encapsulate database operations, ensuring data integrity and security.

In this example, the Product model encapsulates the data attributes name and price, and the __str__ method encapsulates the logic for representing the object as a string



```python
from django.db import models

class Product(models.Model):
    name = models.CharField(max_length=100)
    price = models.DecimalField(max_digits=5, decimal_places=2)

    def __str__(self):
        return self.name
```

Comparative Analysis of OOP Principles and Design Patterns[13]

Spring Boot

- MVC
- Inversion of Control
- Data Access Object (DAO)
- Repository pattern

Django

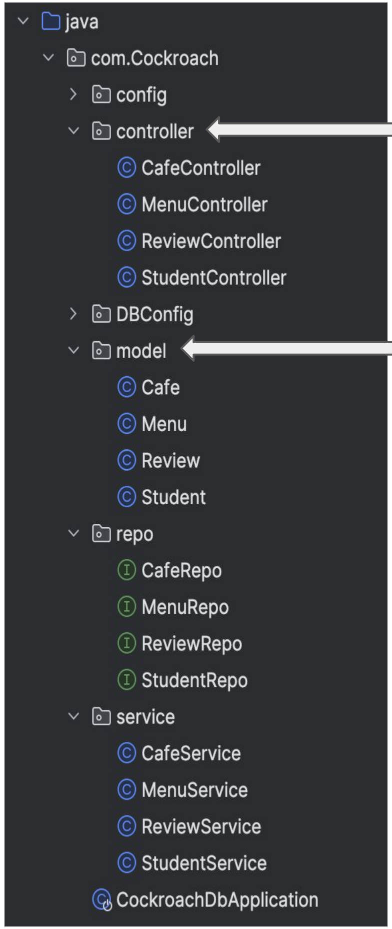
- MVC
- Command Pattern
- Observer Pattern
- Template Pattern

We will be exploring these design patterns in Spring Boot and Django in detail.

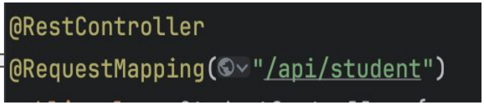
Design Patterns in Spring Boot:

Pattern 1: MVC

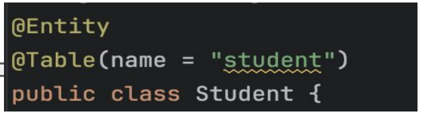
- Model
  - The Model represents the application's data and business logic.
  - In Spring MVC, the model is typically composed of Java objects (POJOs) or entities that are used to carry data between the Controller and the View.
  - Spring Boot allows you to use JPA (Java Persistence API) for data access and Hibernate as the default implementation.
- View
  - The View is responsible for presenting the data to the user and receiving user input.
  - In Spring MVC, the view is often implemented using technologies like JSP (JavaServer Pages), Thymeleaf, or FreeMarker.
  - Thymeleaf is a popular template engine supported by Spring Boot that allows you to create dynamic and easily maintainable HTML templates.
- Controller
  - The Controller handles user input, processes it (possibly involving the Model), and selects the appropriate view to render.
  - In Spring MVC, controllers are typically implemented as Java classes annotated with `**@Controller**.`
  - Request mappings define which method in the controller should handle a specific URL pattern eg, `**@RequestMapping**`, `**@GetMapping**`.



```
java
  com.Cockroach
    config
    controller
      CafeController
      MenuController
      ReviewController
      StudentController
    DBConfig
    model
      Cafe
      Menu
      Review
      Student
    repo
      CafeRepo
      MenuRepo
      ReviewRepo
      StudentRepo
    service
      CafeService
      MenuService
      ReviewService
      StudentService
    CockroachDbApplication
```

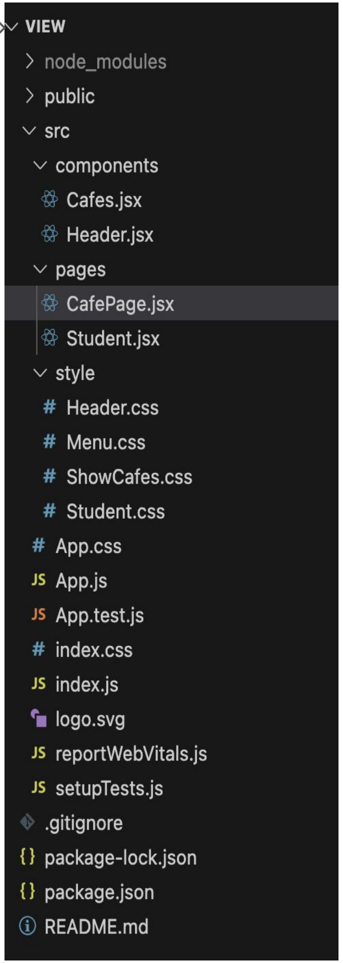


```java
@RestController
@RequestMapping("/api/student")
```



```java
@Entity
@Table(name = "student")
public class Student {
```

React folder
(view)



```
VIEW
  node_modules
  public
  src
    components
      Cafes.jsx
      Header.jsx
    pages
      CafePage.jsx
      Student.jsx
    style
      Header.css
      Menu.css
      ShowCafes.css
      Student.css
    App.css
    App.js
    App.test.js
    index.css
    index.js
    logo.svg
    reportWebVitals.js
    setupTests.js
  .gitignore
  package-lock.json
  package.json
  README.md
```

The Model oversees data and logic, the React View presents data, and the Controller manages user input.

Design Patterns in Spring Boot:

Pattern 2: Inversion of Control (IoC)

IoC is a design pattern that changes the control flow in application components. It establishes a framework to manage dependencies between components, also known as dependency injection.

- Spring Boot implementation
  - Spring Boot utilizes IoC to automatically initialize and connect components during framework execution.
  - The framework ensures a logical order and organization of components, handling their dependencies.
- Dependency Configuration in Spring
  - Spring Boot configures IoC by scanning configuration files and annotations.
  - Components are initialized and stored in the application context, acting as a collection for caching and management.
- Bean Retrieval in IoC:
  - The application context in Spring receives queries for instances, referred to as beans.
  - Beans are retrieved based on criteria like name, supertype, and scope.
- Spring's Closed System Approach:
  - In Spring, the IoC system operates as a closed system.
  - Beans within the application context are only aware of others in the same context.
  - Components are instantiated by Spring only when other Spring components depend on them.

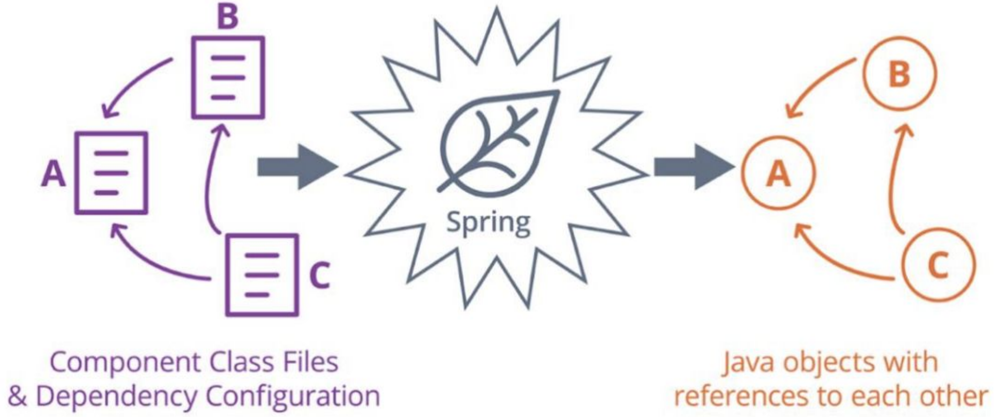


Figure 16. Spring IoC Configuration

Spring Boot IoC Configuration [14]

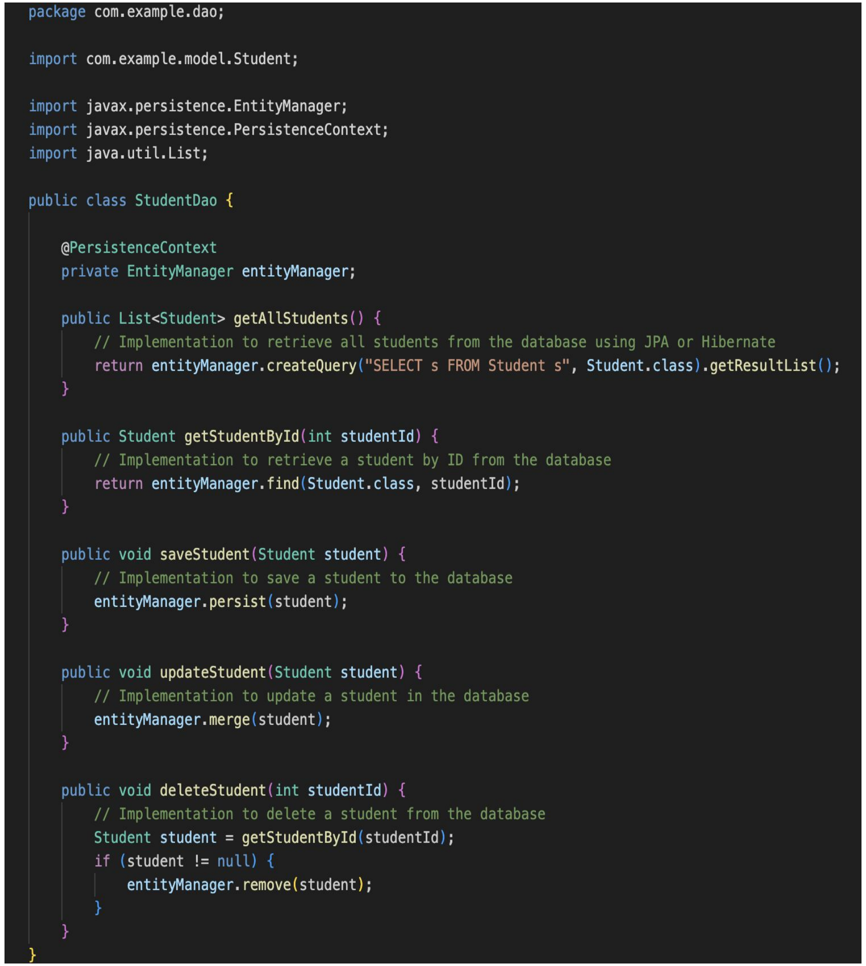
(IoC) is a design principle where the framework controls the flow and lifecycle of components

Design Patterns in Spring Boot:

Pattern 3: Data Access Object (DAO)

DAO (Data Access Object) is an abstraction that provides a simplified interface for interacting with databases, reducing the need for direct and low-level database queries in application code.

- Abstraction of Data Persistence
    - DAO provides a higher-level abstraction for handling data persistence in a table-centric format.
    - This abstraction simplifies how data is managed within the application.
- Elimination of Direct Database Queries
    - DAO eliminates the need for developers to manually construct low-level database queries.
    - It streamlines data operations, providing a more intuitive way to interact with the database.
- Request Abstraction
    - DAO abstracts the requests to access data from the database, simplifying the process for developers.
    - It serves as a bridge between application code and the underlying storage.
- Simplified Data Interaction in Spring Boot
    - In the context of Spring Boot, DAO minimizes the complexity of data operations.
    - Developers can leverage DAO methods, avoiding the intricacies of direct database queries and promoting cleaner code.



```java
package com.example.dao;

import com.example.model.Student;

import javax.persistence.EntityManager;
import javax.persistence.PersistenceContext;
import java.util.List;

public class StudentDao {

    @PersistenceContext
    private EntityManager entityManager;

    public List<Student> getAllStudents() {
        // Implementation to retrieve all students from the database using JPA or Hibernate
        return entityManager.createQuery("SELECT s FROM Student s", Student.class).getResultList();
    }

    public Student getStudentById(int studentId) {
        // Implementation to retrieve a student by ID from the database
        return entityManager.find(Student.class, studentId);
    }

    public void saveStudent(Student student) {
        // Implementation to save a student to the database
        entityManager.persist(student);
    }

    public void updateStudent(Student student) {
        // Implementation to update a student in the database
        entityManager.merge(student);
    }

    public void deleteStudent(int studentId) {
        // Implementation to delete a student from the database
        Student student = getStudentById(studentId);
        if (student != null) {
            entityManager.remove(student);
        }
    }
}
```

`**DAO**` eliminates direct queries, streamlining operations and acting as a bridge for request abstraction

`**StudentDao**` is a simple class handling data access operations.

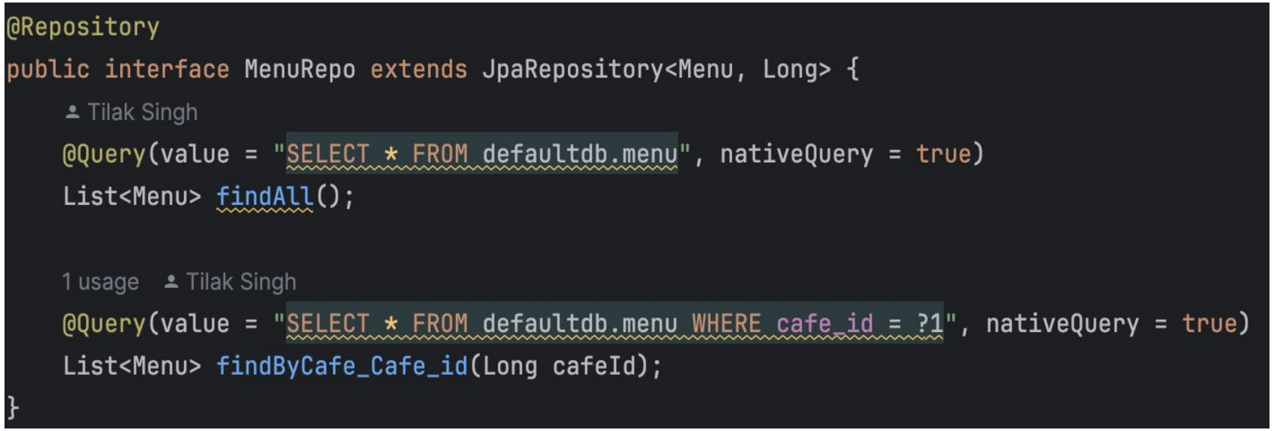
It uses `**EntityManager**` for database interaction (you could use Spring Data JPA repositories as well).

Design Patterns in Spring Boot:

Pattern 4: Repository Pattern

The Repository Pattern is a design pattern that abstracts and centralizes data access logic, providing a unified interface for managing storage, retrieval, and search operations in software applications.

- Abstraction of Data Interactions
  - The repository pattern abstracts away the complexities of storage, retrieval, and search operations, providing a clean and unified interface
  - It encapsulates the interaction with the underlying database, allowing for a more modular and maintainable design.
- Mediation Between Layers
  - Serving as a mediator, the repository bridges the gap between the domain and data mapping layers.
  - It provides a collection-like interface for accessing and managing domain objects, promoting a separation of concerns in the application architecture.
- Implementation of DAO Functionality
  - The repository layer implements the Data Access Object (DAO) functionality in Spring Boot.
  - It handles the communication between the application and the database, abstracting the low-level details of data access.
- Use of Java Persistence API (JPA):
  - Leveraging the Java Persistence API (JPA), the repository simplifies and standardizes the mapping of Java objects to relational database tables.
  - JPA provides a powerful and flexible mechanism for managing entity relationships and performing database operations.



```
@Repository
public interface MenuRepo extends JpaRepository<Menu, Long> {
    Tilak Singh
    @Query(value = "SELECT * FROM defaultdb.menu", nativeQuery = true)
    List<Menu> findAll();

    1 usage   Tilak Singh
    @Query(value = "SELECT * FROM defaultdb.menu WHERE cafe_id = ?1", nativeQuery = true)
    List<Menu> findByCafe_Cafe_id(Long cafeId);
}
```

Repository pattern centralizes data access and the service layer is able to access the data through these repositories. Generally, ORMs in-built functions are used for fetching/manipulating data but here, raw query is written.

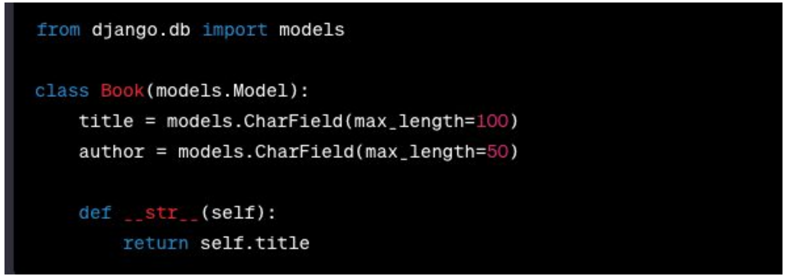
Design Patterns in Django:[15]

Pattern 1: MTV Pattern

- While Django is often associated with the MVC (Model-View-Controller) pattern, it actually follows a slightly different architecture called Model-Template-View (MTV). In this design, Django separates concerns between database interfacing (Model), request processing (View), and final presentation (Template).

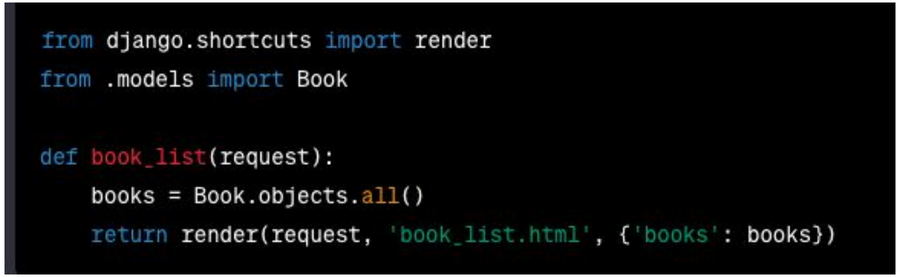
- Django's approach is more practical for web applications, using a pipeline-like framework to process each HTTP request independently. The terminology aligns with MVC to some extent, where Django's Models correspond to MVC's Model, Templates to View, and the framework itself acts as the Controller, handling incoming requests and routing them to the appropriate view function.

Model



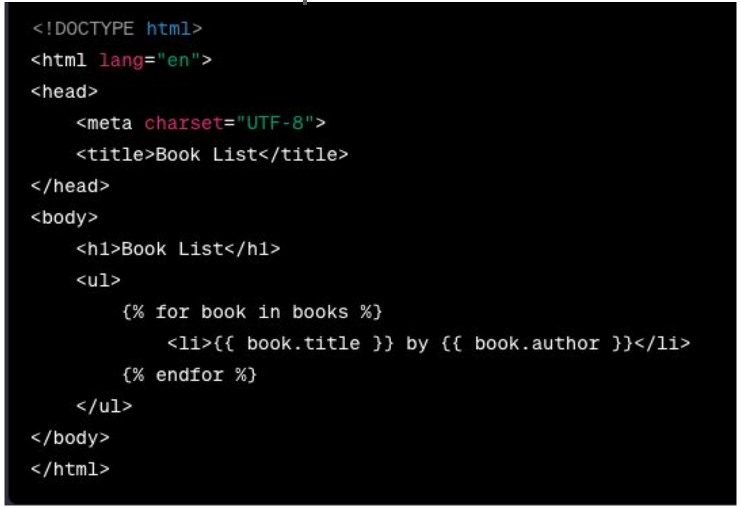
```python
from django.db import models

class Book(models.Model):
    title = models.CharField(max_length=100)
    author = models.CharField(max_length=50)

    def __str__(self):
        return self.title
```

View



```python
from django.shortcuts import render
from .models import Book

def book_list(request):
    books = Book.objects.all()
    return render(request, 'book_list.html', {'books': books})
```

Template



```html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Book List</title>
</head>
<body>
    <h1>Book List</h1>
    <ul>
        {% for book in books %}
            <li>{{ book.title }} by {{ book.author }}</li>
        {% endfor %}
    </ul>
</body>
</html>
```

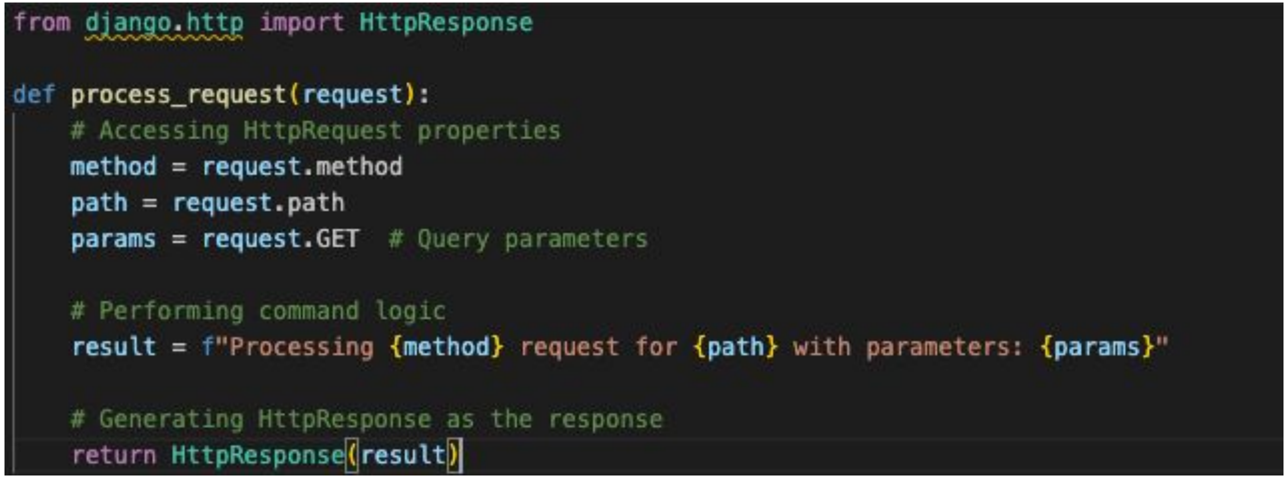
The Book model defines a simple data structure for books with title and author attributes.
The book_list.html template displays a list of books using Django template syntax.
The book_list view function retrieves all books from the database and renders the template with the book data.
This is a simplified example, but it demonstrates the separation of concerns in Django's MTV architecture.

Design Patterns in Django:

Pattern 2: Command Pattern

- In Django, the **HttpRequest** object encapsulates a request in an object, aligning with the principles of the Command pattern.
- Django's approach is more practical for web applications, using a pipeline-like framework to process each HTTP request independently. The terminology aligns with MVC to some extent, where Django's Models correspond to MVC's Model, Templates to View, and the framework itself acts as the Controller, handling incoming requests and routing them to the appropriate view function.
- In the context of the Command pattern:
  - Encapsulation: HttpRequest encapsulates all the details of an HTTP request within a single object.
  - Command Object: HttpRequest can be viewed as a command object that carries all the necessary information for executing a particular request.
  - Request Handling: Django's views, which are essentially command handlers, receive HttpRequest objects as parameters.
  - Flexibility: The use of HttpRequest provides flexibility in handling different types of requests.

Using HttpRequest and HttpResponse in Django



```python
from django.http import HttpResponse

def process_request(request):
    # Accessing HttpRequest properties
    method = request.method
    path = request.path
    params = request.GET  # Query parameters

    # Performing command logic
    result = f"Processing {method} request for {path} with parameters: {params}"

    # Generating HttpResponse as the response
    return HttpResponse(result)
```

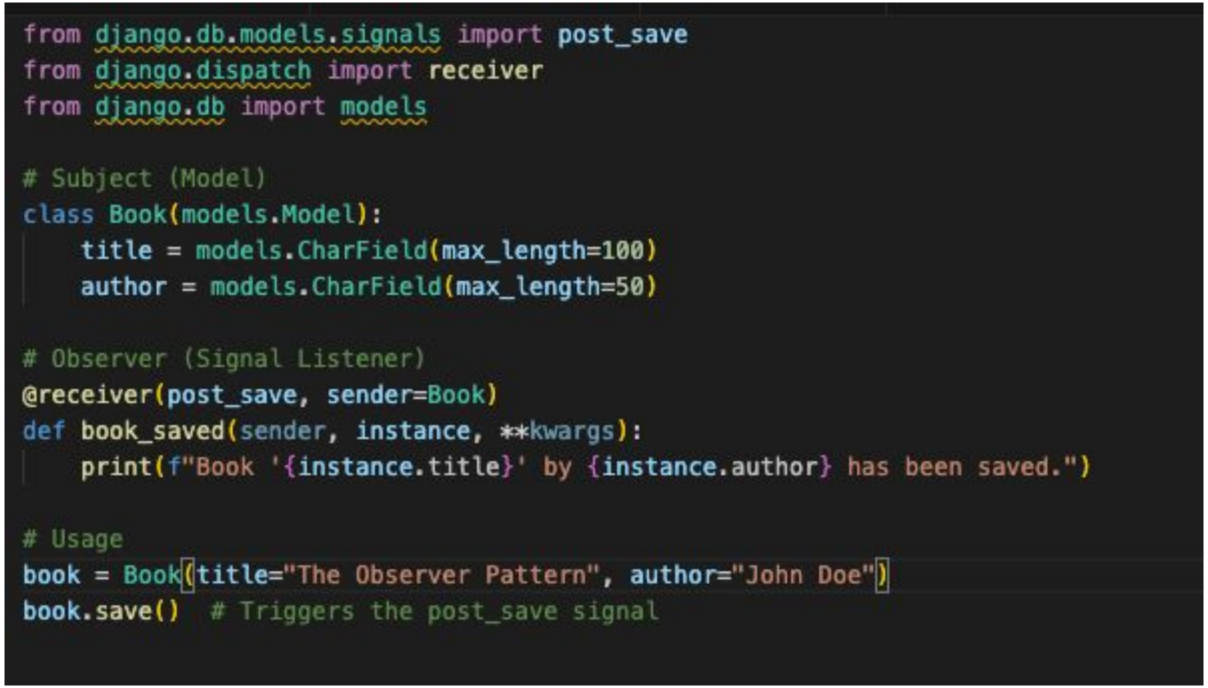
The process_request view function takes an HttpRequest object as a parameter.
It extracts information from the request, such as the HTTP method, path, and query parameters.
The extracted data is used to perform some command logic (processing the request).
The result is then incorporated into an HttpResponse object, forming the response to the client.
This demonstrates how Django's HttpRequest object encapsulates the details of an incoming request, allowing for easy access to information and structured command handling within a view function.

Design Patterns in Django:

<u>Pattern 3: Observer Pattern</u>

- The Observer pattern is a behavioral design pattern where an object, known as the subject, maintains a list of its dependents, called observers, that are notified of any changes in the subject's state. In Django, this pattern is commonly implemented through signals.
- Elaboration:
  - Subject (Signal Sender): In Django, a model or other components can be considered as the subject. When the state of the subject changes (e.g., an instance of the model is saved), signals are emitted.
  - Observers (Signal Listeners): Signals are essentially messages announcing that a certain event has occurred. Other parts of the codebase, such as functions or methods, can register themselves as listeners for specific signals. When the signal is sent, all registered listeners are notified and can take appropriate actions.

Observer Pattern using Signals in Django



```python
from django.db.models.signals import post_save
from django.dispatch import receiver
from django.db import models

# Subject (Model)
class Book(models.Model):
    title = models.CharField(max_length=100)
    author = models.CharField(max_length=50)

# Observer (Signal Listener)
@receiver(post_save, sender=Book)
def book_saved(sender, instance, **kwargs):
    print(f"Book '{instance.title}' by {instance.author} has been saved.")

# Usage
book = Book(title="The Observer Pattern", author="John Doe")
book.save()  # Triggers the post_save signal
```

In this example:

- Book is the subject (model) whose state changes are observed.
- book_saved is the observer (signal listener) function decorated with @receiver. It prints a message whenever a Book instance is saved.
- The post_save signal is emitted whenever a model instance is saved, and the @receiver decorator connects the signal to the observer function.
- When book.save() is called, the post_save signal is triggered, and the book_saved function is automatically called, printing a message about the saved book.

This pattern **promotes loose coupling**, allowing different parts of the codebase to respond to state changes without direct dependencies between them.

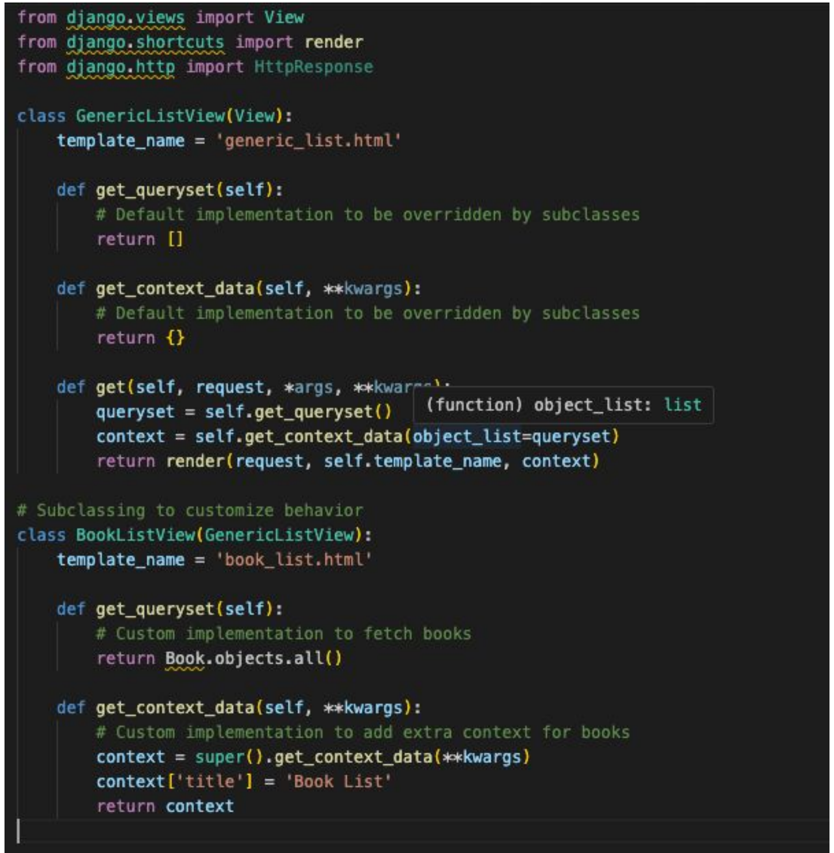
Design Patterns in Django:

Pattern 4: Template Pattern

- In Django, the Template Method pattern can be observed in the use of class-based generic views. The idea is to provide a generic view class that defines the overall structure of a view, including common methods and properties, while allowing specific steps of the algorithm to be redefined by subclassing.
- Elaboration:
    - Algorithm Structure: In the context of Django's class-based generic views, the algorithm represents the process of handling an HTTP request and generating an HTTP response. This includes tasks like retrieving data, processing it, and rendering a template.
    - Template Method: The generic view class serves as the template method, providing a skeletal structure for handling a request. It includes methods for common tasks, and some of these methods may have default implementations.
    - Redefined Steps: Subclasses can then extend or override specific methods of the generic view to tailor the behavior to their specific needs. This allows for flexibility in customizing the view without modifying the overall structure provided by the generic view.

Template Pattern using Class Based Views in Django

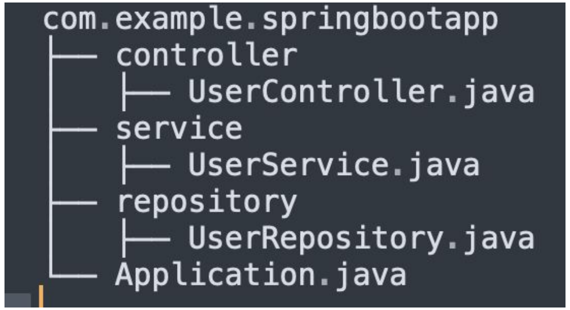
In Django, the Template Method pattern is evident in class-based generic views. These views define the overall structure, including common methods, while allowing specific steps to be redefined by subclassing. This provides flexibility in customizing views without modifying the overall structure.



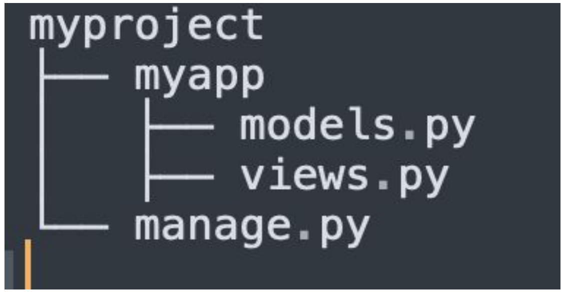
```
from django.views import View
from django.shortcuts import render
from django.http import HttpResponse

class GenericListView(View):
    template_name = 'generic_list.html'

    def get_queryset(self):
        # Default implementation to be overridden by subclasses
        return []

    def get_context_data(self, **kwargs):
        # Default implementation to be overridden by subclasses
        return {}

    def get(self, request, *args, **kwargs):
        queryset = self.get_queryset()   (function) object_list: list
        context = self.get_context_data(object_list=queryset)
        return render(request, self.template_name, context)

# Subclassing to customize behavior
class BookListView(GenericListView):
    template_name = 'book_list.html'

    def get_queryset(self):
        # Custom implementation to fetch books
        return Book.objects.all()

    def get_context_data(self, **kwargs):
        # Custom implementation to add extra context for books
        context = super().get_context_data(**kwargs)
        context['title'] = 'Book List'
        return context
```

Code organization in Spring Boot and Django

- In the given figure, we have showed the code structure. Code structure is essential for maintaining a well-organized and understandable codebase.
- Code organization in Spring Boot and Django follows language-specific conventions.
- Spring Boot uses Java packages and predefined directories, while Django adheres to Python package and module structures.

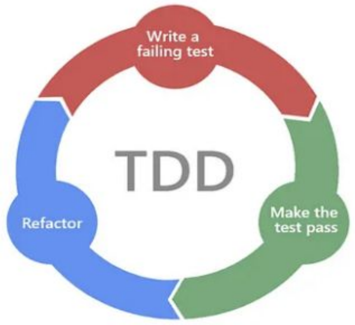


```
com.example.springbootapp
├── controller
│   ├── UserController.java
├── service
│   ├── UserService.java
├── repository
│   ├── UserRepository.java
└── Application.java
```



```
myproject
├── myapp
│   ├── models.py
│   ├── views.py
└── manage.py
```

Test Driven Development:



- Test-driven development (TDD) is a software development approach where tests are written before code, ensuring a focus on specific functionalities.
- The initial tests are expected to fail as no code has been written yet.
- TDD treats these tests as specifications, derived from client user stories, guiding developers to write the minimum code necessary for functionality.
- Similar to the scientific method, TDD involves hypothesis (tests), data gathering, and experimentations, making it a structured and iterative process, but it may be challenging for beginners or unsuitable for exploratory programming.

Test Driven Development: [16]

Uncle Bob's 3 Rules of TDD :

- You are not allowed to write any production code unless it is to make a failing unit test pass.
- You are not allowed to write any more of a unit test than sufficient to fail.
- You are not allowed to write any more production code than sufficient to pass the failing unit test

Test Driven Development: FIRST class test case

- Writing effective test cases is crucial for maintaining code quality and ensuring a smooth development process. Adhering to the "FIRST" class test case principles can guide developers in creating better tests:
  - ***Fast:*** Tests should be quick to execute, ideally completing in a few seconds. Faster tests encourage frequent execution, promoting regular checks before commits.

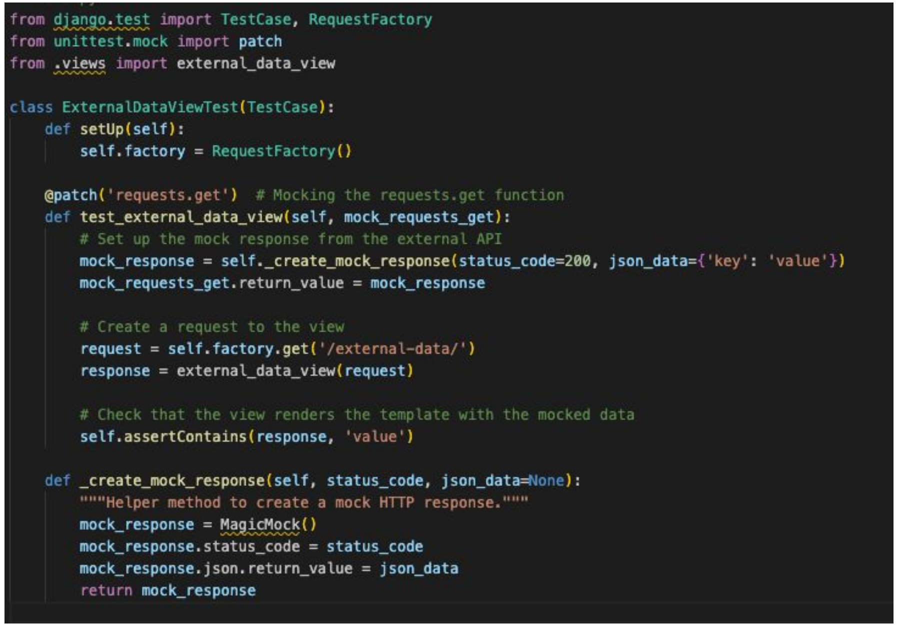
  - ***Independent:*** Each test case should operate independently of others, enabling them to run in any order. Independence prevents one test's failure from affecting others.

  - ***Repeatable:*** Test results must be consistent with each run. Minimize reliance on random factors, ensuring control or setting known values for a reliable outcome.

  - ***Small:*** Test cases should be concise for speed and ease of understanding. Shorter tests are more likely to be executed regularly and are simpler to comprehend.

  - ***Transparent:*** Tests should have clear, straightforward implementations, avoiding ambiguity. Clarity in tests enhances their reliability and maintainability.

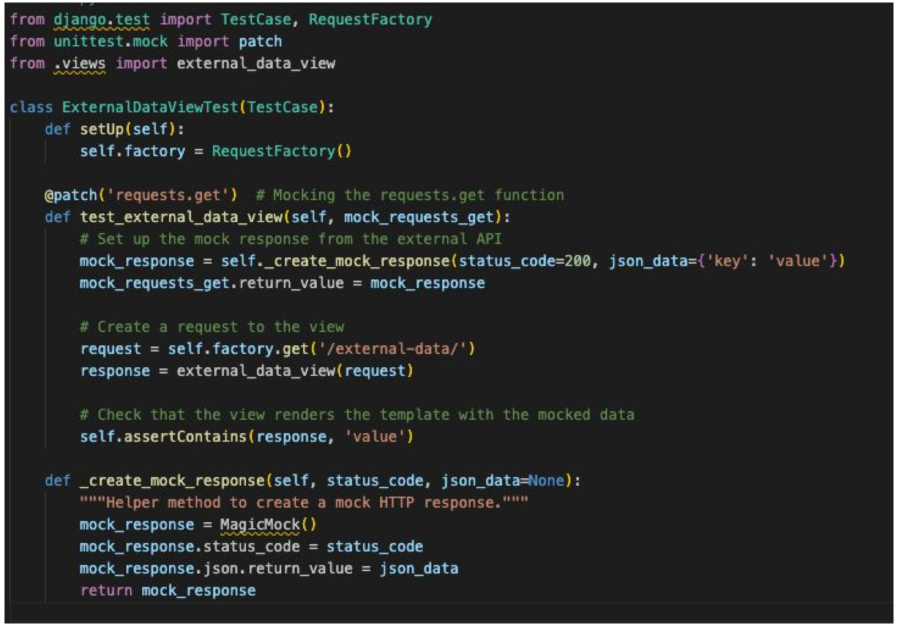
Testing in Django



```
from django.test import TestCase, RequestFactory
from unittest.mock import patch
from .views import external_data_view

class ExternalDataViewTest(TestCase):
    def setUp(self):
        self.factory = RequestFactory()

    @patch('requests.get')  # Mocking the requests.get function
    def test_external_data_view(self, mock_requests_get):
        # Set up the mock response from the external API
        mock_response = self._create_mock_response(status_code=200, json_data={'key': 'value'})
        mock_requests_get.return_value = mock_response

        # Create a request to the view
        request = self.factory.get('/external-data/')
        response = external_data_view(request)

        # Check that the view renders the template with the mocked data
        self.assertContains(response, 'value')

    def _create_mock_response(self, status_code, json_data=None):
        """Helper method to create a mock HTTP response."""
        mock_response = MagicMock()
        mock_response.status_code = status_code
        mock_response.json.return_value = json_data
        return mock_response
```

- Let's consider a scenario where you want to test a Django view that interacts with an external service.
- To isolate the view from the actual external service and make the test independent, you can use mocking.
- We'll use the unittest library for writing the test and the unittest.mock module for mocking.

Testing in Django



```python
from django.test import TestCase, RequestFactory
from unittest.mock import patch
from .views import external_data_view

class ExternalDataViewTest(TestCase):
    def setUp(self):
        self.factory = RequestFactory()

    @patch('requests.get')  # Mocking the requests.get function
    def test_external_data_view(self, mock_requests_get):
        # Set up the mock response from the external API
        mock_response = self._create_mock_response(status_code=200, json_data={'key': 'value'})
        mock_requests_get.return_value = mock_response

        # Create a request to the view
        request = self.factory.get('/external-data/')
        response = external_data_view(request)

        # Check that the view renders the template with the mocked data
        self.assertContains(response, 'value')

    def _create_mock_response(self, status_code, json_data=None):
        """Helper method to create a mock HTTP response."""
        mock_response = MagicMock()
        mock_response.status_code = status_code
        mock_response.json.return_value = json_data
        return mock_response
```

- Suppose you have a Django view that fetches data from an external API and renders it in a template. The external API call is made using the requests library.
- The @patch('requests.get') decorator is used to mock the requests.get function, which is called in the view.
- The test_external_data_view method sets up a mock response for the external API using the helper method _create_mock_response.
- The actual view is then called with the mocked request, and the response is checked to ensure it contains the expected data.
- This way, the test doesn't make an actual external API call, and the behavior of the view can be tested in isolation.

Test Driven Development: Practices to avoid in Django

- Do not (re)test the framework: Trust the robustness of the underlying framework (e.g., Django) and refrain from redundant checks for its functionalities.

- Do not test implementation details: Focus on testing interfaces rather than getting bogged down in minor implementation specifics. This flexibility aids future refactoring without breaking tests.

- Test models most, templates least: Prioritize testing models, as templates typically contain minimal business logic and are subject to frequent changes.

- Avoid HTML output validation: Test views using context variable outputs rather than HTML-rendered outputs for more robust and flexible testing.

- Avoid using the web test client in unit tests: Web test clients are better suited for integration tests due to their interaction with multiple components. Unit tests should remain focused on isolated units of code.

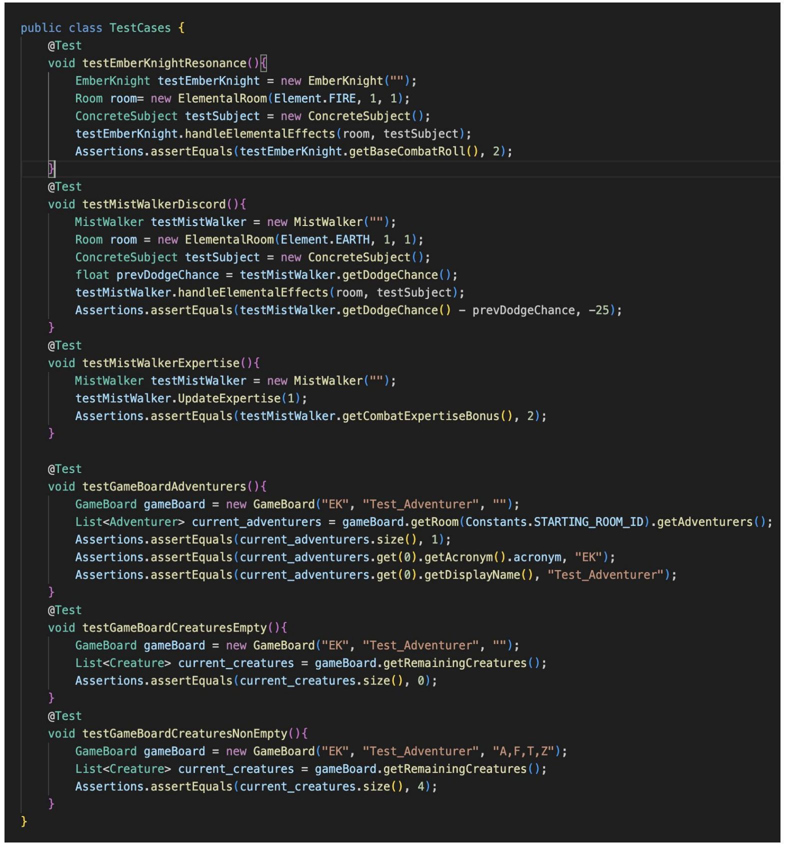
- Avoid interacting with external systems: Whenever possible, mock external systems to maintain test speed and independence. The test database is an exception, given its in-memory nature and efficiency.

Testing in Spring Boot

Testing frameworks used in Spring boot :-

- ***JUnit:*** Widely used for unit testing in Spring Boot applications.
- ***Mockito:*** Used for creating mock objects to isolate and test specific components.
- ***Spring Test:*** Provides annotations and utilities for integration testing within the Spring ecosystem.

Testing in Spring Boot



```java
public class TestCases {
    @Test
    void testEmberKnightResonance(){
        EmberKnight testEmberKnight = new EmberKnight("");
        Room room= new ElementalRoom(Element.FIRE, 1, 1);
        ConcreteSubject testSubject = new ConcreteSubject();
        testEmberKnight.handleElementalEffects(room, testSubject);
        Assertions.assertEquals(testEmberKnight.getBaseCombatRoll(), 2);
    }
    @Test
    void testMistWalkerDiscord(){
        MistWalker testMistWalker = new MistWalker("");
        Room room = new ElementalRoom(Element.EARTH, 1, 1);
        ConcreteSubject testSubject = new ConcreteSubject();
        float prevDodgeChance = testMistWalker.getDodgeChance();
        testMistWalker.handleElementalEffects(room, testSubject);
        Assertions.assertEquals(testMistWalker.getDodgeChance() - prevDodgeChance, -25);
    }
    @Test
    void testMistWalkerExpertise(){
        MistWalker testMistWalker = new MistWalker("");
        testMistWalker.UpdateExpertise(1);
        Assertions.assertEquals(testMistWalker.getCombatExpertiseBonus(), 2);
    }

    @Test
    void testGameBoardAdventurers(){
        GameBoard gameBoard = new GameBoard("EK", "Test_Adventurer", "");
        List<Adventurer> current_adventurers = gameBoard.getRoom(Constants.STARTING_ROOM_ID).getAdventurers();
        Assertions.assertEquals(current_adventurers.size(), 1);
        Assertions.assertEquals(current_adventurers.get(0).getAcronym().acronym, "EK");
        Assertions.assertEquals(current_adventurers.get(0).getDisplayName(), "Test_Adventurer");
    }
    @Test
    void testGameBoardCreaturesEmpty(){
        GameBoard gameBoard = new GameBoard("EK", "Test_Adventurer", "");
        List<Creature> current_creatures = gameBoard.getRemainingCreatures();
        Assertions.assertEquals(current_creatures.size(), 0);
    }
    @Test
    void testGameBoardCreaturesNonEmpty(){
        GameBoard gameBoard = new GameBoard("EK", "Test_Adventurer", "A,F,T,Z");
        List<Creature> current_creatures = gameBoard.getRemainingCreatures();
        Assertions.assertEquals(current_creatures.size(), 4);
    }
}
```

- The example code for TDD in Spring Boot is from our Arcane project.
- The code employs **JUnit** annotations for TDD.
- **Test** annotation identifies test methods, also, assertions validate expected outcomes.
- TDD ensures reliable functionality by writing tests before implementing features

TDD Lifecycle in Spring Boot

- Writing a failing test
  - Begin by creating a test that represents a desired functionality.
  - This test initially fails since the corresponding code is not implemented.
- Write minimal code
  - Implement the minimum code required to make the failing test pass.
  - Focus on simplicity and functionality without overcomplicating the solution.
- Refactor Code
  - Refactor the code to improve its structure and maintainability.
  - Ensure that all tests still pass after refactoring.

Test Driven Development: Practice in Spring Boot

- ***Isolate Tests:*** Ensure each test is independent and does not rely on the state of other tests.
- ***Continuous Integration:*** Integrate TDD into the CI/CD pipeline for automated testing on code changes.
- ***Red-Green-Refactor Cycle:*** Follow the iterative cycle of writing failing tests, making them pass, and refactoring for continuous improvement.

Conclusion

Spring Boot

- Ideal for Java-centric environments and complex enterprise solutions.
- Well-suited for large-scale projects requiring robust scalability and extensive integration capabilities.

Django

- Optimal for Python enthusiasts and projects emphasizing rapid development.
- Excellent for startups and smaller to mid-sized applications with its "batteries-included" approach.

The End