Spring MVC

Based on slides from *Mastering Spring MVC 3*

Intro

- What's the problem with the web development approaches we've seen so far?

```
@Override
protected void doGet(HttpServletRequest request,
  HttpServletResponse response) throws ServletException, IOException {

  response.setContentType("application/json");
  response.setCharacterEncoding("UTF-8");

  String in = request.getParameter("input");
  String result = process(in);
  response.getWriter().println(result);
}
```

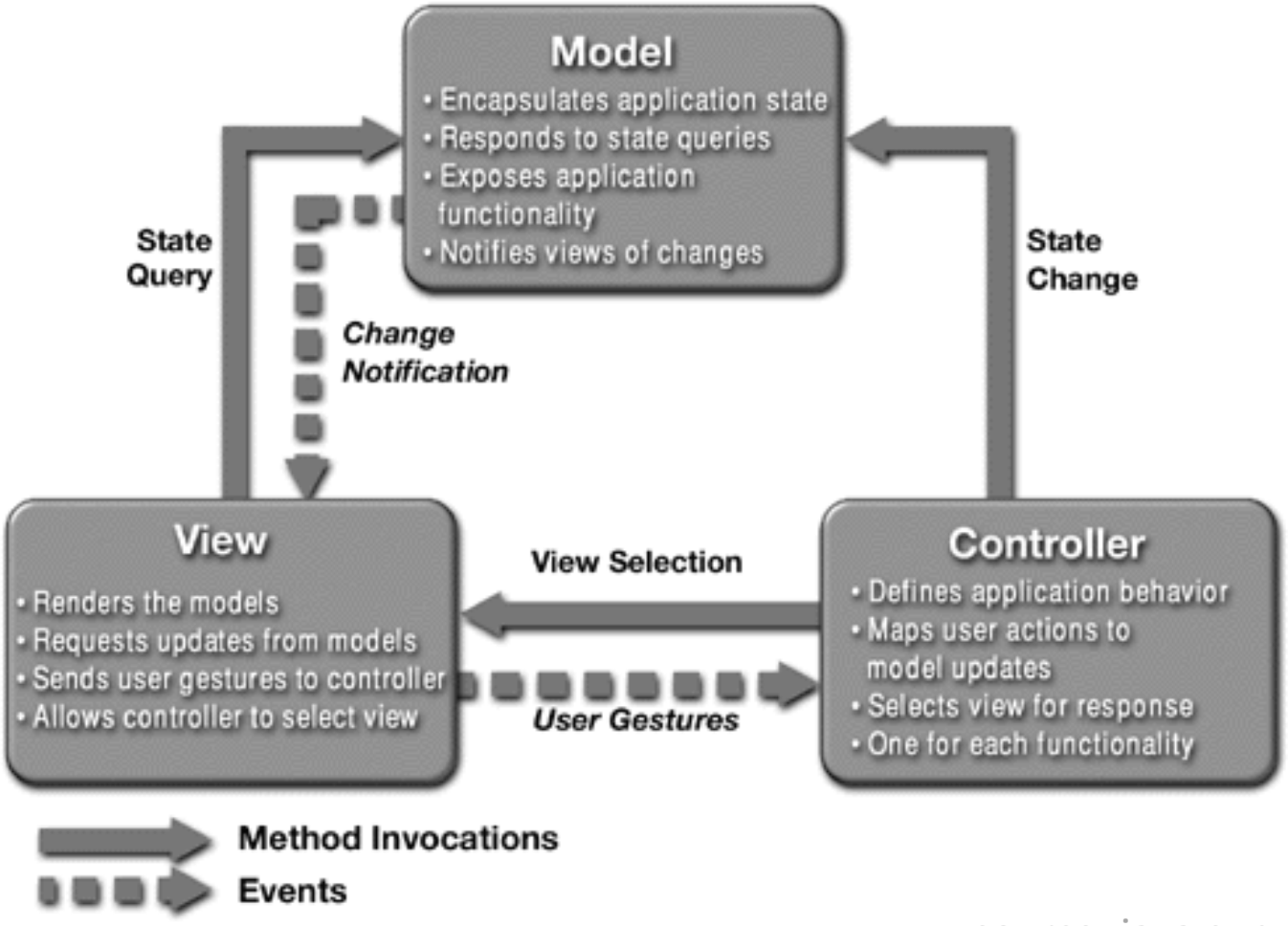
Client/Server coupling

- Servlet need to know the format of the answer (JSON/XML)
- Parameters are resolved within the Servlet (strings, integers, classes)
- As a result most of the Servlet code is handling “bureaucracy” which opens the door for bugs and consume a lot of development time
- Hard to decouple the UI (HTML/JSP) code from the data handling and logic.

Model-View-Controller

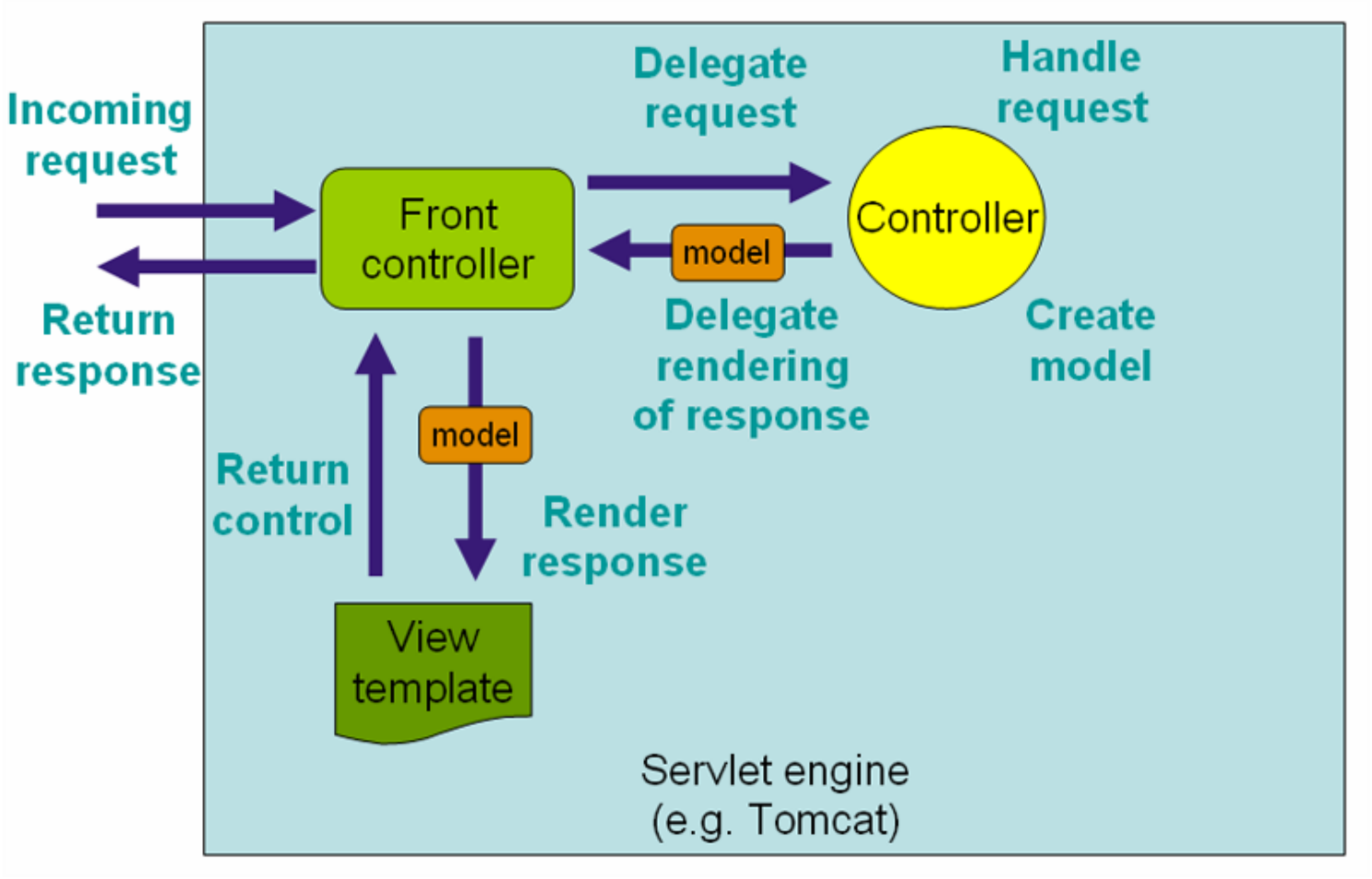
- Simple design pattern
- A logical concerns separation to 3 parts:
  - Model: Data extraction & manipulation
  - View: Graphical presentation / UI
  - Controller: Logic
- Goals:
  - Independent development of each
  - Ignorance is a blessing
  - Simpler code

Model-View-Controller



source: java.sun.com

Spring programming model



The new Servlet

- DispatcherServlet requests are mapped to @Controller methods
  - @RequestMapping annotation used to define mapping rules
  - Method parameters used to obtain request input
  - Method return values used to generate responses
- Simplest possible @Controller

```
@Controller
public class HelloController {
  @RequestMapping(“/”)
  public @ResponseBody String hello() {
    return “Hello World”;
  }
}
```

Mapping Requests

- By path
  - @RequestMapping(“path”)
- By HTTP method
  - @RequestMapping(“path”,method=RequestMethod.GET)
  - POST,PUT,DELETE,OPTIONS and TRACE are also supported
- By presence of query parameter
  - @RequestMapping(“path”,method=RequestMethod.GET,params=“foo”)
  - Negation also supported: params={“foo”,”!bar”}
- By presence of request header
  - @RequestMapping(“path”,header=“content-type=text/*”)
  - Negation also supported

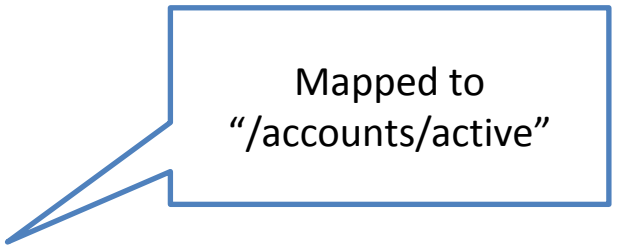
Request mapping at class level

- @RequestMapping can be used at the class level
  - Concise way to map all requests *within* a path to a @Controller

```
@Controller
@RequestMapping("/accounts/*")
public class AccountsController {

  @RequestMapping("active")
  public @ResponseBody List<Account> active() {…}

  @RequestMapping("inactive")
  public @ResponseBody List<Account> inactive() {…}
}
```



Obtaining request data

- Previously we needed to investigate the `HttpServletRequest` object for parameters
- Say we want to add a new Employee to our online DB

```
@Override
protected void doGet(HttpServletRequest req,
  HttpServletResponse res) throws ServletException,IOException {

  String name = req.getParameter("name");
  if (name == null) {… //handle missing param }

  String ageStr = req.getParameter("age");
  if (ageStr == null) {…}
  int age;
  try {
    age = Integer.parseInt(ageStr);
  } catch(NumberFormatException e) {
    //handle invalid numeric value
  }

  Employee employee = new Employee(name,age);
  //Add employee to the DB
  …
}
```

Obtaining request data

- Using @RequestParam

```
@Controller
@RequestMapping(“/employee/*”)
public class EmployeeController {

  @RequestMapping(“add”)
  public String add(@RequestParam String name, @RequestParam String age) {

    Employee employee = new Employee(name,age);
    //Add employee
    …
  }
}
```

Obtaining request data

- But why not just use Employee as the method parameter?

```
@Controller
@RequestMapping("/employee/*")
public class EmployeeController {

  @RequestMapping("add")
  public String add(Employee employee) {
    //Add employee
    …
  }
}
```

- Spring can convert request parameters to all of Java's simple types, as well as a few basic types (String, Arrays, Date, etc.) and to Java beans
- You can get request data from URL/headers/cookies/request body

Java beans

- A Java bean is a Java class with getters & setters for some of its private members

```
@RooJavaBean
public class Employee {

  private String name;
  private int age;

  public String getName() {return name;}
  public void setName(String _name) {this.name = _name;}

  public int getAge() {return age;}
  public void setAge(int _age) {this.age = _age;}
}
```

- Spring will find *name* and *age* in the request parameters and will match them to the Employee bean by their name.
- More complex data conversions are possible via custom *Resolvers.* These will be *injected* when needed

Generating responses

- Can return a POJO annotated with @ResponseBody
  - Will be returned as the body of the response
  - Automatic conversions to client's expected format (JSON,XML,…)

```
@Controller
@RequestMapping("/accounts/*")
public class AccountsController {

        @RequestMapping("active")
        public @ResponseBody List<Account> active() {…}

        @RequestMapping("inactive")
        public @ResponseBody List<Account> inactive() {…}
}
```

- Return a new ResponseEntity<T> object
  - More powerful; allows setting custom headers and status code

Returning a *View*

- In order for the controller to avoid messing with the *view* code, it is custom to simply pass on control to a designated *view.*
- The view can be any static or dynamic created content.

```
@Controller
@RequestMapping("/employee/*")
public class EmployeeController {

  @RequestMapping("add")
  public String add(Employee employee) {
    //Add employee
    …
    if (success) return "successAdd";
    return "failedAdd";
  }
}
```

Will pass request to "successAdd.jsp"

Model processing

- The *Model* is a simple data container.
- The @Controller should prepare the *Model* for the specific action and pass it on to the *View to be rendered accordingly*

```
@Controller
@RequestMapping(“/employee/*”)
public class EmployeeController {

  @RequestMapping(“add”)
  public String add(Model model, Employee employee) {
    //Add employee to DB
    …
    model.addAtrribute(employee);
    if (success) return “successAdd”;
    return “failedAdd”;
  }
}
```

View with model data

- SuccessAdd.jsp code:

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<html>
  <head>
    <title>Employee added</title>
  </head>
  <body>
    <h3>Employee ${employee.getName()} successfully added</h3>
  </body>
</html>
```

- Try doing this using standard methods and you'll need: cookies/sessions/static variables…

Views vs. @ResponseBody

- Which one do I use?
  - Use views to generate documents for display in a web browser: HTML, PDF, etc.
  - Use @ResponseBody to exchange data with web service clients (e.g. JS scripts): JSON, XML, etc.

Demo

Only scratched the surface…

- Haven't covered today:
  - Type conversion
  - Validation
  - FTP services
  - Exception handling
  - Testing using Mockito (to be seen in later lectures)
  - Load balancing
- Reference Manual
  - http://www.springsource.org/documentation
- Show case video
  - http://www.infoq.com/presentations/Mastering-Spring-MVC-3