Lecture 3

Architectural Styles

Implementing Spring Controllers

Topics covered

- ✧ Architecture design
- ✧ Architectural patterns
  - MVC
  - Layered
  - Repository
  - Client-Server
  - Pipe and filter
- ✧ Spring MVC
  - Request handling workflow
  - Controllers

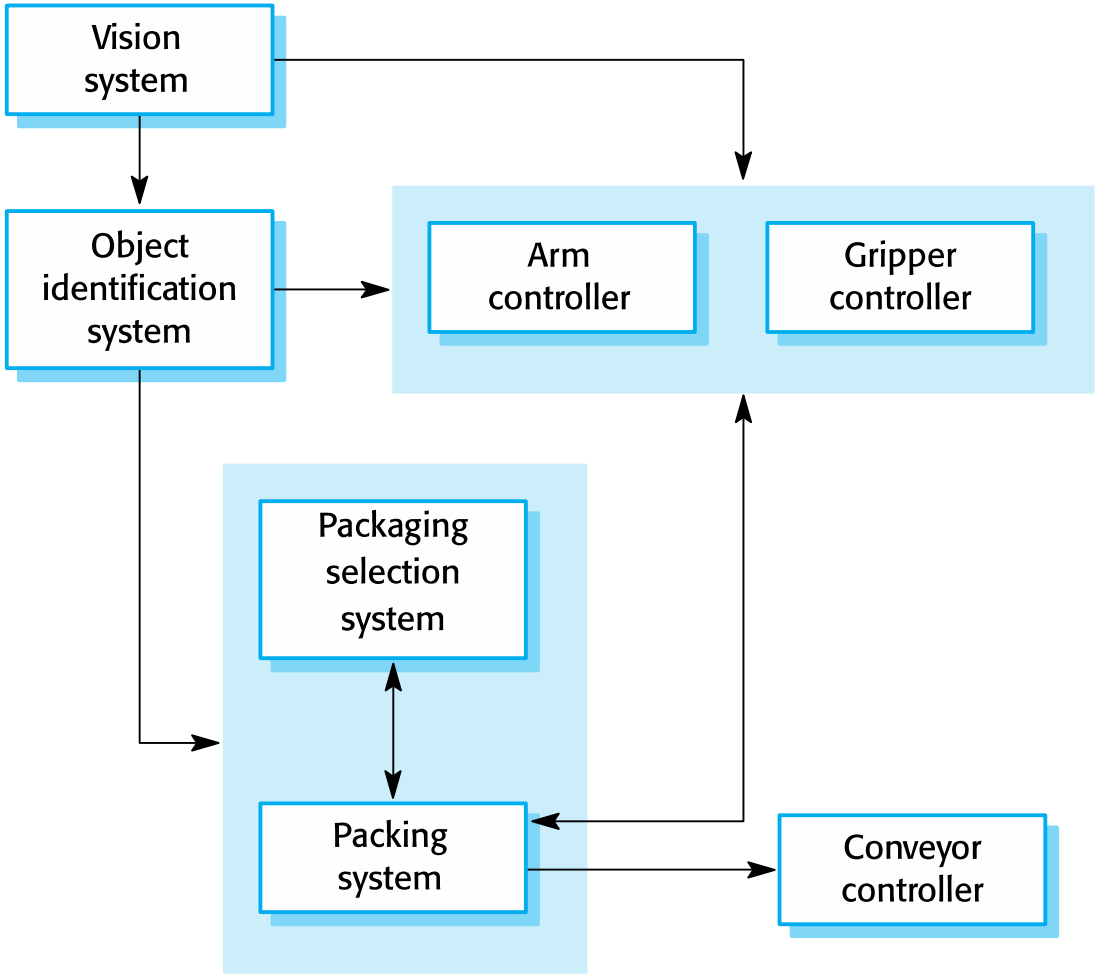
Architectural design

- Architectural design is concerned with understanding how a software system should be organized and designing the overall structure of that system.
- Architectural design is the critical link between design and requirements engineering, as it identifies the main structural components in a system and the relationships between them.
- The output of the architectural design process is an architectural model that describes how the system is organized as a set of communicating components.

Architectural representations

- Simple, informal block diagrams showing entities and relationships are the most frequently used method for documenting software architectures
- Architectures are very abstract and lack semantics
  - They do not show the nature of component relationships nor the externally visible properties of the sub-systems
- However, useful for communication with stakeholders and for project planning

Example: packing robot control system architecture



Advantages of explicit architecture

- ✧ Stakeholder communication
  - Architecture may be used as a focus of discussion by system stakeholders.
- ✧ System analysis
  - Means that analysis of whether the system can meet its non-functional requirements is possible.
- ✧ Large-scale reuse
  - The architecture may be reusable across a range of systems
  - Product-line architectures may be developed.

Use of architectural models

- As a way of facilitating discussion about the system design
  - A high-level architectural view of a system is useful for communication with system stakeholders and project planning because it is not cluttered with detail. Stakeholders can relate to it and understand an abstract view of the system. They can then discuss the system as a whole without being confused by detail.
- As a way of documenting an architecture that has been designed
  - The aim here is to produce a complete system model that shows the different components in a system, their interfaces and their connections.

Architectural Patterns

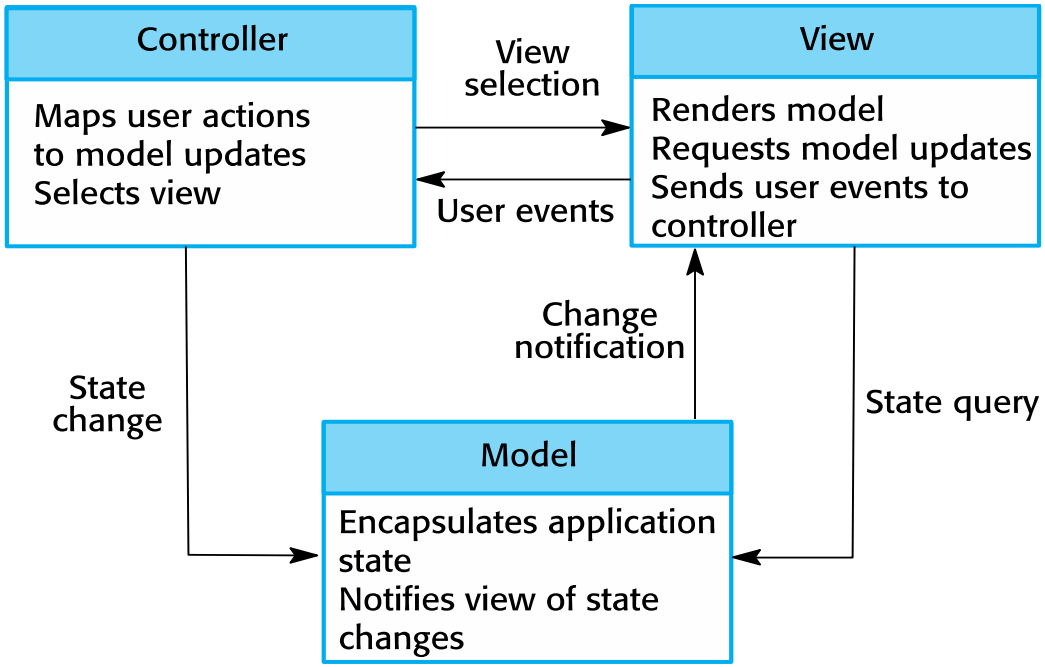
Architectural patterns

- Patterns are a means of representing, sharing and reusing knowledge.
- An architectural pattern is a stylized description of good design practice, which has been tried and tested in different environments.
- Patterns should include information about when they are and when the are not useful.
- Patterns may be represented using tabular and graphical descriptions.

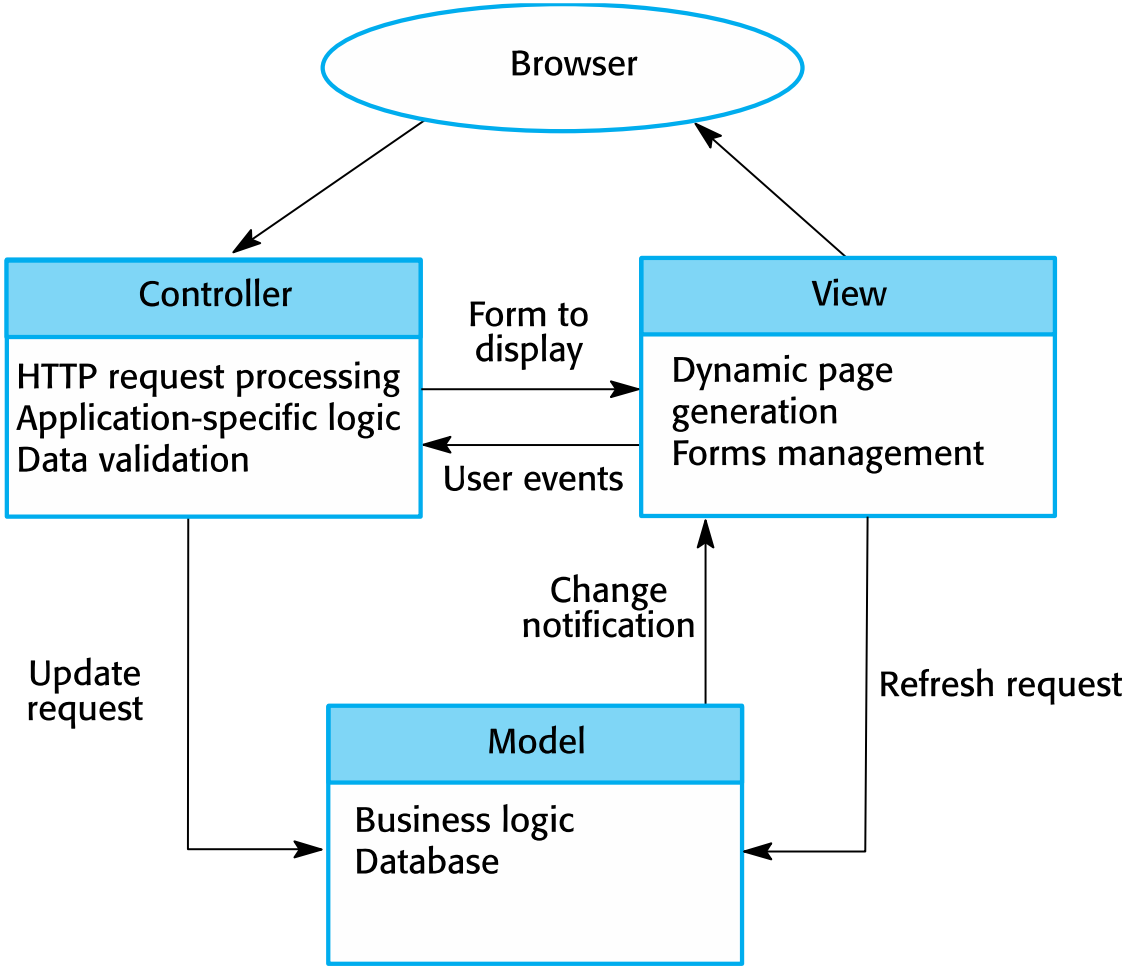
The Model-View-Controller (MVC) pattern

Name	MVC (Model-View-Controller)
Description	Separates presentation and interaction from the system data. The system is structured into three logical components that interact with each other. The Model component manages the system data and associated operations on that data. The View component defines and manages how the data is presented to the user. The Controller component manages user interaction (e.g., key presses, mouse clicks, etc.) and passes these interactions to the View and the Model. See Figure 6.3.
Example	Figure 6.4 shows the architecture of a web-based application system organized using the MVC pattern.
When used	Used when there are multiple ways to view and interact with data. Also used when the future requirements for interaction and presentation of data are unknown.
Advantages	Allows the data to change independently of its representation and vice versa. Supports presentation of the same data in different ways with changes made in one representation shown in all of them.
Disadvantages	Can involve additional code and code complexity when the data model and interactions are simple.

The organization of the Model-View-Controller



Web application architecture using the MVC pattern



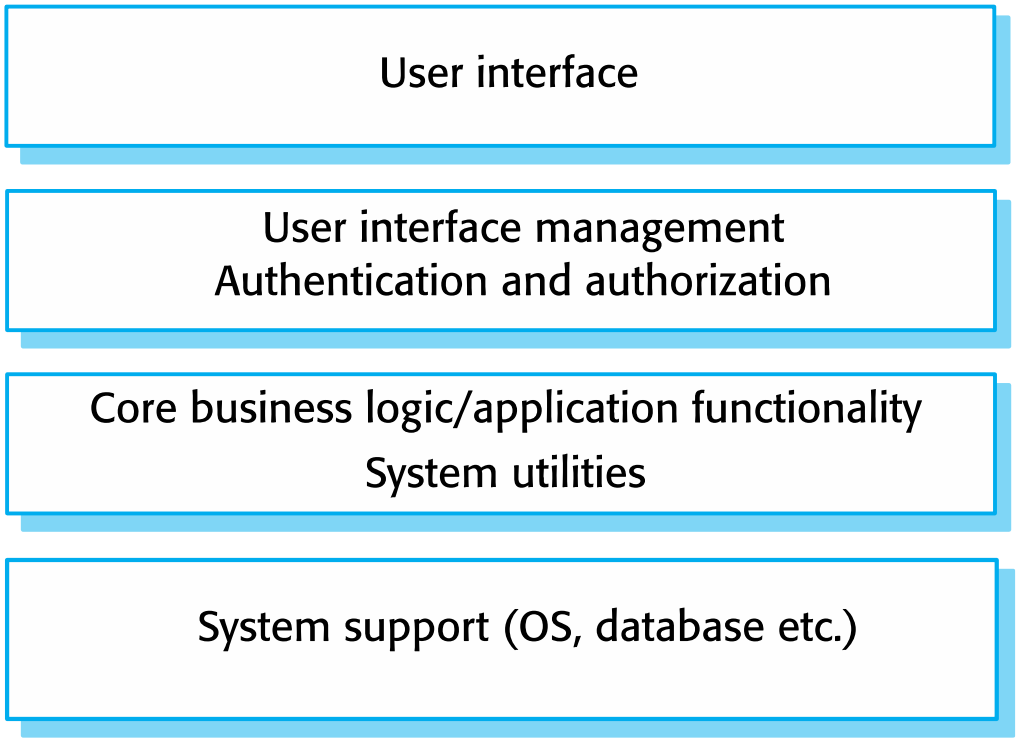
Layered architecture

- Used to model the interfacing of sub-systems.
- Organises the system into a set of layers (or abstract machines) each of which provide a set of services.
- Supports the incremental development of sub-systems in different layers. When a layer interface changes, only the adjacent layer is affected.
- However, often artificial to structure systems in this way.

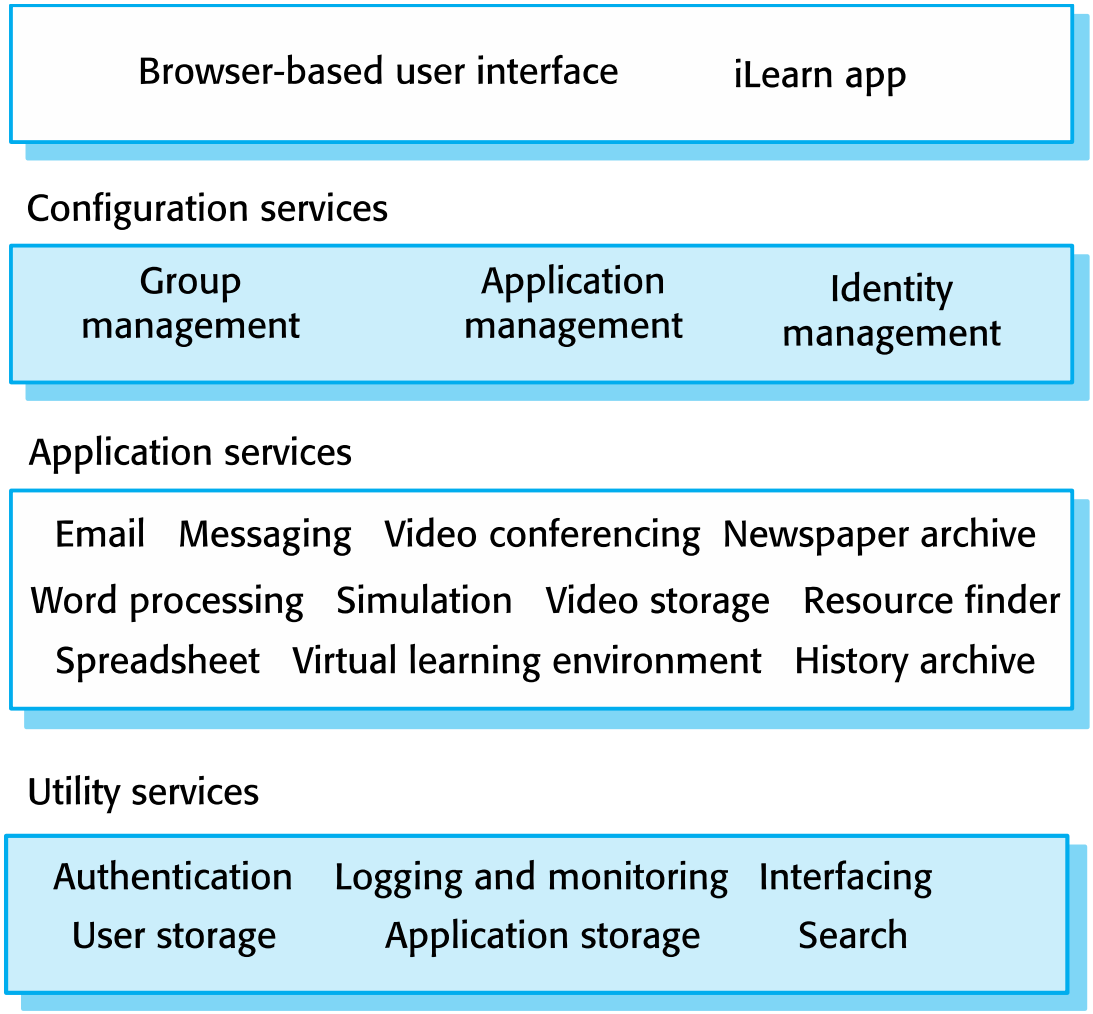
The Layered architecture pattern

Name	Layered architecture
Description	Organizes the system into layers with related functionality associated with each layer. A layer provides services to the layer above it so the lowest-level layers represent core services that are likely to be used throughout the system. See Figure 6.6.
Example	A layered model of a system for sharing copyright documents held in different libraries, as shown in Figure 6.7.
When used	Used when building new facilities on top of existing systems; when the development is spread across several teams with each team responsibility for a layer of functionality; when there is a requirement for multi-level security.
Advantages	Allows replacement of entire layers so long as the interface is maintained. Redundant facilities (e.g., authentication) can be provided in each layer to increase the dependability of the system.
Disadvantages	In practice, providing a clean separation between layers is often difficult and a high-level layer may have to interact directly with lower-level layers rather than through the layer immediately below it. Performance can be a problem because of multiple levels of interpretation of a service request as it is processed at each layer.

A generic layered architecture



The architecture of the iLearn system



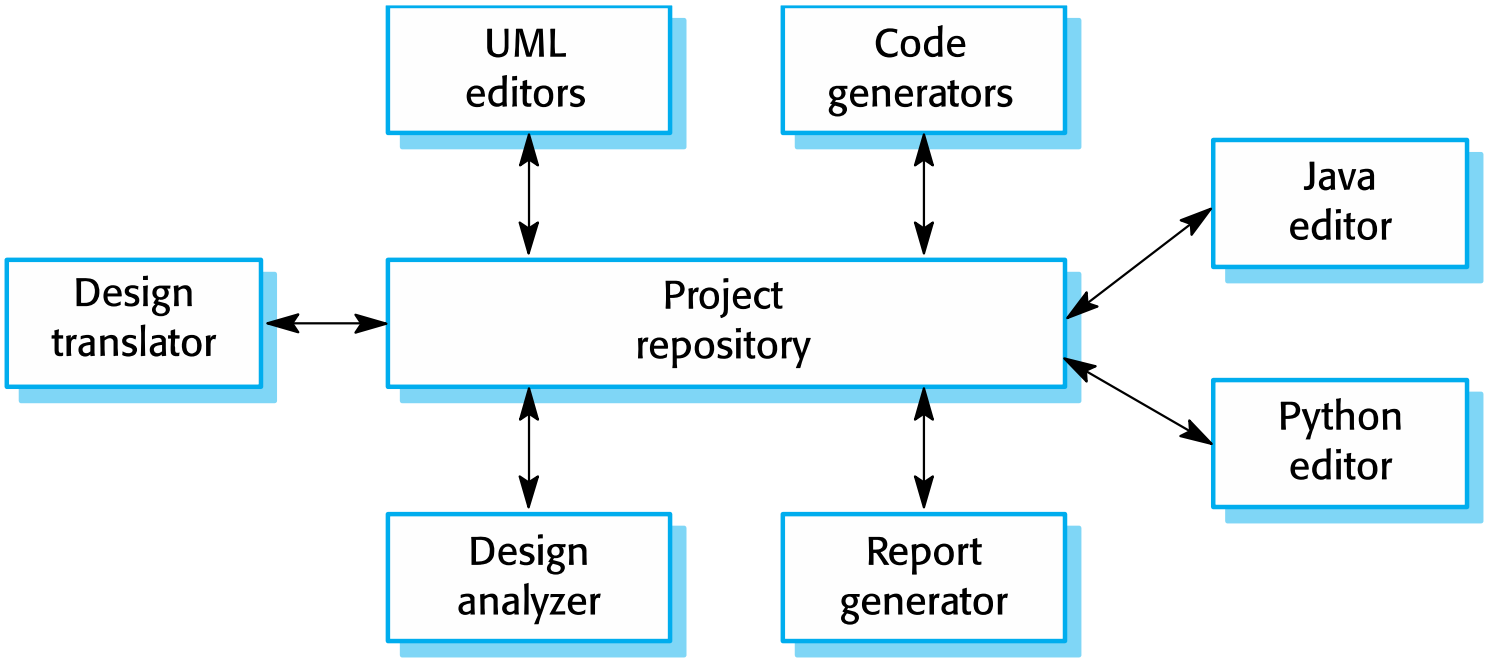
Repository architecture

- Sub-systems must exchange data. This may be done in two ways:
  - Shared data is held in a central database or repository and may be accessed by all sub-systems;
  - Each sub-system maintains its own database and passes data explicitly to other sub-systems.
- When large amounts of data are to be shared, the repository model of sharing is most commonly used a this is an efficient data sharing mechanism.

The Repository pattern

Name	Repository
Description	All data in a system is managed in a central repository that is accessible to all system components. Components do not interact directly, only through the repository.
Example	Figure 6.9 is an example of an IDE where the components use a repository of system design information. Each software tool generates information which is then available for use by other tools.
When used	You should use this pattern when you have a system in which large volumes of information are generated that has to be stored for a long time. You may also use it in data-driven systems where the inclusion of data in the repository triggers an action or tool.
Advantages	Components can be independent—they do not need to know of the existence of other components. Changes made by one component can be propagated to all components. All data can be managed consistently (e.g., backups done at the same time) as it is all in one place.
Disadvantages	The repository is a single point of failure so problems in the repository affect the whole system. May be inefficiencies in organizing all communication through the repository. Distributing the repository across several computers may be difficult.

A repository architecture for an IDE



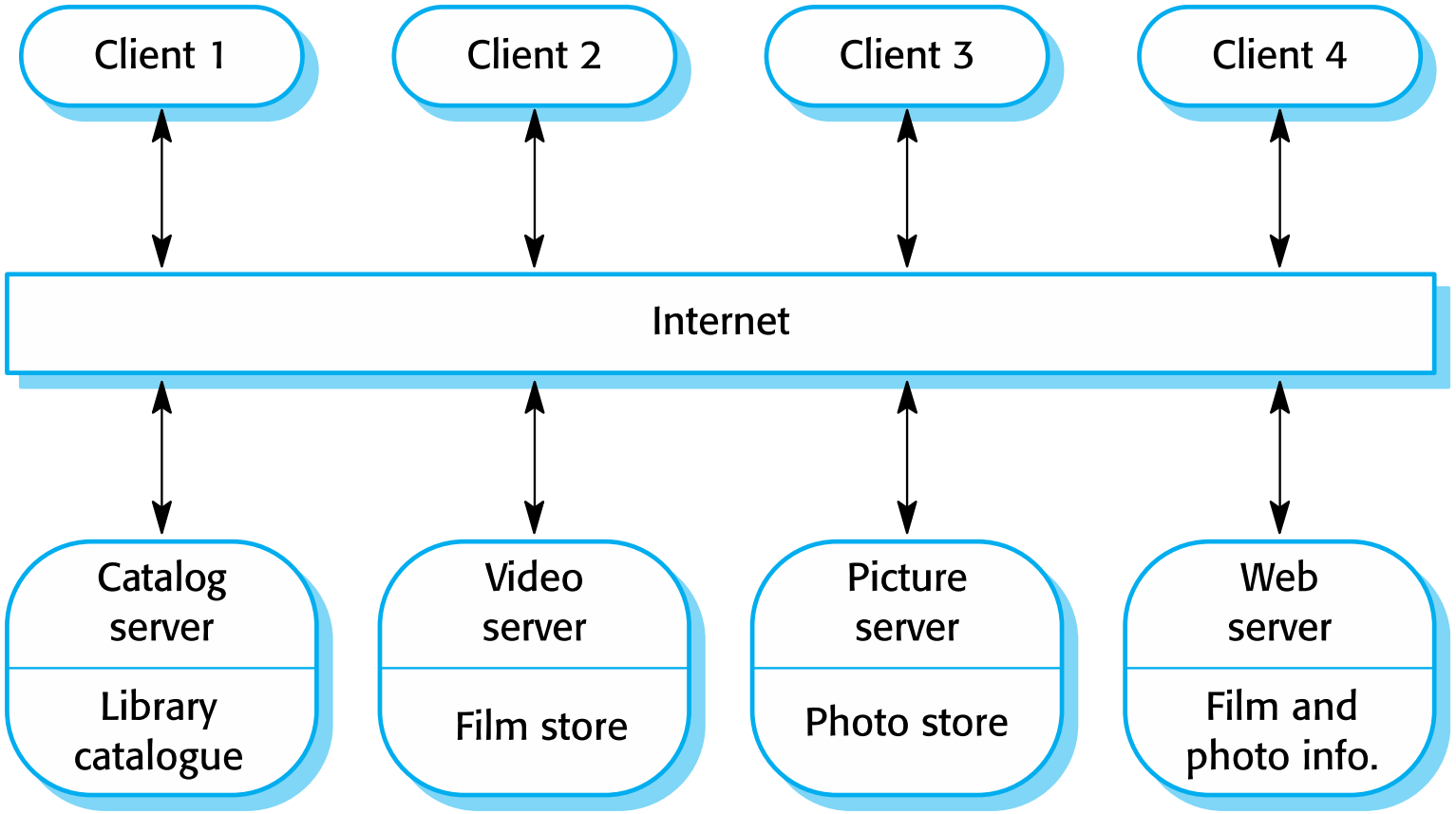
Client-server architecture

- Distributed system model which shows how data and processing is distributed across a range of components.
  - Can be implemented on a single computer.
- Set of stand-alone servers which provide specific services such as printing, data management, etc.
- Set of clients which call on these services.
- Network which allows clients to access servers.

The Client–server pattern

Name	Client-server
Description	In a client–server architecture, the functionality of the system is organized into services, with each service delivered from a separate server. Clients are users of these services and access servers to make use of them.
Example	Figure 6.11 is an example of a film and video/DVD library organized as a client–server system.
When used	Used when data in a shared database has to be accessed from a range of locations. Because servers can be replicated, may also be used when the load on a system is variable.
Advantages	The principal advantage of this model is that servers can be distributed across a network. General functionality (e.g., a printing service) can be available to all clients and does not need to be implemented by all services.
Disadvantages	Each service is a single point of failure so susceptible to denial of service attacks or server failure. Performance may be unpredictable because it depends on the network as well as the system. May be management problems if servers are owned by different organizations.

A client–server architecture for a film library



Web-based information systems

- Information and resource management systems are now usually web-based systems where the user interfaces are implemented using a web browser.
- For example, e-commerce systems are Internet-based resource management systems that accept electronic orders for goods or services and then arrange delivery of these goods or services to the customer.
- In an e-commerce system, the application-specific layer includes additional functionality supporting a ‘shopping cart’ in which users can place a number of items in separate transactions, then pay for them all together in a single transaction.

Web server implementation

- Web systems are often implemented as multi-tier client-server architecture
  - The web server is responsible for all user communications, with the user interface implemented using a web browser;
  - The application server is responsible for implementing application-specific logic as well as information storage and retrieval requests;
  - The database server moves information to and from the database and handles transaction management.

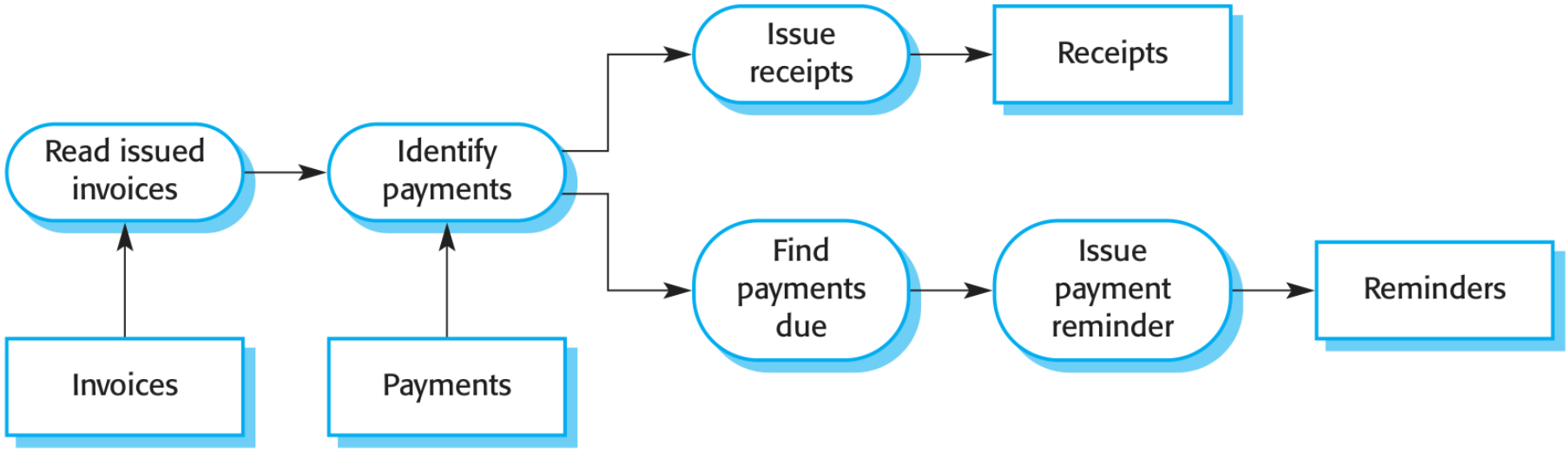
Pipe and filter architecture

- Functional transformations process their inputs to produce outputs.
- May be referred to as a pipe and filter model (as in UNIX shell).
- Variants of this approach are very common. When transformations are sequential, this is a batch sequential model which is extensively used in data processing systems.
- Not really suitable for interactive systems.

The pipe and filter pattern

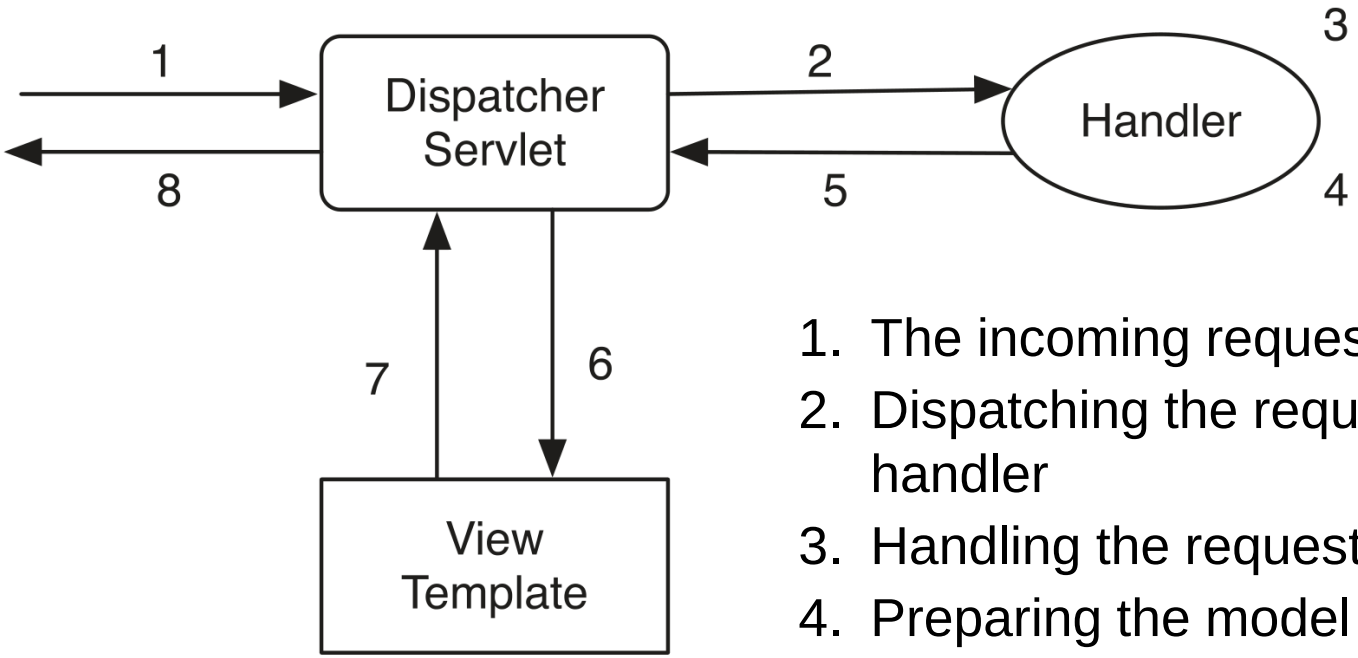
Name	Pipe and filter
Description	The processing of the data in a system is organized so that each processing component (filter) is discrete and carries out one type of data transformation. The data flows (as in a pipe) from one component to another for processing.
Example	Figure 6.13 is an example of a pipe and filter system used for processing invoices.
When used	Commonly used in data processing applications (both batch- and transaction-based) where inputs are processed in separate stages to generate related outputs.
Advantages	Easy to understand and supports transformation reuse. Workflow style matches the structure of many business processes. Evolution by adding transformations is straightforward. Can be implemented as either a sequential or concurrent system.
Disadvantages	The format for data transfer has to be agreed upon between communicating transformations. Each transformation must parse its input and unparse its output to the agreed form. This increases system overhead and may mean that it is impossible to reuse functional transformations that use incompatible data structures.

An example of the pipe and filter architecture used in a payments system



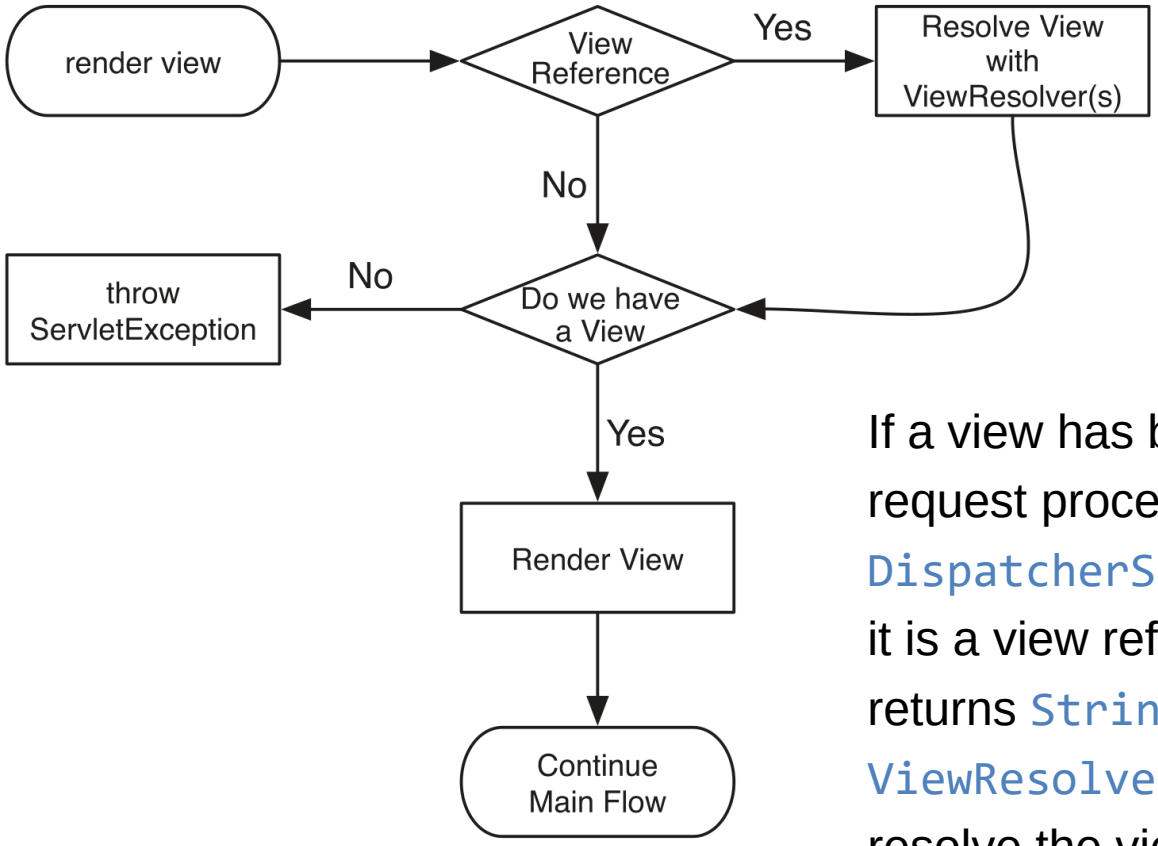
Implementing Spring MVC Controllers

Spring MVC Workflow



1. The incoming request
2. Dispatching the request to the handler
3. Handling the request
4. Preparing the model and selecting the view
5. Returning `ModelAndView` object
6. Rendering `View` with the model
7. Returning control to the servlet
8. Returning the response to the client

The view rendering process



If a view has been selected during the request processing workflow, `DispatcherServlet` first checks whether it is a view reference (in case the controller returns `String`). If so, the configured `ViewResolver` beans are consulted to resolve the view reference to an actual `View` implementation.

What is a Controller?

- The controller is the component that is responsible for responding to the action the user takes
  - form submission, clicking a link, or simply accessing a page
- The controller may:
  - selects or updates the data needed for the view
  - selects the name of the view to render
  - renders the view itself

Types of Spring Controller

✧ Interface-based controller

- Strict method signatures that we must follow
- Explicit mapping of URLs to controllers → all URLs are in a single location

✧ Annotation-based controller:

- **Our focus**
- Flexible method signatures
- Mappings of URLs to controllers scattered throughout the codebase

Annotation-based Controllers

✧ To create an annotation-based controller, we need to:

- Put the `@Controller` annotation on a class `org.springframework.stereotype.Controller`
- Add a `@RequestMapping` annotation to the class, a method of the class, or both `org.springframework.web.bind.annotation.RequestMapping`

```
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.servlet.ModelAndView;

@Controller
public class IndexController {
    @RequestMapping(value = "/index.htm")
    public ModelAndView indexPage() {
        return new ModelAndView("index");
    }
}
```

Mapping Requests to Controllers

- Out of the box, Spring MVC provides four different `HandlerMapping` implementations (most of them are based on URL mapping)
  - `BeanNameUrlHandlerMapping`
  - `SimpleUrlHandlerMapping`
  - `RequestMappingHandlerMapping` (our focus)
  - `RouterFunctionMapping`

Request-Handling Methods

- ✧ These factors (and more) can influence which method is selected to handle the request:
  - The request URL
  - The HTTP method used (e.g., GET or POST)
  - The request parameters, headers or content type (`text/plain`, `application/json`, etc.)
- ✧ How to?
  - Put a `@RequestMapping` annotation on the method
  - Be more specific about which requests to map to the method by specifying the `@RequestMapping` annotation's attributes

The `RequestMapping` Attributes (1)

- ✧ `name`
  - The name to use for this mapping. the name can be used to generate dynamic links
- ✧ `value` or `path`
  - Specifies to which URL or URLs this controller reacts to, such as `/order.htm`.
- ✧ `method`
  - Binds the method on specific http methods (GET, POST, PUT, DELETE, HEAD, OPTIONS, or TRACE)

The `RequestMapping` Attributes (2)

✧ `params`

- Narrows the requests down on the existence or absence of request parameters. Supported expressions include:
  - `param-name=param-value`
  - `param-name!=param-value`
  - `!param-name`

✧ `headers`

- Narrows the requests down on the existence or absence of HTTP request headers. Supported expressions include:
  - `header-name=header-value`
  - `header-name!=header-value`
  - `!header-name`

✧ (more attributes omitted)

RequestMapping examples (1)

```
@Controller
@RequestMapping("/order.htm")
public class IndexController {
    @RequestMapping(method = RequestMethod.GET)
    public String getOrder() {
        return "getorder";
    }

}
```

✧ Map the `getOrder` method to all get requests to the `order.htm` URL

RequestMapping examples (2)

```
@Controller
@RequestMapping("/order.*")
public class IndexController {
    @RequestMapping(method = {RequestMethod.PUT, RequestMethod.POST})
    public String processOrder() {
        return "processorder";
    }

}
```

- Map `processOrder` method to all PUT and POST requests to the `order.*` URL. `*` means any suffix or extension such as .htm, .doc, .xls, etc.

Supported argument types (for Controller methods)

- ✧ `org.springframework.ui.Model`
  - The implicit model belonging to this controller/request
- ✧ `javax.servlet.http.HttpServletRequest`
  - The http request object that triggered this method
- ✧ `org.springframework.web.multipart.MultipartRequest`
  - The request object that triggered this method (only works for multipart requests). This wrapper allows easy access to the uploaded files(s).
- ✧ (*many more argument types not presented here*)

Example of using `Model` argument type

```
@RequestMapping(value = "/")
public String getAllEmployee(Model model) {
    List<Employee> employees = employeeRepository.findAll();
    model.addAttribute("employees", employees);
    return "employeeList";
}
```

Supported argument annotations (for Controller methods)

- ✧ `@RequestParam`
  - Binds the argument to a single request parameter or all request parameters
- ✧ `@RequestHeader`
  - Binds the argument to a single request header or all request headers
- ✧ `@RequestBody`
  - Gets the request body for arguments with this annotation. The value is converted using `org.springframework.http.converter.httpmessageConverter`
- ✧ `@CookieValue`
  - Binds the method parameter to a `javax.servlet.http.Cookie`

Example of using argument annotations

```
@RequestMapping(path = "/api/foo", method = RequestMethod.GET)
public String getFoo(@RequestParam String id) {
    return "ID: " + id;

}
```

/api/foo?id=123

Key points

- A software architecture is a description of how a software system is organized.
- Architectural patterns are a means of reusing knowledge about generic system architectures. They describe the architecture, explain when it may be used and describe its advantages and disadvantages.
- In this lesson, you learned to implement basic Spring Controllers for web application.