Lecture 4: Web UI Adapter

Marlon Dumas

University of Tartu
Hexagonal architecture

Last weeks: Domain model & DDD

- Structural patterns:
  - Entities, Value objects and Aggregates. Module

- Data access abstraction
  - Repositories, Factories

This week:

- Web UI Adapter
  - MVC
  - Classic Spring Web App Architecture
  - DTOs
Model-View-Controller

• How to organize the code that implements the interactions with end-users?

• MVC Pattern
  ◦ Model: consists of application data and business logic
  ◦ View: any output representation of data (e.g., Chart diagrams, GUIs, Java Swing, HTML pages)
  ◦ Controller: Mediates user interaction, converting it to commands for the model or view
A closer look to MVC

- The MVC pattern has its roots on the early attempts to developing Graphical User Interfaces
  - Introduced in Smalltalk-76, in Xerox Park, in the 70’s
  - Re-implemented later in Smalltalk-80’s class library, in the 80’s

**Model**
- Encapsulates application state
- Responds to state queries
- Exposes application functionality
- Notifies views changes

**View**
- Renders the models
- Requests updates from models
- Sends user gestures to controller
- Allows controller to select view

**Controller**
- Defines application behavior
- Maps user actions to model updates
- Selects view for response
- One for each functionality

**Source:** Oracle
Java Web Applications

HTTP verbs
- Get
- Post
- Update
- Delete

HTTP request
- URL: http://example.com/app?name=Adam

HTTP response
- Status code: 200, 201, 404, 500, etc.
- + content (e.g., HTML, XML, JSON)
Hello world, Servlets!

public class HelloWorld extends HttpServlet {

    public void doGet(HttpServletRequest request, HttpServletResponse response)
            throws ServletException, IOException {

        String name = request.getParameter("name");

        response.setContentType("text/html");

        PrintWriter out = response.getWriter();

        out.println("<HTML>");
        out.println("<HEAD><TITLE>Hello world</TITLE></HEAD>");
        out.println("<BODY><H1>Hello " + name + "]</H1></BODY>");
        out.println("</<HTML>");
    }
}
MVC revisited: The Struts 2 architecture
Spring MVC architecture
Classic Spring Web App Architecture

A Web Application needs to:

1. Process the user’s input and return the correct response back to the user.
2. An exception handling mechanism that provides reasonable error messages to the user.
3. A transaction management strategy.
4. Handle authentication and authorization.
5. Implement the business logic of the application.
6. Communicate with the data storage and other external resources.

Taken from blog by Petri Kainulainen
https://www.petrikainulainen.net/software-development/design/understanding-spring-web-application-architecture-the-classic-way/
Classic Spring Web App Architecture

- **Web Layer**
  (controllers, exception handlers, filters, view templates, and so on)

- **Service Layer**
  (application services and infrastructure services)

- **Repository Layer**
  (repository interfaces and their implementations)

Taken from blog by Petri Kainulainen
[https://www.petrikainulainen.net/software-development/design/understanding-spring-web-application-architecture-the-classic-way/](https://www.petrikainulainen.net/software-development/design/understanding-spring-web-application-architecture-the-classic-way/)
Classic Spring Web App Architecture

**Web Layer**
- (controllers, exception handlers, filters, view templates, and so on)
- Responsible for:
  - processing user’s input and returning the correct response back to the user
  - Handling exceptions thrown by the other layers.
  - Authentication.

**Service Layer**
- (application services and infrastructure services)
- Responsible for:
  - Provides the operations of the application (business logic, validation)
  - Responsible for authorization
  - Communicating with external resources such as file systems, databases, or email servers.

**Repository Layer**
- (repository interfaces and their implementations)
- Responsible for:
  - Communicating with the data storage (search, read, write)
  - Data integrity

Taken from blog by Petri Kainulainen
https://www.petrikainulainen.net/software-development/design/understanding-spring-web-application-architecture-the-classic-way/
Communication between layers

• It is undesirable that domain objects are sent from the repository layer to the Web layer
  ◦ If a domain object (entity object) is read via one connection in the Web layer, then read by a second, then altered by the first, then altered by the second \(\Rightarrow\) inconsistency
  ◦ If we expose our domain model to the outside world, we cannot change it without breaking everything that depends from it. We should be able to change our domain model without breaking all UI layers that depend on it.

• To avoid this, the Web layer only receives copies of data in the form of Data Transfer Objects

• A Data Transfer Object (DTO) is an object used to carry data between the service and the Web layer

• The web layer handles only data transfer objects.

• The service layer takes data transfer objects (and basic types) as method parameters. It can handle domain model objects but it can return only data transfer objects back to the web layer.

• The repository layer takes entities (and basic types) as method parameters and returns entities (and basic types).

Taken from blog by Petri Kainulainen
https://www.petrikainulainen.net/software-development/design/understanding-spring-web-application-architecture-the-classic-way/
Classic Spring Web App Architecture

Web Layer
(controllers, exception handlers, filters, view templates, and so on)

Service Layer
(application services and infrastructure services)

Repository Layer
(repository interfaces and their implementations)

Public

Private

DTOs

Domain
Model
(domain services, entities, and value objects)

Taken from blog by Petri Kainulainen
https://www.petrikainulainen.net/software-development/design/understanding-spring-web-application-architecture-the-classic-way/
What this means in the hexagonal architecture?
The service layer – creating objects

```java
@Service
public class SalesService {
    @Autowired
    PurchaseOrderRepository orderRepo;
    @Autowired
    PlantInventoryEntryRepository plantRepo;
    @Autowired
    SalesIdentifierFactory idFactory;

    public void createPO(PlantInventoryEntryDTO plantDTO, BusinessPeriodDTO periodDTO) {
        PurchaseOrder po = PurchaseOrder.of(
            idFactory.nextPurchaseOrderID(),
            PlantInventoryEntryID.of(plantDTO.getId()),
            periodDTO.asBusinessPeriod());
        DataBinder binder = new DataBinder(po);
        binder.addValidators(new PurchaseOrderValidator());
        binder.validate();

        if (!binder.getBindingResult().hasErrors())
            orderRepo.save(po);
    }
}
```
public class PurchaseOrderValidator implements Validator {

    public void validate(Object object, Errors errors) {
        PurchaseOrder po = (PurchaseOrder) object;

        if (po.getId() == null)
            errors.rejectValue("id", "Purchase Order id cannot be null");
        if (po.getPlantId() == null)
            errors.rejectValue("plantId", "Plant id cannot be null");

        if (!po.status.equals(POStatus.PENDING_CONFIRMATION)) {
            if (po.getReservationId() == null)
                errors.rejectValue("reservationId", "Purchase order’s ...");
        }
        errors.pushNestedPath("rentalPeriod");
        ValidationUtils.invokeValidator(periodValidator, po.getRentalPeriod(), errors);
        errors.popNestedPath();
    }
}
The controller

```java
@Controller
@RequestMapping("/dashboard")
public class DashboardController {
    @Autowired
    PlantCatalogService plantCatalog;
    @Autowired
    SalesService salesService;

    @RequestMapping("/catalog/form")
    String getQueryForm(Model model) {
        model.addAttribute("catalogQuery", new CatalogQueryDTO());
        return "dashboard/catalog/query-form";
    }

    ...
}
```
The controller

@Controller
@RequestMapping("/dashboard")
public class DashboardController {

...
@RequestMapping("/catalog/query")
String executeQuery(CatalogQueryDTO query, Model model) {
    ...
    return "dashboard/catalog/query-result";
}

@RequestMapping("/orders")
String createPO(PurchaseOrderDTO poDTO, Model model) {
    ...
    // extract data from DTO & invoke createPO in SalesService to do the job
    return "redirect:/dashboard";
}
}
Assembler services: From domain objects to DTOs

```java
@Entity
@Getter
public class PlantInventoryEntry {
    @Id
    private String id;
    private String name;
    private String description;
    @Column(precision=8,scale=2)
    private BigDecimal price;
}

@Data
public class PlantInventoryEntryDTO {
    private String _id;
    private String name;
    private String description;
    private BigDecimal price;
}

@Service
public class PlantInventoryEntryAssembler {
    public PlantInventoryEntryDTO toResource(PlantInventoryEntry plant) {
        PlantInventoryEntryDTO dto = new PlantInventoryEntryDTO();
        dto.set_id(plant.getId());
        ...
        return dto;
    }
    public List<PlantInventoryEntryDTO> toResources(List<PlantInventoryEntry> plants) {
        return plants.stream().map(p -> toResource(p)).collect(Collectors.toList());
    }
}
```