Spring MVC

Spring MVC

- We've seen how spring can integrate with regular servlets. Unfortunately this a bit clunky.
  - Servlets are not spring beans, no DI or AOP

- Spring MVC is a web application framework
  - **Request-based** framework (not component)
  - using the **Front-Controller** pattern
  - Built on top of the **Servlet API**

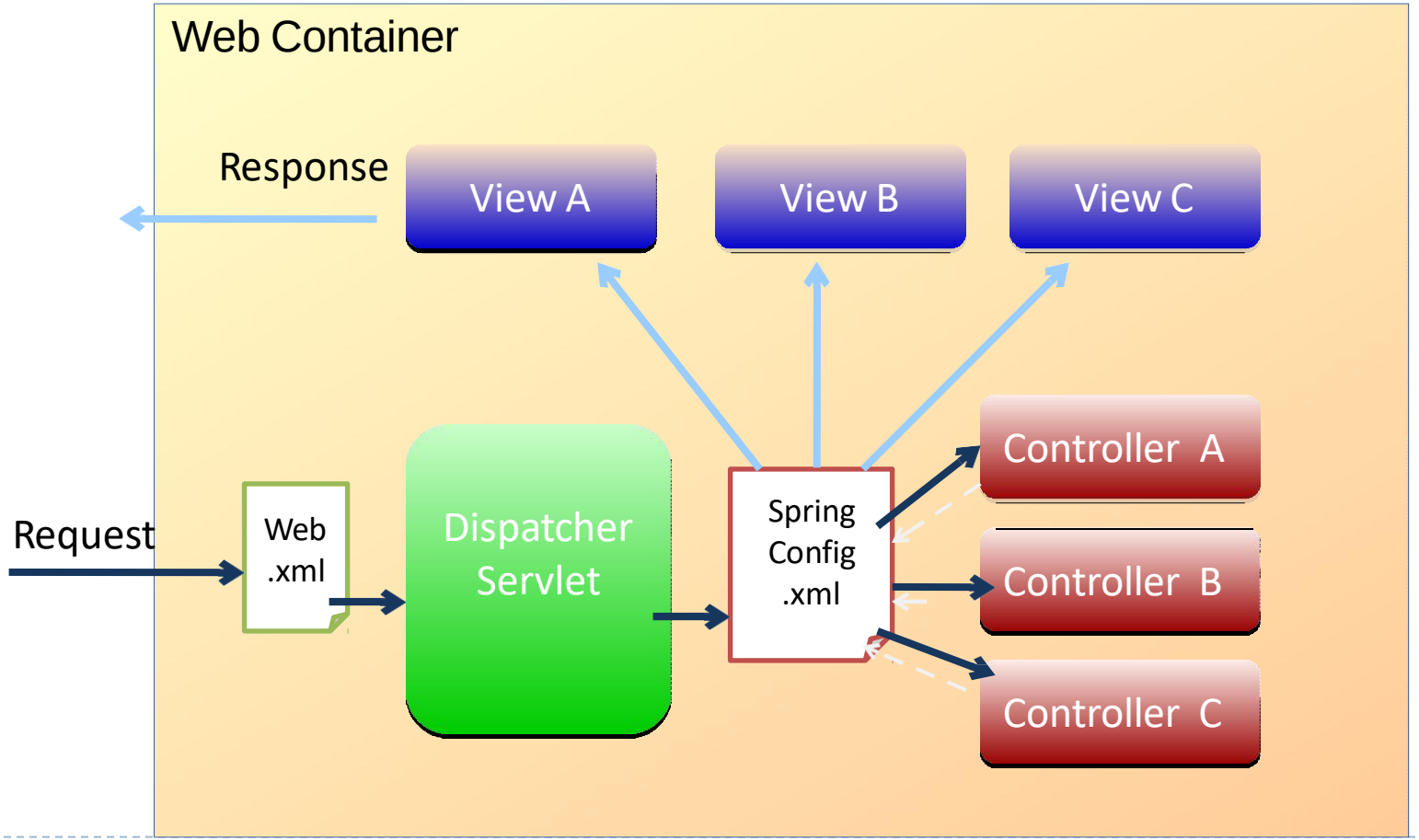
available if desired

Request Based

- Spring MVC **embraces the web**
  - In the API it's clear that you're parsing a HTTP request
  - And outputting a HTTP / HTML response

- As opposed to Component Based frameworks (JSF)
  - Which try to abstract out the web
  - To make it feel more like a desktop application

Front Controller / Dispatcher



WebApplicationInitializer

Instead of web.xml

```
public class MyWebAppInitializer implements WebApplicationInitializer {

        @Override
        public void onStartup(ServletContext container) throws ServletException {
                // Create the 'root' Spring application context
                AnnotationConfigWebApplicationContext rootContext =
                        new AnnotationConfigWebApplicationContext();
                rootContext.register(WebConfig.class);
                container.addListener(new ContextLoaderListener(rootContext));

                // Create the dispatcher servlet
                ServletRegistration.Dynamic appServlet = container.addServlet("mvc",
                        new DispatcherServlet(new GenericWebApplicationContext()));
                appServlet.setLoadOnStartup(1);
                appServlet.addMapping("/");
        }
}
```

AbstractAnnotationConfigDispatcherServletInitializer

```
public class MyWebInitializer extends
        AbstractAnnotationConfigDispatcherServletInitializer {

    @Override
    protected Class<?>[] getRootConfigClasses() {
        return new Class[]{RootConfig.class};
    }

    @Override
    protected Class<?>[] getServletConfigClasses() {
        return new Class[]{WebConfig.class};
    }

    @Override
    protected String[] getServletMappings() {
        return new String[]{"/"};
    }

}
```

Web.xml

Can be used instead of WebApplicationInitializer

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app ...>

  <!-- The definition of the Root Spring Container shared by all Servlets and Filters -->
  <context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>/WEB-INF/spring/root-context.xml</param-value>
  </context-param>

  <! -- Creates/Starts the Root Spring Container shared by all Servlets and Filters -->
  <listener>
    <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>
  </listener>

  <!-- Creates the dispatcher servlet and its configuration -->
  <servlet>
    <servlet-name>spring</servlet-name>
    <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
    <init-param>
      <param-name>contextConfigLocation</param-name>
      <param-value>/WEB-INF/spring/dispatcher-context.xml</param-value>
    </init-param>
    <load-on-startup>1</load-on-startup>
  </servlet>

  <servlet-mapping> <!-- Maps the dispatcher servlet to all requests in this project -->
    <servlet-name>spring</servlet-name>
    <url-pattern>/</url-pattern>
  </servlet-mapping>
</web-app>
```

Optional root Application Context

Optional param. Defaults to: [servlet-name]-servlet.xml

WebConfig.java

```
@Configuration
@EnableWebMvc
@ComponentScan("cs544")
public class WebConfig implements WebMvcConfigurer{
		@Bean
		public ViewResolver viewResolver() {
				InternalResourceViewResolver bean = new InternalResourceViewResolver();

				bean.setViewClass(JstlView.class);
				bean.setPrefix("/WEB-INF/view/");
				bean.setSuffix(".jsp");

				return bean;
		}
}
```

Dispatcher-context.xml

```xml
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:context="http://www.springframework.org/schema/context"
xmlns:mvc="http://www.springframework.org/schema/mvc"
xsi:schemaLocation="http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-beans.xsd
       http://www.springframework.org/schema/context
       http://www.springframework.org/schema/context/spring-context.xsd
       http://www.springframework.org/schema/mvc
       http://www.springframework.org/schema/mvc/spring-mvc.xsd">

  <!- scan for @RequestMapping annotations-->
  <mvc:annotation-driven />

  <!- scan for @Controller (and other component) annotations in the following package -->
  <context:component-scan base-package="springmvc.helloworld" />

  <!-- Resolves views to .jsp resources in the /WEB-INF/views directory -->
  <bean class="org.springframework.web.servlet.view.InternalResourceViewResolver">
    <property name="viewClass" value="org.springframework.web.servlet.view.JstlView" />
    <property name="prefix" value="/WEB-INF/views/" />
    <property name="suffix" value=".jsp" />
  </bean>
</beans>
```

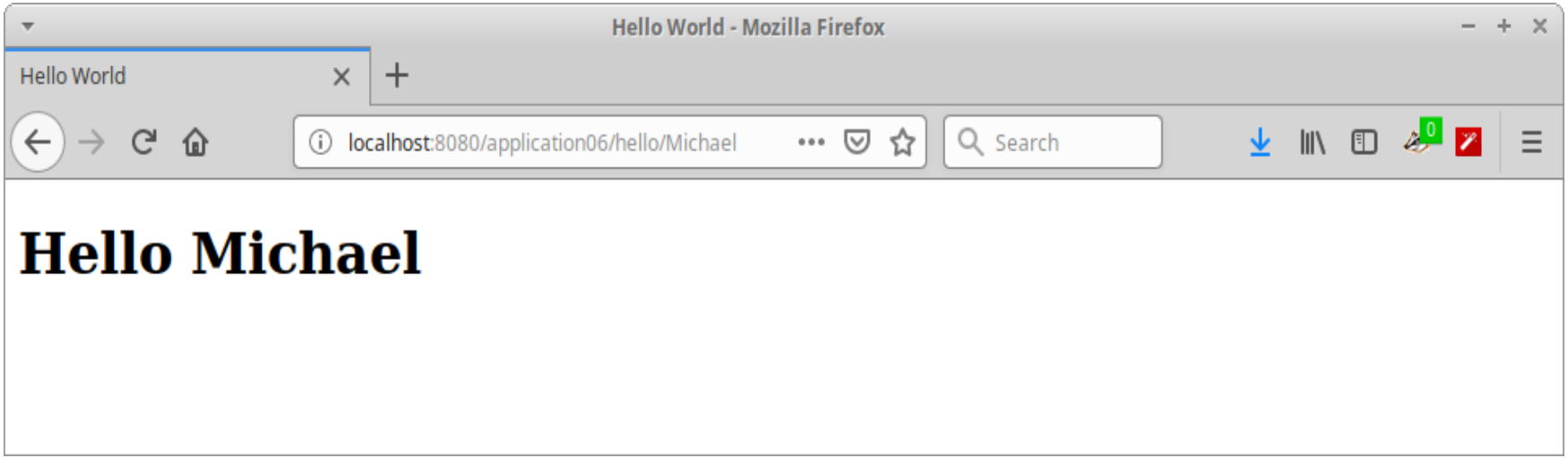
HelloWorld Controller

```
@Controller
public class HelloWorld {
		@GetMapping("/hello/{name}")
		public String Hello(@PathVariable String name, Model model) {
			model.addAttribute("name", name);
			return "helloView";
		}
}
```

helloView.jsp

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
   pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<title>Hello World</title>
</head>
<body>
   <h1>Hello ${name}</h1>
</body>
</html>
```

Result


Hello World - Mozilla Firefox
Hello World
localhost:8080/application06/hello/Michael
Search
Hello Michael

CS544 EA

Spring MVC: Request Mapping

Request Mapping

- @RequestMapping can be used to map an incoming HTTP request to a method
- The following shortcuts also exist:
  - @GetMapping
  - @PostMapping
  - @PutMapping
  - @DeleteMapping
  - @PatchMapping

Best to use these, because most controller methods should be mapped to only one HTTP method

RequestMapping by Path and Method

```
@Controller
public class CarController {
  @Resource
  private CarDao carDao;

  @RequestMapping(value="/cars", method=RequestMethod.GET)
  public String getAll(Model model) {
    model.addAttribute("cars", carDao.getAll());
    return "carList";
  }
}
```

Exactly the same

```
@Controller
public class CarController {
  @Resource
  private CarDao carDao;

  @GetMapping(value="/cars")
  public String getAll(Model model) {
    model.addAttribute("cars", carDao.getAll());
    return "carList";
  }
}
```

Multiple Methods per Controller

```
@Controller
public class CarController {
  @Resource
  private CarDao carDao;

  @GetMapping(value="/cars")
  public String getAll(Model model) {
    model.addAttribute("cars", carDao.getAll());
    return "carList";
  }

  @PostMapping(value="/cars")
  public String add(Car car) {
    carDao.add(car);
    return "redirect:/cars";
  }

  ...
}
```

Class Level Path Mapping

```
@Controller
public class CarController {

  @GetMapping(value="/cars/{id}")
  public String get(@PathVariable int id, Model model) {
    model.addAttribute("car", carDao.get(id));
    return "carDetail";
  }

  @PostMapping(value="/cars/{id}")
  public String update(Car car, @PathVariable int id) {
    carDao.update(id, car);
    return "redirect:/cars";
  }
}
```

Exactly the same

```
@Controller
@RequestMapping(value="/cars")
public class CarController {

  @RequestMapping(value="/{id}", method=RequestMethod.GET)
  public String get(@PathVariable int id, Model model) {
    model.addAttribute("car", carDao.get(id));
    return "carDetail";
  }

  @RequestMapping(value="/{id}", method=RequestMethod.POST)
  public String update(Car car, @PathVariable int id) {
    carDao.update(id, car);
    return "redirect:/cars";
  }
}
```

Parameters and Headers

```
@Controller
public class CarController {
  @Autowired
  private CarDao carDao;

  @GetMapping(value="/cars", params="myParam=myValue")
  public String getAllParam(Model model) {
    model.addAttribute("cars", carDao.getAll());
    return "carList";
  }

  @GetMapping(value="/cars", headers="myHeader=myValue")
  public String getAllHeader(Model model) {
    model.addAttribute("cars", carDao.getAll());
    return "carList";
  }

  ...

}
```

params="myParam" or
params="!myParam"
also possible

Only requests for "/cars?myParam=myvalue"
will be mapped here

Only Requests that have an http header:
myHeader: myValue
Will be mapped here

Produces and Consumes

Useful for web services

```
@RestController
public class WebService {
		@Resource
		private CarService carService;

		@GetMapping(value="/cars", produces="application/json")
		public List<Car> getAll() {
			return carService.getAll();
		}

		@PostMapping(value="/addCar", consumes="application/json")
		public void addCar(@RequestBody Car car) {
			carService.add(car);
		}
}
```

Mapping to non-Controllers

```
@Configuration
@EnableWebMvc
@ComponentScan("cs544")
public class WebConfig implements WebMvcConfigurer{
    @Override
    public void addViewControllers(ViewControllerRegistry registry) {
        registry.addViewController("/").setViewName("index");;
    }

    @Override
    public void addResourceHandlers(ResourceHandlerRegistry registry) {
        registry.addResourceHandler("/resources/**")
            .addResourceLocations("/public", "classpath:/static/")
            .setCachePeriod(31556926);
    }

    @Override
    public void configureDefaultServletHandling(DefaultServletHandlerConfigurer configurer) {
        configurer.enable();
    }
}
```

View Controller
lets the request go directly
to a JSP page

For static resources such as:
CSS, JS, HTML

Ask container's default Servlet
To handle static resources

XML

```
<?xml version="1.0" encoding="UTF-8"?>
<beans ...>

  <!-- Maps '/' requests to the 'home' view -->
  <mvc:view-controller path="/" view-name="index"/>

  <!-- Handles HTTP GET requests for /resources/** by efficiently serving
       up static resources in the ${webappRoot}/resources/ directory -->
 <mvc:resources mapping="/resources/**" location="/resources/" cache-period="31556926" />

  <!-- Lets us find resources (static and dynamic) through the web.xml -->
  <mvc:default-servlet-handler/>
</beans>
```

Same things now in XML

CS544 EA

Spring MVC: URI Templates

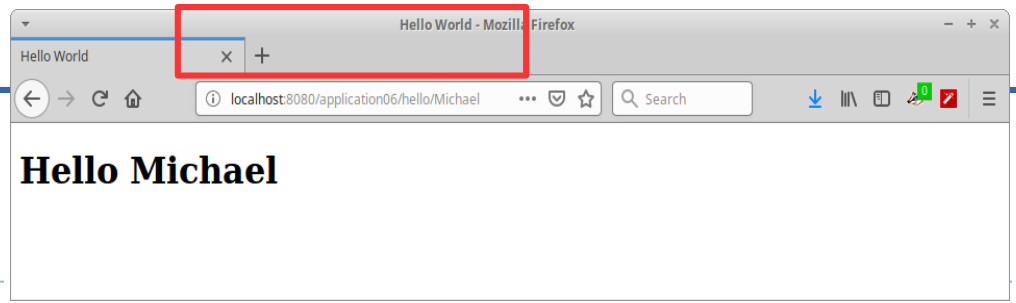
URI Templates

- A URI Template is a URI-like string, containing one or more variable names. When you substitute values for these variables, the template becomes a URI.

@PathVariable can be used to bind the variable to an input parameter

Parameter and variable name have to match

```
@Controller
public class HelloWorld {
        @GetMapping("/hello/{name}")
        public String Hello(@PathVariable String name, Model model) {
                model.addAttribute("name", name);
                return "helloView";
        }
}
```



Multiple Variables

```
@Controller
public class CustomerController {

  @RequestMapping(value="/customer/{customerId}/order/{orderId}")
  public String getOrder(@PathVariable long customerId,
                         @PathVariable long orderId, Model model) {
    // implementation ...
  }
}
```

Either directly on method level
or combined from class and method

```
@Controller
@RequestMapping(value="/customer/{customerId}")
public class CustomerController {

  @RequestMapping(value="/order/{orderId}")
  public String getOrder(@PathVariable long customerId,
                         @PathVariable long orderId, Model model) {
    // implementation ...
  }
}
```

Regex and Path Patterns

```
// Regular Expression Matching
@RequestMapping(value="/email/{user:\w+}@{host:\w+}.{tld:\w+}")
public void getInfo(@PathVariable String user,
                    @PathVariable String host, @PathVariable String tld) {
  // implementation ...
}
```

```
// Ant-Style path patterns
@RequestMapping(value="/customer/*/order/{orderId}")
public void getOrder(@PathVariable long orderId, Model model) {
  // implementation ...
}
```

CS544 EA

Spring MVC: Data Input

Request Input

We've seen how path variables can be used for input

GET /cars/1

```
@RequestMapping(value="/cars/{id}", method=RequestMethod.GET)
public String get(@PathVariable int id, Model model) {
  model.addAttribute("car", carDao.get(id));
  return "carDetail";
}
```

The same can of course be done with normal request params

Params specification is optional!

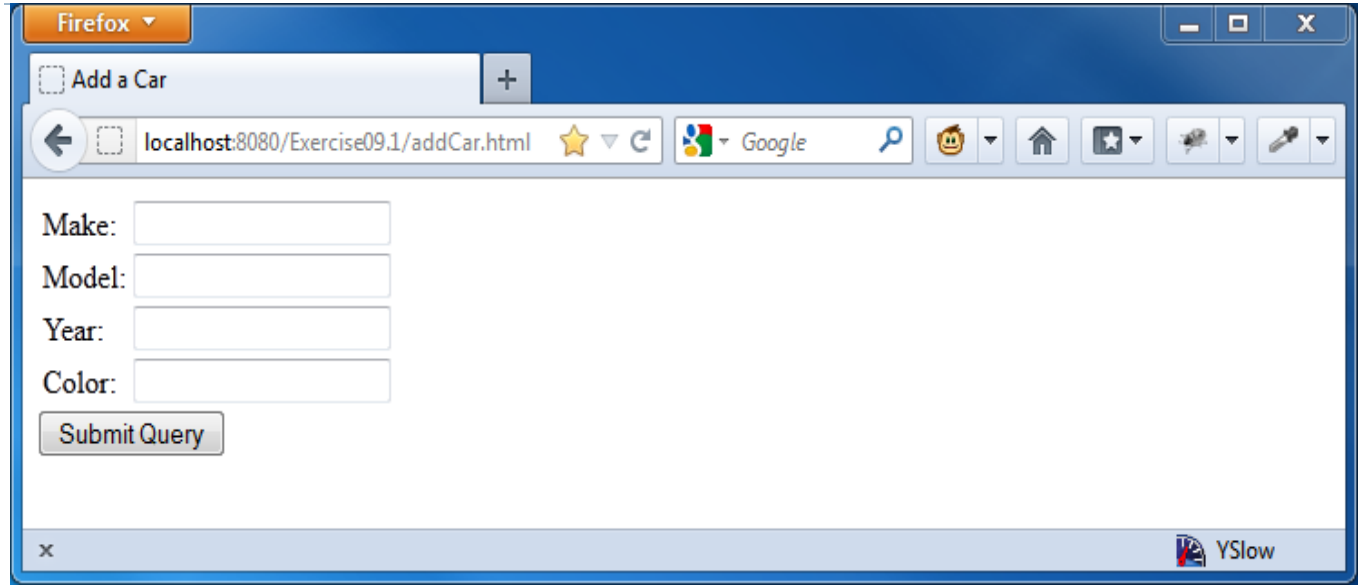
GET /cars?id=1

```
@RequestMapping(value="/cars", params="id", method=RequestMethod.GET)
public String getParam(@RequestParam int id, Model model) {
  model.addAttribute("car", carDao.get(id));
  return "carDetail";
}
```

@RequestParam is optional

Method param name has to match request param name

Many Parameters




```
public class Car {
  private int id;
  private String make;
  private String model;
  private int year;
  private String color;
```

Do Less Accomplish More

```
@RequestMapping(value="/cars", method=RequestMethod.POST)
public String addParams(String make, String model, int year, String color) {
  Car car = new Car(make, model, year, color);
  carDao.add(car);
  return "redirect:/cars";
}
```

You can receive the from params and then combine them into a Car

But you can also have Spring do all the work for you

```
@RequestMapping(value="/cars", method=RequestMethod.POST)
public String add(Car car) {
  carDao.add(car);
  return "redirect:/cars";
}
```

Car class does have to adhere to JavaBean standard

@RequestBody

- Web Services usually need to process the entire incoming request body (instead of parts)

```
@RestController
public class WebService {
        @Resource
        private CarService carService;

        @GetMapping(value="/cars", produces="application/json")
        public List<Car> getAll() {
                return carService.getAll();
        }

        @PostMapping(value="/addCar", consumes="application/json")
        public void addCar(@RequestBody Car car) {
                carService.add(car);
        }
}
```

Additional Parameters

- The following objects can be passed into Methods:

@PathVariable	HttpServletRequest
@RequestParam	HttpServletResponse
@RequestHeader	HttpSession
@RequestBody	InputStream
@RequestPart (file upload)	OutputStream
Map / Model / ModelMap	Reader
BindingResult / Errors	Writer
SessionStatus	Principal (security)
RedirectAttributes	Locale (internationalization)

- You can also define your own custom injectors
  - See Spring documentation

CS544 EA

Spring MVC: Data Output

Data Output

- There are two main ways to output data:
  - Render a view

    For web pages

    - Several ways to specify a view name
    - Providing it 'model' data to render
  - Output an object

    For web services

    - Use @ResponseBody on return type
    - Message converter transform to desired format
    - View name can be used to specify a transformer

Return StringView Name

```
@Bean
public ViewResolver viewResolver() {
     InternalResourceViewResolver bean = new InternalResourceViewResolver();

     bean.setViewClass(JstlView.class);
     bean.setPrefix("/WEB-INF/view/");
     bean.setSuffix(".jsp");

     return bean;
}
```

Many other types of ViewResolvers and Views are supported out of the box: Tiles, Velocity, PDF, Excel, Jasper Reports, XSLT

```
@GetMapping(value="/cars/{id}")
public String get(@PathVariable int id, Model model) {
     model.addAttribute("car", carDao.get(id));
     return "carDetail";
}
```

Model is an OUT param

Add data to model

Specify view

What is the name of our view? / Where will Spring MVC look for it?

View

```
<!DOCTYPE html>
<html>
<head><title>Add a Car</title></head>
<body>
  <form action="../cars/${car.id}" method="post">
  <table>
    <tr>
      <td>Make:</td>
      <td><input type="text" name="make" value="${car.make}" /> </td>
    </tr>
    <tr>
      <td>Model:</td>
      <td><input type="text" name="model" value="${car.model}" /> </td>
    </tr>
    <tr>
      <td>Year:</td>
      <td><input type="text" name="year" value="${car.year}" /> </td>
    </tr>
    <tr>
      <td>Color:</td>
      <td><input type="text" name="color" value="${car.color}" /> </td>
    </tr>
  </table>
  <input type="submit" value="update"/>
  </form>
  <form action="delete?carId=${car.id}" method="post">
    <button type="submit">Delete</button>
  </form>
</body>
</html>
```

ModelAndView

```
<!-- Resolves views selected for rendering by @Controllers to .jsp resources
    in the /WEB-INF/views directory -->
<bean class="org.springframework.web.servlet.view.InternalResourceViewResolver">
  <property name="viewClass" value="org.springframework.web.servlet.view.JstlView" />
  <property name="prefix" value="/WEB-INF/views/" />
  <property name="suffix" value=".jsp" />
</bean>
```

```
@RequestMapping(value="/cars", params="id", method=RequestMethod.GET)
public ModelAndView get(@RequestParam int id) {
  ModelAndView mav = new ModelAndView();
  mav.setViewName("carDetail");
  mav.addObject("car", carDao.get(id));
  return mav;
}
```

ModelAndView is old pre-spring 3

Use .setViewName() not .setView() !

@ModelAttribute

- Critical for form data
  - Especially if you want to show an empty form
  - The view(2 slides ago) can only display as empty form with the following code:

```
@GetMapping(value="/addCar")
public String get(@ModelAttribute("car") Car car) {
    return "addCar";
}
```

Places an empty Car Object in the Model with key “car”

Implicit View Name

If you've configured it

- You can omit (not specify) a view name
  - Convention: convert the request URL to view name

```
@Bean
public DefaultRequestToViewNameTranslator defaultRequestToViewNameTranslator() {
                return new DefaultRequestToViewNameTranslator();
}
```

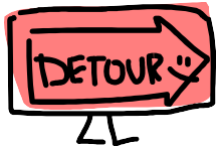
```
<!-- when using XML config -->
<bean class="org.springframework.web.servlet.view.DefaultRequestToViewNameTranslator" />
```

Can be declared in addition to normal ViewResolver

```
@GetMapping(value="/cars")
public void getAll(Model model) {
  model.addAttribute("cars", carDao.getAll());
}
```

No view name – what view is called?

Redirects



- **Redirects are important!**
  - After processing POST (input) → **always redirect**!
  - Known as POST/REDIRECT/GET **pattern**
  - Separation of concerns
  - No problem with refresh
  - No duplicate submissions
  - No problem with bookmarks

 See: http://en.wikipedia.org/wiki/Post/Redirect/Get

Redirects

```
@PostMapping(value="/cars")
public String add(Car car, Model model) {
  carDao.add(car);
  model.addAttribute("id", car.getId());
  return "redirect:/cars/{id}";
}
```

Redirect can contain URI Template

```
@PostMapping(value = "/list")
public RedirectView addItem(Item item)
{
  shoppingListService.addToList(item);
  return new RedirectView("list");
}
```

Pre Spring 3

CS544 EA

Spring MVC: Session

HttpSession

- If you want you can have direct access to the HttpSession, by **requesting it as a parameter**

Creates session if it didn't exist yet, or passes existing

```
@GetMapping(value="/cars/session")
public @ResponseBody String session(HttpSession session) {
  Enumeration<String> attributes = session.getAttributeNames();
  StringBuilder output = new StringBuilder();
  while (attributes.hasMoreElements()) {
    output.append(attributes.nextElement());
    output.append(" ");
  }
  return output.toString();
}
```

@SessionAttributes

- @Controller can specify @SessionAttributes
  - Intended for the duration of the controller
- Lists the names of **model attributes** that should be **stored in the session** (instead of request)
  - Once in the session will be inside model on each subsequent request

```
@Controller
@SessionAttributes(value={"cars", "currentId"})
public class CarController {
        ...
}
```

Removed on Completion

```
@Controller
@SessionAttributes(value={"cars", "currentId"})
public class CarController {

  ...

  @GetMapping(value="/cars/clear")
  public String clear(SessionStatus status) {
    // clears SessionAttributes specified on classlevel
    status.setComplete();
    return "redirect:/cars";
  }

  ...
}
```

Additional Parameter that allows you to clear (complete) the session

Storing / Retrieving

```
@Controller
@SessionAttributes(value={"cars", "currentId"})
public class CarController {

  @PostMapping(value="/cars")
  public String getCars(Model m) {
    m.addAttribute("cars", carDao.getCars());
    m.addAttribute("currentId", 1);
    return "redirect:/cars";
  }

  @GetMapping(value="/cars")
  public String viewCars(Model m) {
    List<Car> cars = (List<Car>)m.get("cars"); // just to demonstrate
    int num = m.get("currentId")
    return "cars";
  }
}
```

Method never explicitly uses HttpSession

Once added to the model it will be available to subsequent requests

Works if it is called after the method shown above

Spring MVC: Flash Attributes

Flash Attributes

- Flash attributes are a way for a request to store attributes intended for use in the next (single) request
  - Stored in the **session** for a very **short time**
  - Removed right away after first use
- Commonly used in combination with redirect
  - POST/REDIRECT/GET when the ‘get’ needs data
  - Also when there are validation errors on the POST

Specifying Flash Attributes

```
@Controller
public class CarController {

  @PostMapping(value="/cars")
  public String add(Car car, RedirectAttributes redirectAttrs) {
    carDao.add(car);
    String msg = "Added " + car.getMake() + " " + car.getModel();
    redirectAttrs.addFlashAttribute("message", msg);
    return "redirect:/cars";
  }
```

RedirectAttributes additional parameter

This URL will receive them

Make sure you use the **.addFlashAttribute()** method not the .addAttribute() method!

Receiving Flash Attributes

```
@Controller
public class CarController {

  @GetMapping(value="/cars")
  public String getAll(ModelMap model) {

    if (model.containsAttribute("message")) {
      System.out.println("Message: " + model.get("message"));
    } else {
      System.out.println("No Message");
    }

    model.addAttribute("cars", carDao.getAll());
    return "carList";
  }

}
```

The URL that receives

May be called from Flash or just normally called

If there are Flash attributes they are stored in the Model

Using a Flash Attribute

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<%@ page language="java" contentType="text/html; charset=UTF-8"
pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
<head>
  <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
  <title>Cars currently in the shop</title>
</head>
<body>
  <h1>Cars currently in the shop</h1>
  <table>
  <c:forEach var="car" items="${cars}">
    <tr>
      <td>${car.make}</td>
      <td>${car.model}</td>
      <td>${car.year}</td>
      <td>${car.color}</td>
      <td><a href="cars/${car.id}">edit</a></td>
    </tr>
  </c:forEach>
  </table>

  <c:if test="${not empty message}">
    <p>Message: <strong>${message}</strong></p
  </c:if>

  <a href="addCar.html"> Add a Car</a>
</body>
</html>
```

If Flash attribute passed in it will be available during View rendering

CS544 EA

Spring MVC: Exception Handling

Exception Handling

- @Controller level
  - Annotate methods with **@ExceptionHandler**
  - These will only handle exceptions that occur for methods inside this @Controller class

```
@Controller
public class CarController {

        ...

        @ExceptionHandler
        public String handle(IOException e) {
                // do something with the exception
                return "exception";
        }
}
```

Limiting to a Specific Exception

- There are **two ways to limit** the type of exception your exception handler handles

```
@Controller
public class CarController {

        ...

        @ExceptionHandler
        public String handle(IOException e) {
                // do something with the exception
                return "exception";
        }

        @ExceptionHandler({FileSystemException.class, RemoteException.class})
        public String handle(Exception e) {
                // do something with the exception
                return "exception";
        }
}
```

Specify the specific type on your method

Specify one or more types on the annotation

@ControllerAdvice

- To create @ExceptionHandlers that work for **multiple controllers** use @ControllerAdvice
  - AOP is used to add handlers to all controllers

```
@ControllerAdvice
public class ExampleAdvice {

        @ExceptionHandler
        public String handle(IOException e) {
                // do something with the exception
                return "exception";
        }
}
```

Some Controllers

- By default @ControllerAdvice applies to all controllers – but you can also specify which

```
@ControllerAdvice("cs544.controllers")
public class ExampleAdvice {
```

Apply to controllers in this package (or below)

```
@ControllerAdvice(annotations = RestController.class)
public class ExampleAdvice {
```

Apply to controllers that have this annotation

```
@ControllerAdvice(assignableTypes = {ControllerInterface.class, CarController.class})
public class ExampleAdvice {
```

Apply to controllers that have one of these types

Summary

- We've seen:
  - how to create the Spring context in a web container
  - Spring MVC Request Mapping
  - URI Templates
  - Data Input / Data Output
  - Session & Flash attributes
  - Exception Handling

CS544 EA

Spring Data

Spring Data

- We've been writing our own DAOs
  - Seems like we're repeating ourselves everytime
  - Spring can generate DAOs for us

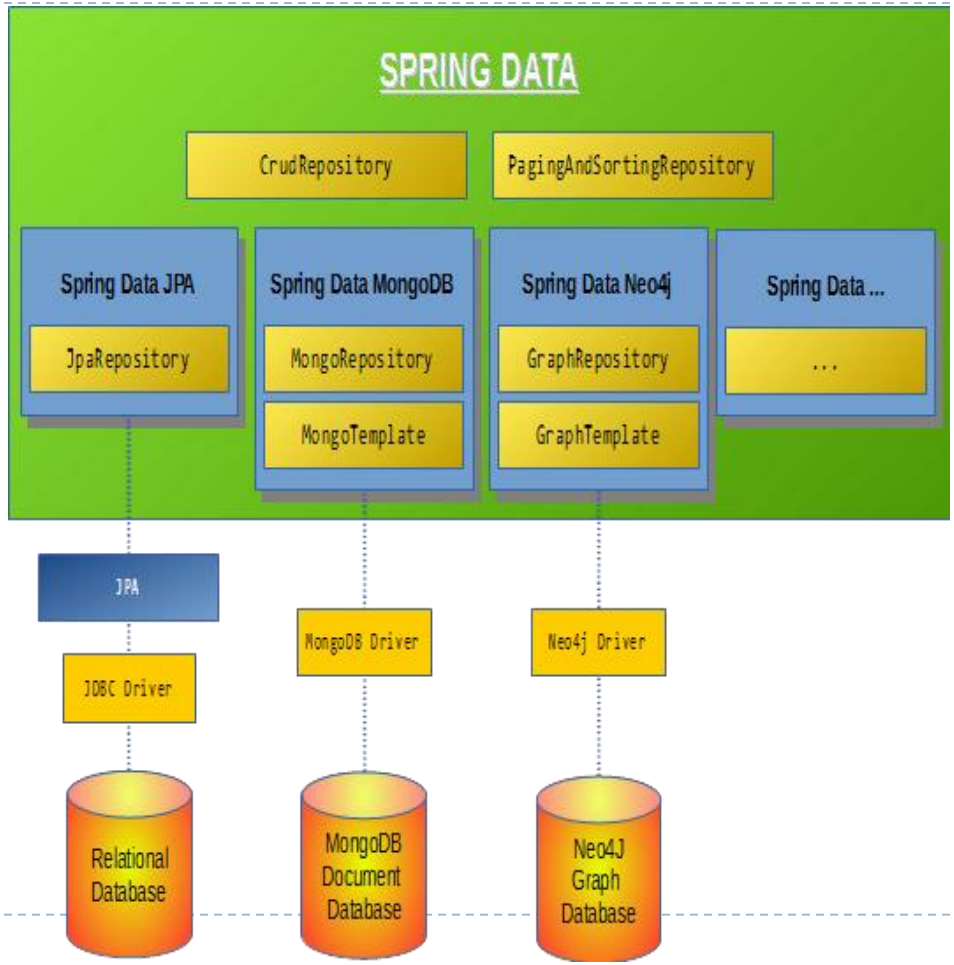
- We'll discuss how Spring Data works
  - Then look at finder methods

What is Spring Data

- Spring Data offers a flexible abstraction for working with data access frameworks
- Purpose is to **unify and ease the access** to different kinds of persistence stores
- Both relational DB systems and NoSQL data stores
- Addresses common difficulties developers face when working with databases in applications
- Spring Data is an umbrella project that provides you with easy to use data access technologies for all kinds of relational and non-relational DBs

Spring Data Modules

- Modules supports:
  - Templating
  - Object/Datastore mapping
  - Repository support
- Every repository offers:
  - CRUD operations
  - Finder Methods
  - Sorting and Pagination
- Spring data module repositories provide generic interfaces:
  - CrudRepository
  - PagingAndSortingRepository



Mapping

- JPA uses Object Relational Mapping
  - Spring Data extends this concept to NoSQL datastores

- All Spring Data modules provide an Object to data store mapping
  - Because these data store structures can be so different, there is no common API
  - Each type of data store has it's own set of annotations to map between objects and data store structures.

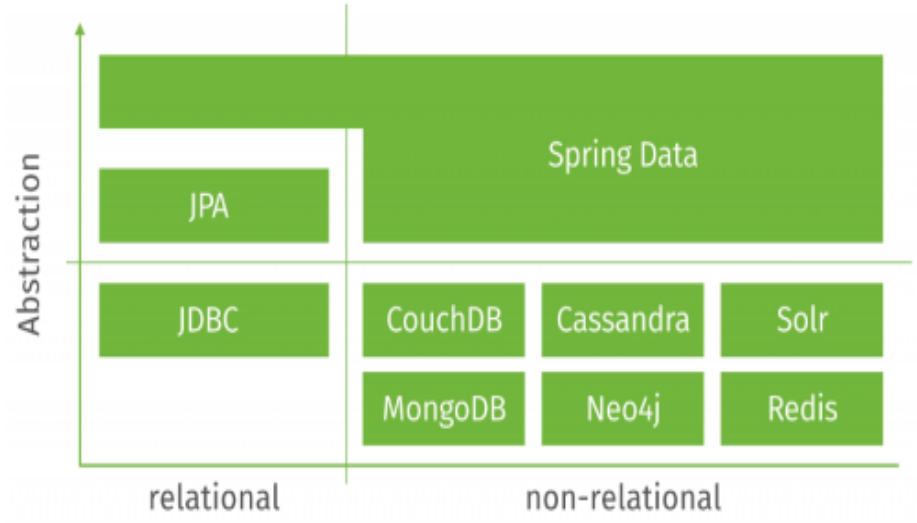
Mapping Examples

JPA	MongoDB	Neo4J

@Entity
@Table(name="USR")
public class User {
 	
@Id
private String id;
 	
@Column(name="fn")
private String name;
 	
private Date lastLogin;
 	
...
}
	@Document(
	collection="usr")
	public class User {
	
	@Id
	private String id;
	
	@Field("fn")
	private String name;
	
	private Date lastLogin;
	
	...
	}
		@NodeEntity
		public class User {
		
		@GraphId
		Long id;
		
		private String name;
		
		private Date lastLogin;
		
		...
		}

Spring Data Templates

- Each type of DB has its own template that:
  - Connection Management
  - CRUD operations, Queries, Map/Reduce jobs
  - Exception Translation to DataAccessExceptions
- No template for JPA
  - It provides most of these already (is an abstraction)



MongoDB Template Example

```java
@Configuration
public class MongoConfig extends AbstractMongoConfiguration {

    @Override
    protected String getDatabaseName() {
        return "test";
    }

    @Override
    public MongoClient mongoClient() {
        return new MongoClient("127.0.0.1", 27017);
    }

    @Override
    protected String getMappingBasePackage() {
        return "org.baeldung";
    }
}
```

```xml
<!-- Connection to MongoDB server -->
<mongo:mongo-client id="mongoClient" host="localhost" />
<mongo:db-factory id="mongoFactory" dbname="test"  mongo-ref="mongoClient"  />

<!-- MongoDB Template -->
<bean id="mongoTemplate" class="org.springframework.data.mongodb.core.MongoTemplate">
  <constructor-arg name="mongoDbFactory" ref="mongoDbFactory"/>
</bean>
```

Spring Data: JPA Example

DAOs aka Repositories

- DAOs (also known as repositories) are classes that implement CRUD operations and finder methods for a Entity

- Spring Data generates DAOs which:
  - Have the **same basic interface regardless of data store**
  - Provide a common way of querying for data
  - Provide a CRUD interface (if reasonable for DS)

CrudRepository Interface

- Spring Data JPA is instructed to scan package
- DAOs are generated for any **interface** that extends **Repository<T, id>**
  - JpaRepository extends PagingAndSortingRepostirory extends CrudRepository extends Repository

count()	saveAll(Iterable<S> entities)
exists(ID id)	save(S entity)
findAll()	delete(ID id)
findAllById(Iterable<ID> ids)	delete(T entity)
findById(id)	deleteAll(Iterable<? extends T> entities)
	deleteAll()

JPA Repository methods

Example JPA Config

Java Config

```
@Configuration
@ComponentScan("edu.mum.cs544")
@EnableJpaRepositories("edu.mum.cs544.dao")
public class Config {

}
```

Which package to scan for repository interfaces

Same thing

XML Config

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
                xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                xmlns:jpa="http://www.springframework.org/schema/data/jpa"
                xsi:schemaLocation="
    http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans.xsd
    http://www.springframework.org/schema/data/jpa
    http://www.springframework.org/schema/data/jpa/spring-jpa.xsd">

  <jpa:repositories  base-package="edu.mum.cs544.dao"/>
```

Remember to add the namespace

Which package to scan for repository interfaces

Example Code

```
public interface ContactDao extends JpaRepository<Contact, Long> {
}
```

Use an interface to declare that there should be a ContactDao

Spring Data will automatically generate an implementation

```
@Service
@Transactional
public class ContactService {

    @Autowired
    private ContactDao contactDao;

                ...
}
```

Generated DAO can be used like any other DAO before this

CS544 EA

Spring Data: Finder Methods

Finder Methods

- The basic provided methods may be enough for a small application
  - Real applications often need more specific methods
- You can **add a finder method**
  - by using the **name of the property**

```
public interface ContactDao extends JpaRepository<Contact, Long> {
        List<Contact> findByName(String name);
}
```

Contact Entity has a name property

Finder Method Prefixes

- The following prefixes start a finder method:
  - findBy, readBy, queryBy, getBy, and countBy
- They can then contain further expressions like:
  - Distinct, Top10, First5

```
public interface ContactDao extends JpaRepository<Contact, Long> {
        List<Contact> findDistinctContactByLastname(String lastname);
        List<Contact> findContactDistinctByLastname(String lastname);

        Page<Contact> queryFirst10ByLastname(String lastname, Pageable pageable);
        Slice<Contact> findTop3ByLastname(String lastname, Pageable pageable);
        List<Contact> findFirst10ByLastname(String lastname, Sort sort);
        List<Contact> findTop10ByLastname(String lastname, Pageable pageable);
}
```

Criteria

- The first **By** acts as a delimiter to indicate the start of the actual criteria

```
List<Contact> findDistinctContactByLastname(String lastname);
```

- You can combine criteria with **and** and **or**

```
List<Person> findByEmailAddressAndLastname(EmailAddress emailAddress, String lastname);
List<Person> findByLastnameOrFirstname(String lastname, String firstname);
```

- Typical operators are also supported (see next)

List of Keywords

And	After	Containing
Or	Before	OrderBy
Is,Equals	IsNull	Not
Between	IsNotNull, NotNull	In
LessThan	Like	NotIn
LessThanEqual	NotLike	True
GreaterThan	StartingWith	False
GreaterThanEqual	EndingWith	IgnoreCase (AllIgnoreCase)

Examples:

```
findByFirstname,findByFirstnameIs,findByFirstnameEquals // all the same
findByStartDateBetween // expects two parameters
findByAgeLessThanEqual
findByAgeIsNotNull, findByAgeNotNull // same
findByFirstnameLike // first parameter matched as Like (including %'s)
findByAgeIn(Collection<Age> ages) // or subclass of collection
```

IgnoreCase

- You can ignore case for a single property

```
List<Person> findByFirstNameAndLastnameIgnoreCase(String firstname, String lastname);
```

Only LastName is case insensitive

- Or ignore case for all properties

```
List<Person> findByLastnameAndFirstnameAllIgnoreCase(String lastname, String firstname);
```

PagingAndSortingRepository

- The PagingAndSortingRepository interface adds methods to sort and paginate entities
  - findAll(Pageable pageable), findAll(Sort sort)
- Enables you to use the orderBy keyword

```
public List<Student> findByLastNameOrderByLastNameAsc(String lastName);
public List<Student> findByLastNameOrderByLastNameDesc(String lastName);
```

- You can request a page using PageRequest

```
PageRequest pageRequest = new PageRequest(0, 10);
Page<Student> page = studentDao.findAll(pageRequest);
Slice<Student> slice = studentDao.findByFirstName("Lisa", pageRequest);
```

Difference Between Page and Slice

- Page extends Slice
  - Slice only knows if there is more
  - Not how much more

- Page executes a select count to find out

Limiting Results

- The keywords **first** and **top** will limit the amount of rows that will be returned
  - Optional numeric value can be added

```
User  findFirstByOrderByLastnameAsc();
User  findTopByOrderByAgeDesc();
Page<User>  queryFirst10ByLastname(String lastname, Pageable pageable);
Slice<User>  findTop3ByLastname(String lastname, Pageable pageable);
List<User>  findFirst10ByLastname(String lastname, Sort sort);
List<User>  findTop10ByLastname(String lastname, Pageable pageable);
```

- If pagination or slicing is applied it is within the limited result

CS544 EA

Spring Data: Advanced Methods

Property Expressions

- Similar to HQL you can traverse properties

```
List<Student> findByAddressZipCode(ZipCode zipCode);// x.address.zipCode
```

- What if a student has a AddressZipCode property?

```
List<Student> findByAddress_ZipCode(ZipCode zipCode);// x.address.zipCode
```

Optionally underscores can be used to separate properties

Java Methods should use CamelCase there should not be any conflicts caused by this

@Query

- What if you don't like typing long names or need to have a complicated query?

```
public interface UserRepository extends JpaRepository<User, Long>{
        @Query("Select u from User u where u.emailAddress = ?1")
        User findByEmailAddrss(String emailAddress);
}
```

- Or want to write SQL instead of HQL?

```
public interface UserRepository extends JpaRepository<User, Long>{
        @Query(value = "SELECT * FROM USERS WHERE EMAIL_ADDRESS = ?0", nativeQuery = true)
        User findByEmailAddrss(String emailAddress);
}
```

Custom Functionality

- You can also write your own custom methods
  - Allowing you to do whatever you want

Step1: create an interface

```
public interface UserDaoCustom {
        void someCustomMethod(User user);
}
```

Step2: create an implementation

```
public class UserDaoImpl implements UserDaoCustom {
        void someCustomMethod(User user) {
                // your custom implementation
        }
}
```

Step3: add the interface to where you want to add the functionality

```
public interface UserRepository extends JpaRepository<User, Long>, UserDaoCustom {
}
```

Summary

- Spring data allows you to generate DAOs independent of the type of data store you use
- The default methods of the generated DAOs are good for simple applications
- It's easy to add additional Finder methods using nothing more than the method names
- Custom queries or complete method implementations can easily be added