JAVA DAVE ENVIRONMENT

Spring MVC, ExtJS, Hibernate & More…

Java Boot Camp, August, 2017.

Dr. Kishore Biswas (Forrest/柯修)

PhD. Artificial Intelligence—Natural Language Processing.

CanWay IT Training ®.

CONTENTS

1) Spring example & practice
2) Spring MVC explained
3) Ext JS code example & practice
4) Introduction to Hibernate

1) Spring Example & Practice

SPRING - ANNOTATION BASED CONFIGURATION

- The another way of configuring the dependency injection using annotations. So instead of using XML to describe a bean wiring, we can move the bean configuration into the component class itself by using annotations on the relevant class, method, or field declaration.

- Annotation injection is performed before XML injection. Thus, the latter configuration will override the former for properties wired through both approaches.

- Annotation wiring is not turned on in the Spring container by default. So, before we can use annotation-based wiring, we will need to enable it in our Spring configuration file.

DIFFERENT ANNOTATION

1	**@Required** The @Required annotation applies to bean property setter methods.
2	**@Autowired** The @Autowired annotation can apply to bean property setter methods, non-setter methods, constructor and properties.
3	**@Qualifier** The @Qualifier annotation along with @Autowired can be used to remove the confusion by specifiying which exact bean will be wired.
4	**JSR-250 Annotations** Spring supports JSR-250 based annotations which include @Resource, @PostConstruct and @PreDestroy annotations.

AUTOWIRED ANNOTATION

- @Autowired on Setter Methods
- You can use @Autowired annotation on setter methods to get rid of the <property> element in XML configuration file. When Spring finds an @Autowired annotation used with setter methods, it tries to perform byType autowiring on the method.

- @Autowired on Properties
- We can use **@Autowired** annotation on properties to get rid of the setter methods. When we pass the values of autowired properties using <property>, Spring will automatically assign those properties with the passed values or references.

SPRING @QUALIFIER ANNOTATION

- There may be a situation when you create more than one bean of the same type and want to wire only one of them with a property. In such cases, you can use the @Qualifier annotation along with @Autowired to remove the confusion by specifying which exact bean will be wired.

EVENT HANDLING IN SPRING

- The core of Spring is the ApplicationContext, which manages the complete life cycle of the beans. The ApplicationContext publishes certain types of events when loading the beans. For example, a ContextStartedEvent is published when the context is started and ContextStoppedEvent is published when the context is stopped.

- Event handling in the ApplicationContext is provided through the ApplicationEvent class and ApplicationListener interface. Hence, if a bean implements the ApplicationListener, then every time an ApplicationEvent gets published to the ApplicationContext, that bean is notified.

SPRING'S STANDARD EVENTS

1	**ContextRefreshedEvent** This event is published when the *ApplicationContext* is either initialized or refreshed. This can also be raised using the refresh() method on the *ConfigurableApplicationContext* interface.
2	**ContextStartedEvent** This event is published when the *ApplicationContext* is started using the start() method on the *ConfigurableApplicationContext* interface. You can poll your database or you can restart any stopped application after receiving this event.
3	**ContextStoppedEvent** This event is published when the *ApplicationContext* is stopped using the stop() method on the *ConfigurableApplicationContext* interface. You can do required housekeep work after receiving this event.
4	**ContextClosedEvent** This event is published when the *ApplicationContext* is closed using the close() method on the *ConfigurableApplicationContext* interface. A closed context reaches its end of life; it cannot be refreshed or restarted.
5	**RequestHandledEvent** This is a web-specific event telling all beans that an HTTP request has been serviced.

Spring's event handling is single-threaded so if an event is published, until and unless all the receivers get the message, the processes are blocked and the flow will not continue. Hence, care should be taken when designing your application if the event handling is to be used.

Listening to Context Events
To listen to a context event, a bean should implement the *ApplicationListener* interface which has just one method **onApplicationEvent()**.

AOP WITH SPRING FRAMEWORK

- One of the key components of Spring Framework is the Aspect oriented programming (AOP) framework. Aspect-Oriented Programming entails breaking down program logic into distinct parts called so-called concerns. The functions that span multiple points of an application are called cross-cutting concerns and these cross-cutting concerns are conceptually separate from the application's business logic. There are various common good examples of aspects like logging, auditing, declarative transactions, security, caching, etc.

- The key unit of modularity in OOP is the class, whereas in AOP the unit of modularity is the aspect. Dependency Injection helps you decouple your application objects from each other and AOP helps you decouple cross-cutting concerns from the objects that they affect. AOP is like triggers in programming languages such as Perl, .NET, Java, and others.

- Spring AOP module provides interceptors to intercept an application. For example, when a method is executed, you can add extra functionality before or after the method execution.

AOP CONCEPTS

Sr.No	Terms & Description
1	**Aspect** This is a module which has a set of APIs providing cross-cutting requirements. For example, a logging module would be called AOP aspect for logging. An application can have any number of aspects depending on the requirement.
2	**Join point** This represents a point in your application where you can plug-in the AOP aspect. You can also say, it is the actual place in the application where an action will be taken using Spring AOP framework.
3	**Advice** This is the actual action to be taken either before or after the method execution. This is an actual piece of code that is invoked during the program execution by Spring AOP framework.
4	**Pointcut** This is a set of one or more join points where an advice should be executed. You can specify pointcuts using expressions or patterns as we will see in our AOP examples.
5	**Introduction** An introduction allows you to add new methods or attributes to the existing classes.
6	**Target object** The object being advised by one or more aspects. This object will always be a proxied object, also referred to as the advised object.
7	**Weaving** Weaving is the process of linking aspects with other application types or objects to create an advised object. This can be done at compile time, load time, or at runtime.

2) Spring MVC

SPRING - MVC FRAMEWORK

- The Spring Web MVC framework provides Model-View-Controller (MVC) architecture and ready components that can be used to develop flexible and loosely coupled web applications. The MVC pattern results in separating the different aspects of the application (input logic, business logic, and UI logic), while providing a loose coupling between these elements.

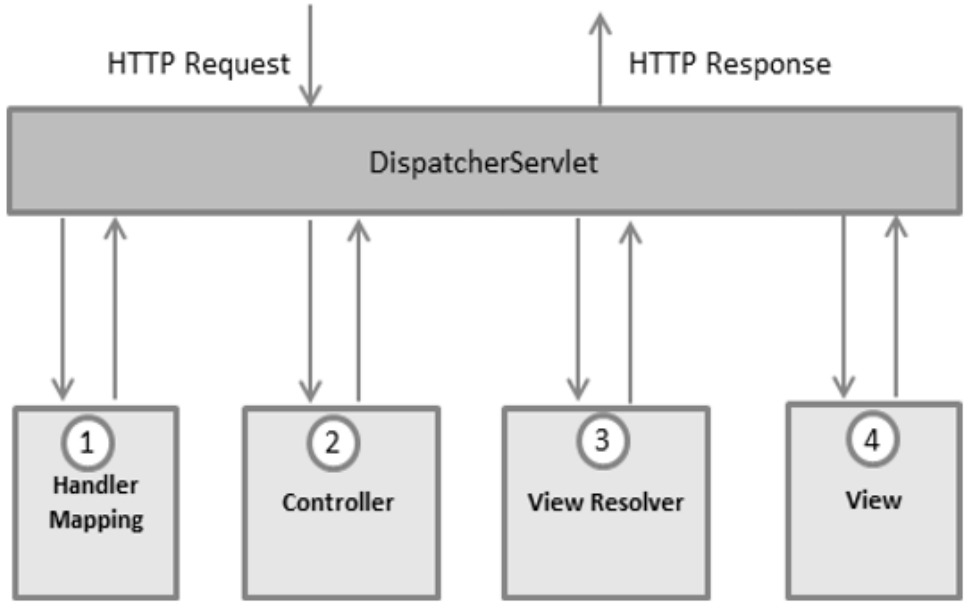
- The Model encapsulates the application data and in general they will consist of POJO.

- The View is responsible for rendering the model data and in general it generates HTML output that the client's browser can interpret.

- The Controller is responsible for processing user requests and building an appropriate model and passes it to the view for rendering.

THE DISPATCHERSERVLET

The Spring Web model-view-controller (MVC) framework is designed around a DispatcherServlet that handles all the HTTP requests and responses.



1) After receiving an HTTP request, DispatcherServlet consults the HandlerMapping to call the appropriate Controller.

2) The Controller takes the request and calls the appropriate service methods based on used GET or POST method. The service method will set model data based on defined business logic and returns view name to the DispatcherServlet.

3) The DispatcherServlet will take help from ViewResolver to pickup the defined view for the request.

4) Once view is finalized, The DispatcherServlet passes the model data to the view which is finally rendered on the browser.

SPRING MVC FORM HANDLING EXAMPLE

- Steps
- 1 Create a Dynamic Web Project with a name HelloWeb and create a package under the src folder in the created project.
- 2 Drag and drop below mentioned Spring and other libraries into the folder WebContent/WEB-INF/lib.
- 3 Create a Java classes Student and StudentController under the com.tutorialspoint package.
- 4 Create Spring configuration files Web.xml and HelloWeb-servlet.xml under the WebContent/WEB-INF folder.
- 5 Create a sub-folder with a name jsp under the WebContent/WEB-INF folder. Create a view files student.jsp and result.jsp under this sub-folder.
- 6 The final step is to create the content of all the source and configuration files and export the application

CONFIGURATION

- You need to map requests that you want the DispatcherServlet to handle, by using a URL mapping in the web.xml file.

```
<web-app id = "WebApp_ID" version = "2.4"
   xmlns = "http://java.sun.com/xml/ns/j2ee"
   xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation = "http://java.sun.com/xml/ns/j2ee
   http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd">

   <display-name>Spring MVC Application</display-name>

   <servlet>
      <servlet-name>HelloWeb</servlet-name>
      <servlet-class>
         org.springframework.web.servlet.DispatcherServlet
      </servlet-class>
      <load-on-startup>1</load-on-startup>
   </servlet>

   <servlet-mapping>
      <servlet-name>HelloWeb</servlet-name>
      <url-pattern>*.jsp</url-pattern>
   </servlet-mapping>

</web-app>
```

The web.xml file will be kept in the WebContent/WEB-INF directory of your web application. Upon initialization of HelloWeb DispatcherServlet, the framework will try to load the application context from a file named [servlet-name]-servlet.xml located in the application's WebContent/WEB-INFdirectory. In this case, our file will be HelloWebservlet.xml.

Next, <servlet-mapping> tag indicates what URLs will be handled by which DispatcherServlet. Here all the HTTP requests ending with .jsp will be handled by the HelloWeb DispatcherServlet.

If you do not want to go with default filename as [servlet-name]-servlet.xml and default location as WebContent/WEB-INF, you can customize this file name and location by adding the servlet listener ContextLoaderListener in your web.xml file

WEB.XML

```
<web-app...>

   <!-------- DispatcherServlet definition goes here----->
   ....
   <context-param>
      <param-name>contextConfigLocation</param-name>
      <param-value>/WEB-INF/HelloWeb-servlet.xml</param-value>
   </context-param>

   <listener>
      <listener-class>
         org.springframework.web.context.ContextLoaderListener
      </listener-class>
   </listener>

</web-app>
```

- commons-logging-x.y.z.jar
- org.springframework.asm-x.y.z.jar
- org.springframework.beans-x.y.z.jar
- org.springframework.context-x.y.z.jar
- org.springframework.core-x.y.z.jar
- org.springframework.expression-x.y.z.jar
- org.springframework.web.servlet-x.y.z.jar
- org.springframework.web-x.y.z.jar
- spring-web.jar

HELLOWEB-SERVLET.XML FILE,

```
<beans xmlns = "http://www.springframework.org/schema/beans"
   xmlns:context = "http://www.springframework.org/schema/context"
   xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation = "http://www.springframework.org/schema/beans
   http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
   http://www.springframework.org/schema/context
   http://www.springframework.org/schema/context/spring-context-3.0.xsd">

   <context:component-scan base-package = "com.tutorialspoint" />

   <bean class = "org.springframework.web.servlet.view.InternalResourceViewResolver'
      <property name = "prefix" value = "/WEB-INF/jsp/" />
      <property name = "suffix" value = ".jsp" />
   </bean>

</beans>
```

IMPORTANT POINTS ABOUT HELLOWEB-SERVLET.XML FILE

- The [servlet-name]-servlet.xml file will be used to create the beans defined, overriding the definitions of any beans defined with the same name in the global scope.

- The <context:component-scan...> tag will be use to activate Spring MVC annotation scanning capability which allows to make use of annotations like @Controller and @RequestMapping etc.

- The InternalResourceViewResolver will have rules defined to resolve the view names. As per the above defined rule, a logical view named hello is delegated to a view implementation located at /WEB-INF/jsp/hello.jsp .

- The following section will show you how to create your actual components, i.e., Controller, Model, and View.

DEFINING A CONTROLLER

- The DispatcherServlet delegates the request to the controllers to execute the functionality specific to it. The @Controllerannotation indicates that a particular class serves the role of a controller. The @RequestMapping annotation is used to map a URL to either an entire class or a particular handler method.

```
@Controller
public class HelloController {
   @RequestMapping(value = "/hello", method = RequestMethod.GET)
   public String printHello(ModelMap model) {
      model.addAttribute("message", "Hello Spring MVC Framework!");
      return "hello";
   }
}
```

CONTROLLER CONFIGURATION

- The @Controller annotation defines the class as a Spring MVC controller. Here, the first usage of @RequestMapping indicates that all handling methods on this controller are relative to the /hello path. Next annotation@RequestMapping(method = RequestMethod.GET) is used to declare theprintHello() method as the controller's default service method to handle HTTP GET request. You can define another method to handle any POST request at the same URL.

- You can write the above controller in another form where you can add additional attributes in @RequestMapping.

```java
@Controller
public class HelloController {
   @RequestMapping(value = "/hello", method = RequestMethod.GET)
   public String printHello(ModelMap model) {
      model.addAttribute("message", "Hello Spring MVC Framework!");
      return "hello";
   }
}
```

The value attribute indicates the URL to which the handler method is mapped and the method attribute defines the service method to handle HTTP GET request. The following important points are to be noted about the controller defined above −

You will define required business logic inside a service method. You can call another method inside this method as per requirement.
Based on the business logic defined, you will create a model within this method. You can use setter different model attributes and these attributes will be accessed by the view to present the final result. This example creates a model with its attribute "message".
A defined service method can return a String, which contains the name of the view to be used to render the model.

Creating JSP Views

- Spring MVC supports many types of views for different presentation technologies. These include - JSPs, HTML, PDF, Excel worksheets, XML, Velocity templates, XSLT, JSON, Atom and RSS feeds, JasperReports, etc. But most commonly we use JSP templates written with JSTL.

3) Ext JS Code Example & Practice

FEATURES

- These are the highlighted features of Ext JS

- Customizable UI widgets with collection of rich UI such as Grids, pivot grids, forms, charts, trees.
- Code compatibility of new versions with the older one.
- A flexible layout manager helps to organize the display of data and content across multiple browsers, devices, and screen sizes.
- Advance data package decouples the UI widgets from the data layer. The data package allows client-side collection of data using highly functional models that enable features such as sorting and filtering.
- It is protocol agnostic, and can access data from any back-end source.
- Customizable Themes Ext JS widgets are available in multiple out-of-the-box themes that are consistent across platforms.

WHY

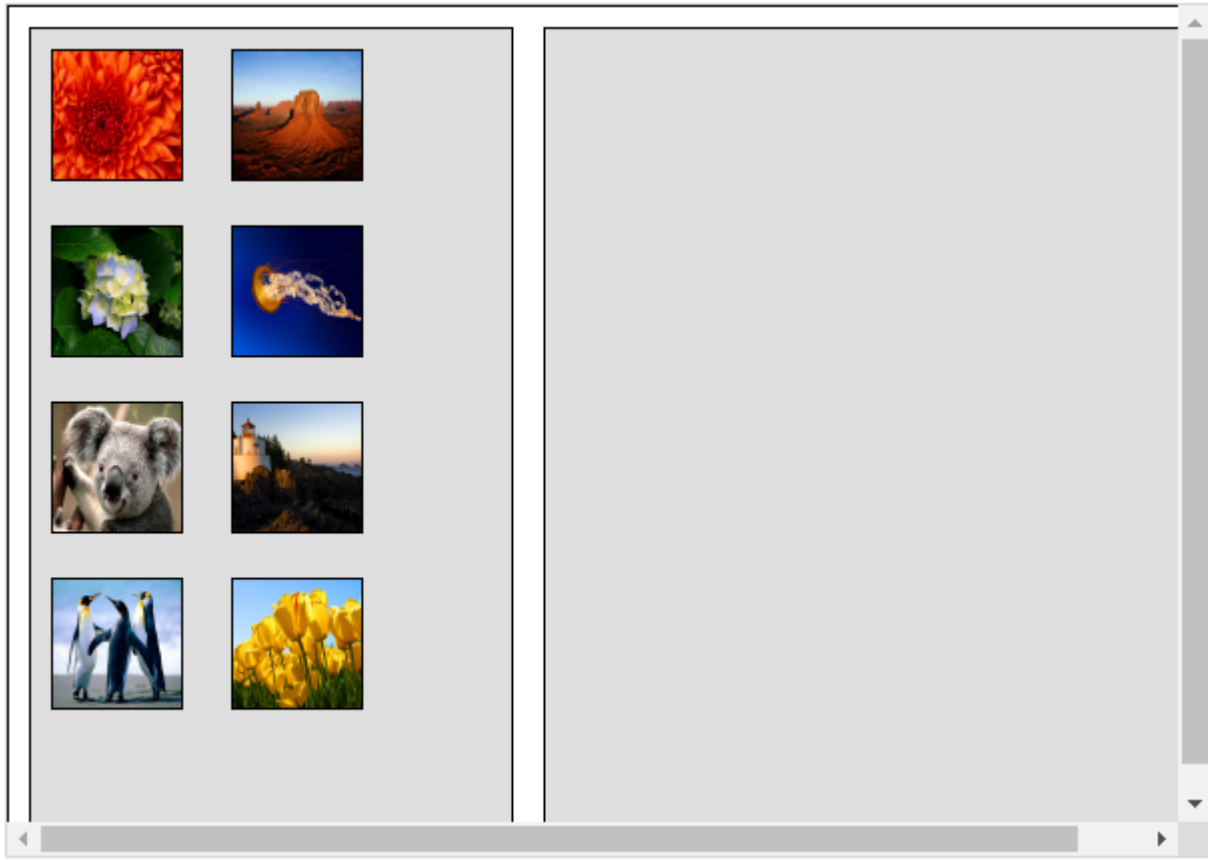
- This is really what matters (MVC)
  - Easy Client-side data modeling
    - Relational Models
  - Simple to use GUI widgets
  - Full robustness of EcmaScript control

- Benefits
- Sencha Ext JS is the leading standard for business-grade web application development. Ext JS provides the tools necessary to build robust applications for the desktop and tablets.
- Streamlines cross-platform development across desktops, tablets, and smartphones — for both modern and legacy browsers.
- Increases the productivity of development teams by integrating into enterprise development environments via IDE plugins.
- Reduces the cost of web application development.
- Empowers teams to create apps with a compelling user experience.
- It has set of widgets for making UI powerful and easy.
- It follows MVC architecture so highly readable code.

EXT JS PRACTICE

EXT JS PRACTICE

- Code in Ext JS to implement drag and drop windows



TOOLS

These are the tools provided by sencha used for Ext JS application development mainly for production level.

Sencha Cmd

- Sencha CMD is a tool which provides the features of Ext JS code minification, scaffolding, production build generation.

Sencha IDE Plugins

- Sencha IDE plugins which are integrates Sencha frameworks into IntelliJ, WebStorm IDEs. Which helps in improving developer's productivity by providing features such as code completion, code inspection, code navigation, code generation, code refactoring, template creation, and spell-checking etc.

Sencha Inspector

- Sencha Inspector is a debugging tool which helps debugger to debug any issue while development.

MVC

- Why is MVC so important?
  - In this case, it is because it is 100%, agent-based, client side code
  - This means typical MVC on the server is not needed
    - Good or Bad? Design decision

EXT.JS - NAMING CONVENTION...

- Naming convention is a set of rule to be followed for identifiers.
- It makes code more readable and understandable to the other programmers as well.
- Naming convention in Ext JS follows the standard JavaScript convention which is not mandatory but a good practice to follow.
- It should follows camel case syntax for naming the class, method, variable and properties.

- If name is combined with two words, second word will start with uppercase letter always e.g. doLayout(), StudentForm, firstName etc.

EXT.JS - NAMING CONVENTION

Name	Convention
Class Name	It should start with uppercase letter and followed by camel case E.g. StudentClass
Method Name	It should start with lowercase letter and followed by camel case E.g. doLayout()
Variable Name	It should start with lowercase letter and followed by camel case E.g. firstName
Constant Name	It should be in uppercase only E.g. COUNT, MAX_VALUE
Property Name	It should start with lowercase letter and followed by camel case e.g. enableColumnResize = true

EXT.JS - ARCHITECTURE

- Ext JS follows MVC/ MVVM architecture.
- MVC – Model View Controller architecture (version 4)
- MVVM – Model View Viewmodel (version 5)
- This architecture is not mandatory for the program but it is best practice to follow this structure to make your code highly maintainable and organized.

```
<!DOCTYPE html>
<html>
  <head>
    <link href="https://cdnjs.cloudflare.com/ajax/libs/extjs/6.0.0/classic/theme-classic/resources/theme-classic-all.css" rel="stylesheet" />
    <script type="text/javascript" src="https://cdnjs.cloudflare.com/ajax/libs/extjs/6.0.0/ext-all.js"></script>
    <script type="text/javascript">
      Ext.onReady(function() {
      Ext.create('Ext.Panel', {
        renderTo: 'helloWorldPanel',
        height: 200,
        width: 600,
        title: 'Hello world',
        html: 'First Ext JS Hello World Program'
        });
      });
    </script>
  </head>
  <body>
    <div id="helloWorldPanel" />
  </body>
</html>
```

EXPLANATION

- Ext.onReady() method will be called once the Ext JS is ready to render the Ext JS elements.
- Ext.create() method is used to create object in Ext JS here we are creating an object of simple panel class Ext.Panel.
- Ext.Panel is the predefined class in Ext JS for creating a panel.
- Every Ext JS class has different properties to perform some basic functionalities.
- Ext.Panel class has various properties as:
- renderTo is the element where this panel has to be render. 'helloWorldPanel' is the div id in Index.html file.
- Height and width properties are for giving custom size of the panel.
- Title property is to provide the title to the panel.
- Html property is the html content to be shown in the panel.

EXT.JS - CLASS SYSTEM

Ext JS is a JavaScript framework which has functionalities of object oriented programming. Ext is the namespace which encapsulates all the classes in Ext JS.

Defining a class in Ext JS

- Ext provides more than 300 classes which we can use for various functionalities.
- Ext.define() is used for defining classes in Ext JS.

SYNTAX:

Ext.define(class name, class members/properties, callback function);

- Class name is the name of class according to app structure e.g. appName.folderName.ClassName studentApp.view.StudentView.

- Class properties/members - which define the behavior of class.

- Callback function is optional. It is called when the class has loaded properly.

EXAMPLE OF EXT JS CLASS DEFINITION

```
Ext.define(studentApp.view.StudentDeatilsGrid, {
   extend : 'Ext.grid.GridPanel',
   id : 'studentsDetailsGrid',
   store : 'StudentsDetailsGridStore',
   renderTo : 'studentsDetailsRenderDiv',
   layout : 'fit',
   columns : [{
      text : 'Student Name',
      dataIndex : 'studentName'
   },{
      text : 'ID',
      dataIndex : 'studentId'
   },{
      text : 'Department',
      dataIndex : 'department'
   }]
});
```

CREATING OBJECTS

Like other OOPS based languages we can create objects in Ext JS as well. Different ways of creating objects in Ext JS:

1) Using new keyword:

```
var studentObject = new student();
studentObject.getStudentName();
```

2) Using Ext.create():

```
Ext.create('Ext.Panel', {
    renderTo : 'helloWorldPanel',
    height : 100,
    width : 100,
    title : 'Hello world',
    html :              'First Ext JS Hello World Program'
});
```

USING INHERITANCE IN EXT JS

Inheritance is the principle of using functionality defined in class A into class B. In Ext JS inheritance can be done using two methods- Ext.extend:

```
Ext.define(studentApp.view.StudentDetailsGrid, {
    extend : 'Ext.grid.GridPanel',
    ...
});
```

Here our custom class StudentDetailsGrid is using basic features of Ext JS class GridPanel.

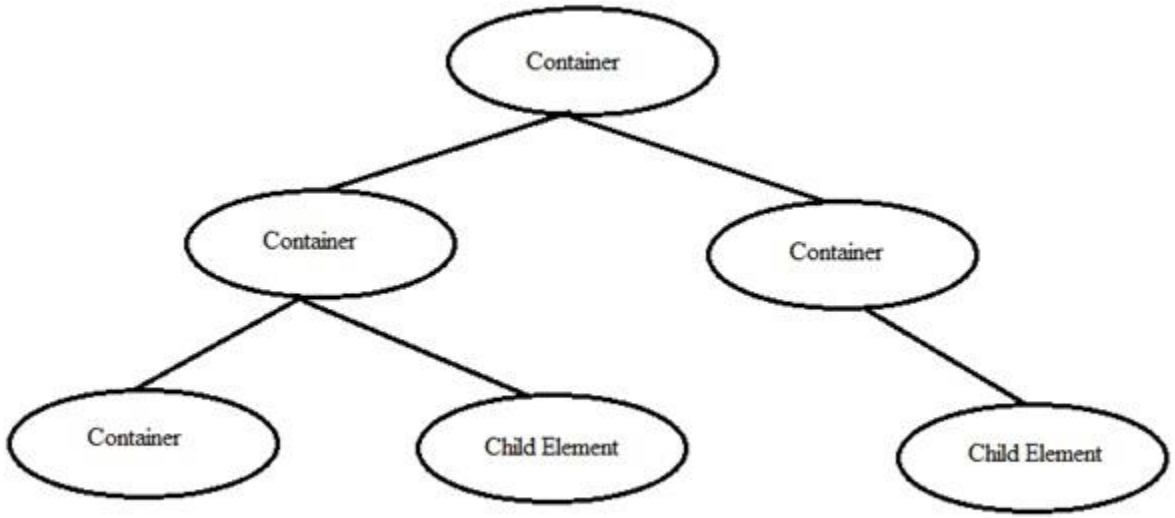
- Using Mixins: Mixins is the different way of using class A in class B without extend.

```
mixins : {
    commons : 'DepartmentApp.utils.DepartmentUtils'
},
```

- Mixins we add in controller where we declare all the other classes such as store, view etc. In this way we can call DepartmentUtils class and use its functions in controller or in this application.

EXT.JS - CONTAINERS

- Container in Ext JS is the component where we can add other container or child components. These containers can have multiple layout to arrange the components in the containers. We can add or remove components from container and from its child elements. Ext.container.Container is the base class for all the containers in Ext JS.



1	**Components inside Container** This example shown how to define components inside container
2	**Container inside container** This example shown how to define container inside container with other components

EXT.JS - LAYOUTS

- Layout is the way the elements are arranged in a container. That could be horizontal, vertical or any other. Ext JS has different layout defined in its library but we can always write custom layouts as well.

1) Absolute

- This layout allows to position the items using XY coordinates in the container.

2) Accordion

- This layout allows to place all the items in stack fashion (one on top of other) inside container.

3) Anchor

- This layout gives the privilege to the user to give the size of each element with respect to the container size.

4) Border

- In this layout various panels are nested and separated by borders.

5) Auto

- This is the default layout decides the layout of the elements based on the number of elements.

EXT.JS - LAYOUTS

6) Card(TabPanel)

- This layout arranges different components in tab fashion. Tabs will be displayed on top of the container.Every time only one tab is visible and each tab is considered as different component.

7) Card(Wizard)

- In this layout every time the elements comes for full container space. There would be a bottom tool bar in the wizard for navigation.

8) Column

- This layout is to show multiple columns in the container. We can define fixed or percentage width to the columns. The percentage width will be calculated based on the full size of the container.

9) Fit

- In this layout the container is filled with a single panel and when there is no specific requirement related to the layout then this layout is used.

10) Table

- As name implies this layout arranges the components in container in the HTML table format.

11) vBox

- This layout allows the element to be distributed in the vertical manner. This is one of the mostly used layout.

12) hBox

- This layout allows the element to be distributed in the horizontal manner.

EXT.JS - COMPONENTS

1) Grid

- Grid component can be used to show the data in the tabular format.

2) Form

- Form widget is to get the data from the user.

3) Message Box

- Message box is basically used to show data in the form of alert box.

4) Chart

- Charts are used to reprent data in pictorial format.

5) Tool tip

- Tool tip is used to show some basic information when any event occurs.

6) Window

- This UI widget is to create a window which should pop up when any event occurs.

7) HTML editor

- HTML Editor is one of the very useful UI component which is used for styling the data which user enters in terms of fonts, color, size etc.

8) Progress bar

- To show the progress of the backend work

DRAG & DROP

Drag and drop feature is one of the powerful feature added for making developers task easy.A drag operation, essentially, is a click gesture on some UI element while the mouse button is held down and the mouse is moved. A drop operation occurs when the mouse button is released after a drag operation.

Adding drag and drop class to the draggable targets.

```
var dd = Ext.create('Ext.dd.DD', el, 'imagesDDGroup', {
    isTarget: false
});
```

Adding drag and drop target class to drappable target

```
var mainTarget = Ext.create('Ext.dd.DDTarget', 'mainRoom', 'imagesDDGroup', {
    ignoreSelf: false
});
```

EXT.JS - THEMES

- Ext.js provides a number of themes to be used in your applications. You can add different theme inplace of classic theme and see the difference in output, this is done simply by replacing theme CSS file.

EXT.JS - CUSTOM EVENTS AND LISTENERS

Events are something which get fired when something happens to the class. For example when a button is getting clicked or before/ after element is rendered.

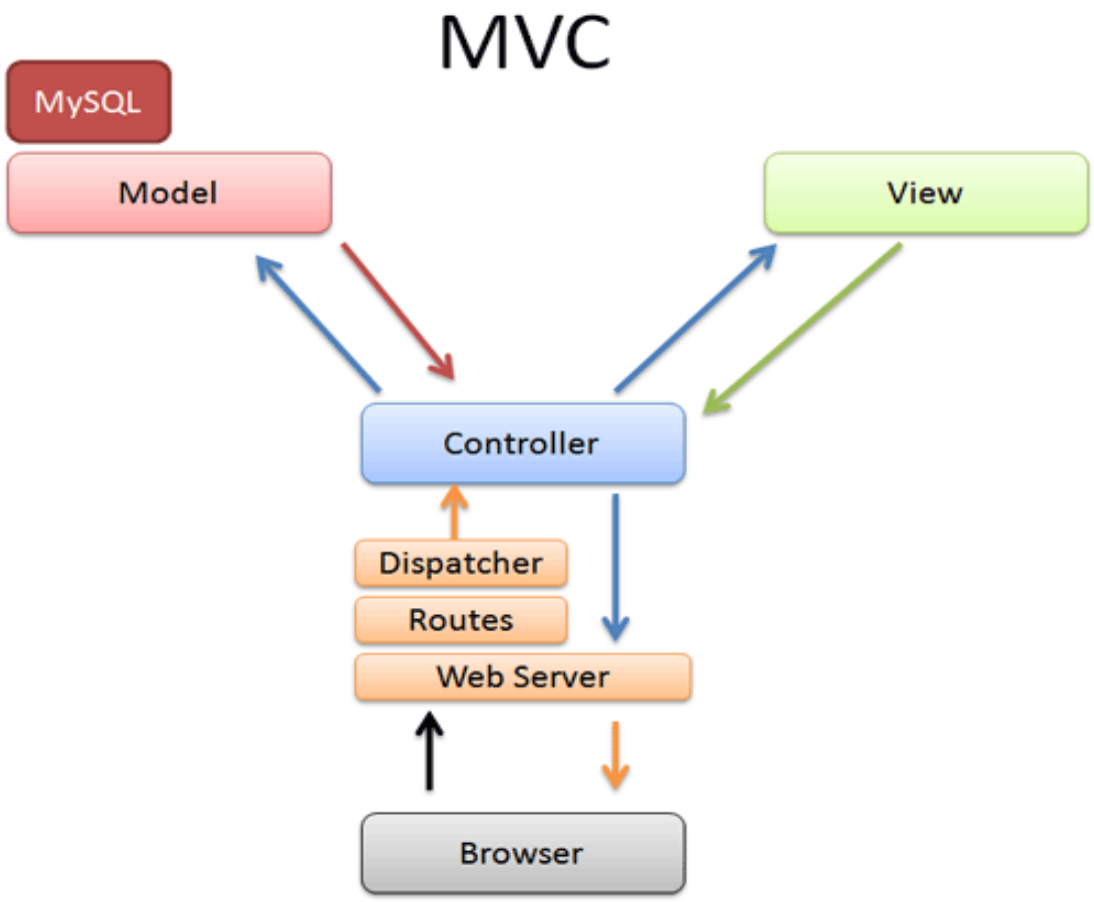
Methods of writing events:

- Built in events using listeners
- Attaching events later
- Custom events

Built in events using listeners

- Ext JS provides listener property for writing events and custom events in Ext JS files.
- Writing listener in Ext JS
- We will add the listener in the previous program itself by adding listen property to the panel a

SERVER SIDE MVC



Server Side Models

- Server Side Models are simple classes that house an 'instantiated' version of a resource record
  - Resource Records can be a row from a MySql Table, required parameters from another server public api, web service, etc
- These models should be transparent to the controller on how the raw data is represented, but rather be in a specified format for the controller

Server Side Models

- To facilitate how the model instantiates data, usually a map is present
- The Map is capable of understanding how to speak with the resource
  - "Select `id`, `first`, `last` from `names`.....
- The model would then have member variables:
  - $model->id
  - $model->first
  - $model->last
  - ....

Server Side Models

- All of my models have key features
  - 1-to-1 resource mapping
  - $model->save()
  - $model->find()
  - $model->delete()
- Similar to CRUD operations except I leave save() to determine wether it is Create or Update
  - CRUD === 'Create Read Update Destroy'

Server Side Views

- Sever Side View classes, for most frameworks, take the model data and return the requested type of view
  - echo($view->buildTable(records));
- This buildTable() function is called by a controller, who then echo()'s the html generated by the view
- Has one major fault
  - What happens when I want to use this server side stack for mobile apps?
- Are there any performance flaws?

Server Side Control

- We have seen that how models and views work
  - These require some sort of delegation
- Controllers will receive the request from the client (old view), do any preprocessing, call the model (CRUD), use the model data, call the view, and return the html
- Within this return, we usually find JavaScript embedded as a code agent to 'enchance' our viewing pleasure.
- What if we mixed this up a bit and used JavaScript as our primary source of control?

Client Side JS with ExtJS

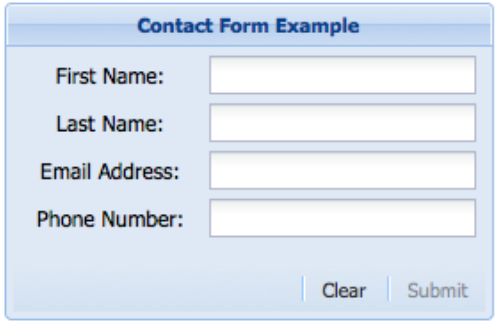
- MVC for JavaScript
- Exactly same process for server side stack, except we now try to use the server as little as possible
  - This allows for powerful client machines to do most of our processing and rendering
  - Only allow the client to manipulate data that can be considered hostile!

ExtJS Models

- The most important feature of ExtJS
  - Can have relational models!!!!!!!
  - Example:
    - Orders (a model) can have many Order Items (another model)
  - Each record of orders is stored in store
  - Each record of orders points to another store that has its Order items
  - This allows us to select an order, and then immediately have access to all its order items

ExtJS View

- Since this is JavaScript, we immediately expect robust GUI widgets



- Of course, you can add CSS and style them

ExtJS Control

- JavaScript is a functional language
  - This allows for very strong and very easy control logic

```
//create record and submit to store
function submitButtonHandler(){

    //first get the form values
    var vals = form.getForm().getValues();

    //now create a record
    var rec = Ext.ModelMgr.create(vals, 'Contacts');

    //insert it at the beginning
    contactStore.insert(0, rec);

    //clear the form
    form.getForm().reset();
}
```

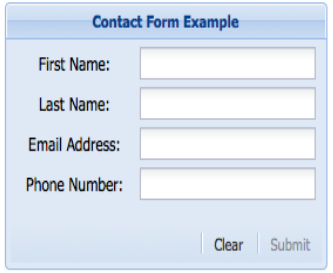
  - Of course, you can still use OOP style if desired

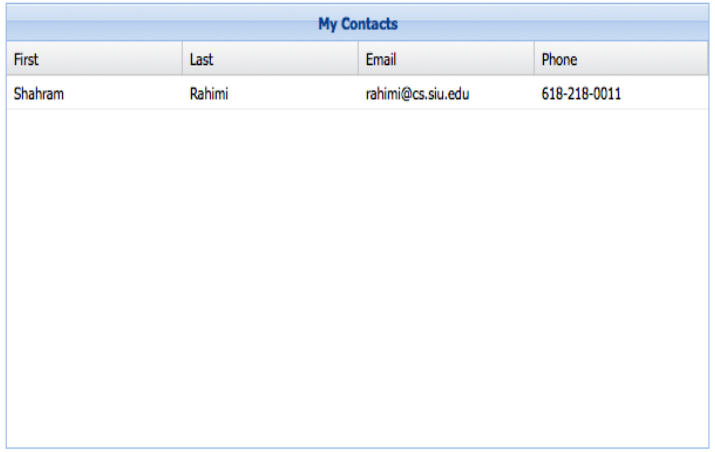
So how does this all work?

- By using MVC on the client side:
  - We only need to contact the server when using CRUD operations
  - By control logic when otherwise needed
- Lets go through an example

Simple Form With Grid

- Our goal will be to make a form, that upon submit, updates a local store
- This store feeds a grid and will automatically update when a new record is inserted





3) Spring Hibernate

OUTLINE

- Object Relational Mapping
- Hibernate
- DAO and Generic DAO Patterns

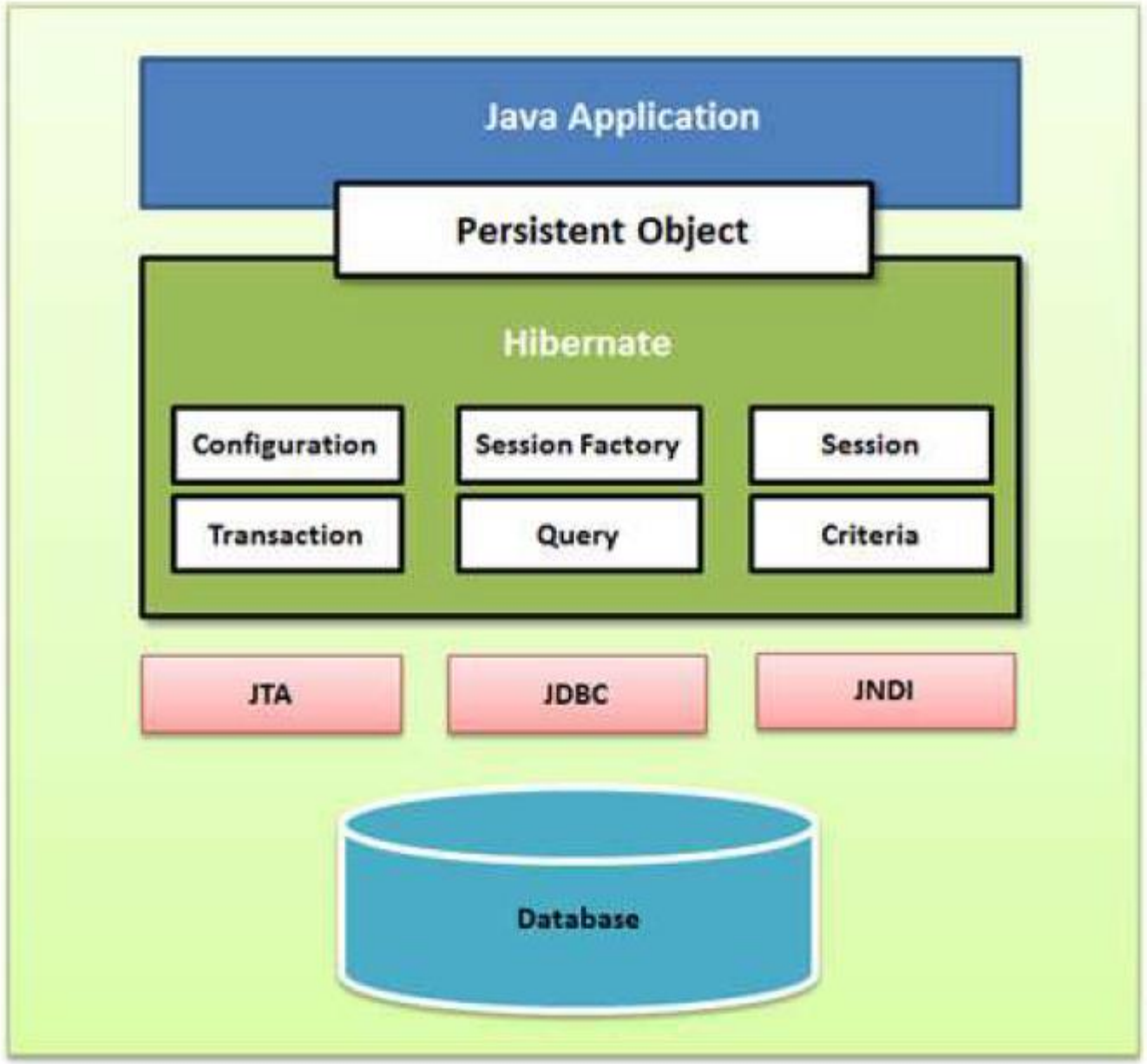
HIBERNATE

- http://www.hibernate.org/
  - Hosted by the JBoss Community
- Open Source
- created by Gavin King in 2001
- Hibernate is a high-performance Object/Relational persistence and query service
- Mapping Java objects to relational databases
- Today
  - Collection of projects/frameworks for extended use of POJO (plain old Java objects)

ROLE OF HIBERNATE

- Hibernate sits between traditional Java objects and database server
- To handle all the work in persisting those objects
- Based on the appropriate O/R mechanisms and patterns

HIBERNATE POSITION



HIBERNATE TERMINOLOGY

- Configuration Object
  - Database Connection
  - Class Mapping Setup
- SessionFactory Object
  - Factory for sessions
  - Thread-safe
  - heavyweight object
  - created during application start up and kept for later use
  - At least one SessionFactory object per database

Hibernate Terminology (2)

- Session Object
  - For physical connection with a database
  - lightweight
  - instantiated each time an interaction is needed with DB
  - Not thread-safe
- Transaction Object
- Query Object
  - String-based
  - HQL: some OO facilities is available
  - SQL: usually is not recommended. Why?
- Criteria Object
  - Fully object oriented queries

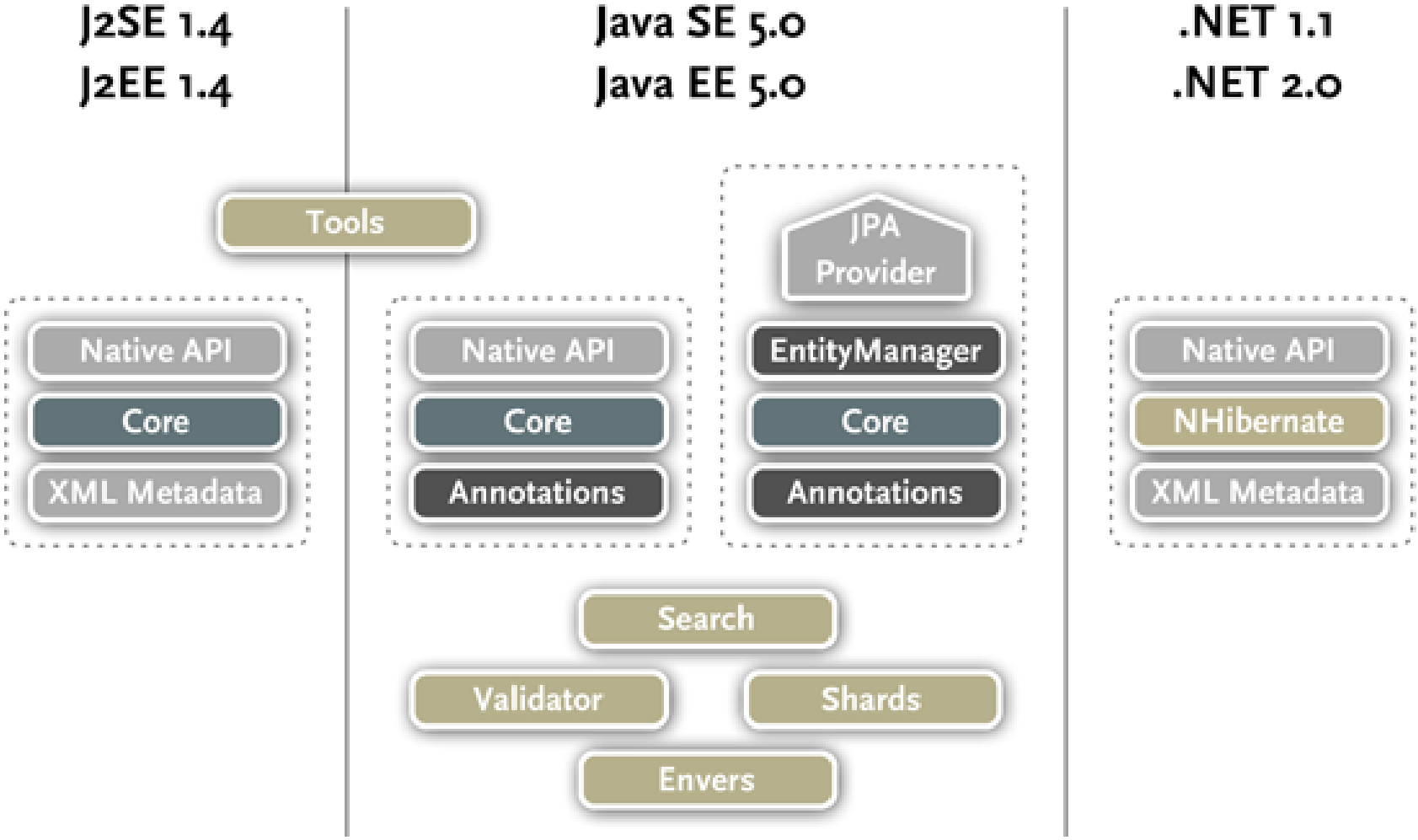
Benefits of Hibernate

- Simpler data persistence
- Automatically handles mapping SQL to Object and vice versa
- Automatic creation of database schemas
- Automatic updating of database schemas
  - Add a field to an object; Hibernate converts your existing database for you.
- Provides search functionality

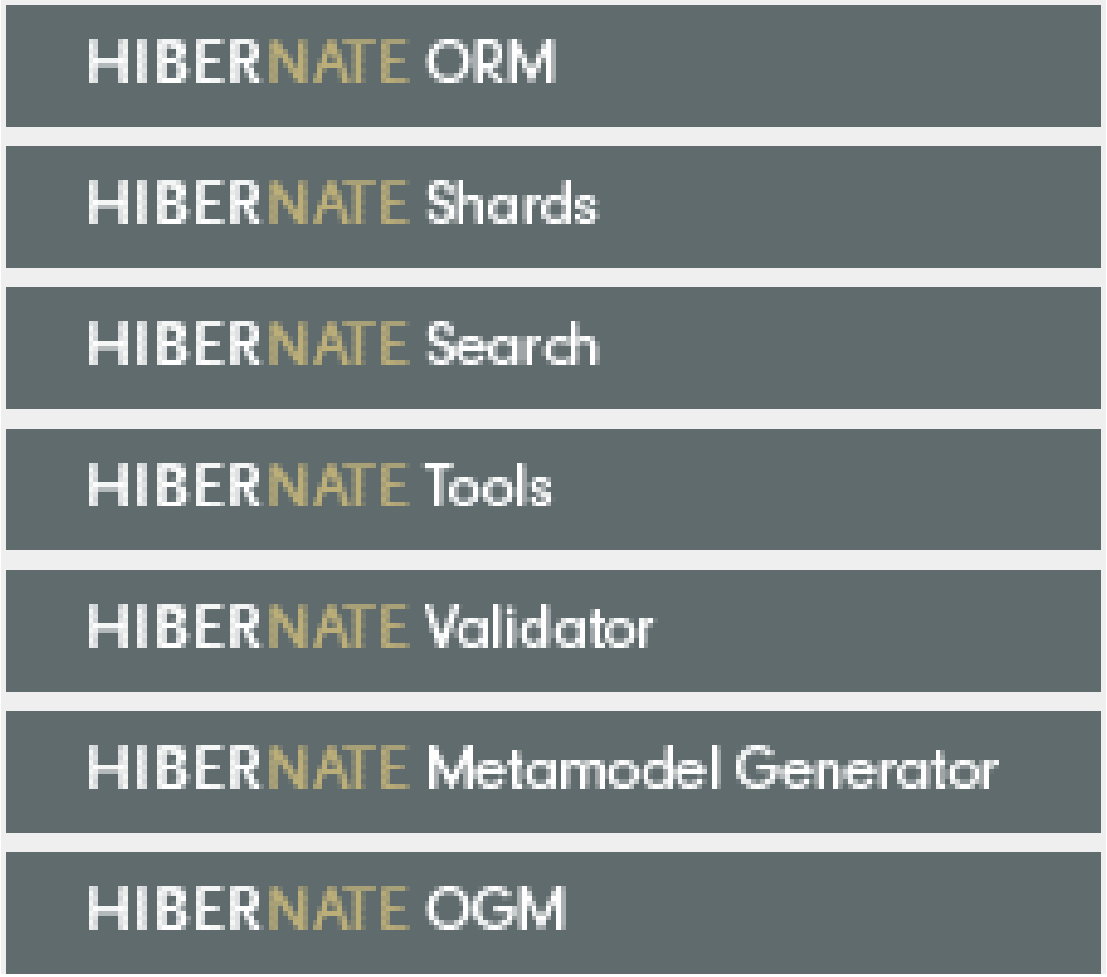
Benefits of Hibernate (2)

- Simpler database management
- No JDBC code or SQL code needed
  - Yet you can still use SQL if you want
- Easy to swap out database engines by a simple configuration change
  - DBMS independence
  - No need to create the schema on the new database
- It’s free
  - LGPL (use in open or closed source project)
  - Open source and standards = no vendor lock-in

HIBERNATE STACK



HIBERNATE PROJECTS



HIBERNATE PROJECTS (2)

- Core
  - Provides the core functionality of object relational mapping and persistence
- Shards
  - Provides for horizontal partitioning of Core so you can put object data in multiple databases
  - Sharding: Rows of a database table are held separately
- Search
  - Combines Apache Lucene full-text search engine with Core.
  - Extends the basic query capabilities of Core.
- Tools
  - Eclipse plugins to facilitate
    - Creating Hibernate mapping files
    - Database connection configurations
    - Reverse engineering existing databases into Java class code

HIBERNATE PROJECTS (3)

- Validator
  - JSR 303 - Bean Validation
  - Standardized way of annotating JavaBean fields with value constraints
  - @NotNull on a bean field also gets set in the database schema
- Metamodel Generator
  - Auto-creation of Criteria classes for use in JPA Criteria API
  - Non-string (strongly-typed) based API for object-based queries
- OGM (Object/Grid Mapper)
  - Allows use of NoSQL data stores versus SQL relational
  - providing Java Persistence (JPA) support for NoSQL solutions

JDBC

- Java DataBase Connectivity
- API for accessing relational databases from a Java program
- SQL-based
- Flexible architecture to write DBMS-independent applications
  - Fully independent? no.

JDBC Pros and Cons

- Pros
  - Clean and simple SQL processing
  - Good performance with large data
  - Very good for small applications
  - Simple syntax so easy to learn
- Cons
  - Complex if it is used in large projects
  - Large programming overhead
  - No encapsulation
  - Query is DBMS specific

JDBC EXAMPLE

```
Class.forName("com.mysql.jdbc.Driver");
String SQL = "SELECT * FROM classes";
try (
    Connection conn = DriverManager.getConnection(
        "jdbc:mysql://localhost/cse3330a",
        "root", "aSecret123");
    Statement stmt = conn.createStatement();
    ResultSet rs = stmt.executeQuery(SQL)
    ) {

  while (rs.next()) {
    System.out.println(rs.getString("prof"));
  }
}
```

JDBC + SPRING

- DataSource object
- JdbcTemplate object
- RowMapper
  - a simple ORM for select queries

```
String SQL = "select age from Student where id = ?";
int age = jdbcTemplate.queryForInt(SQL, new Object[]{10});


String SQL = "select * from Student where id = ?";
Student student = jdbcTemplate.queryForObject(
SQL, new Object[]{10}, new StudentMapper());
```

Object Relational Mapping

- RDBMSs represent data in a tabular format
  - Relational
- Object-oriented languages, such as Java or C# represent data as an interconnected graph of objects
- Context gap

Hibernate Class Entities

- Class attributes
  - Hibernate uses reflection to populate
  - Can be private or whatever
- Class requirements
  - Default constructor
- JavaBean pattern common
- Do not make the entity classes **final**
- 3 methods of serialization definition
  - Following slides

Hibernate Annotation Mappings

- Annotations in code
  - Beginning of class
  - Indicate class is Entity
    - Class doesn’t have to implement java.lang.Serializable
  - Define database table
  - Define which attributes to map to columns
    - Supports auto-increment IDs too
    - Can dictate value restrictions (not null, etc)
    - Can dictate value storage type
- Existed before JPA standard (later slides)
- Doesn’t require a separate hbm.xml mapping file (discussed later)
  - But is tied to code

HIBERNATE ANNOTATION EXAMPLE

```
@Entity
@Table( name = "EVENTS" )
public class Event {
   private Long id;
…
    @Id
    @GeneratedValue(generator="increment”
    @GenericGenerator(name="increment", strategy = "increment")
    public Long getId() {  return id;    }

    @Temporal(TemporalType.TIMESTAMP)
    @Column(name = "EVENT_DATE")
    public Date getDate() { return date;    }
    public void setDate(Date date) { this.date = date;    }
…
}
```

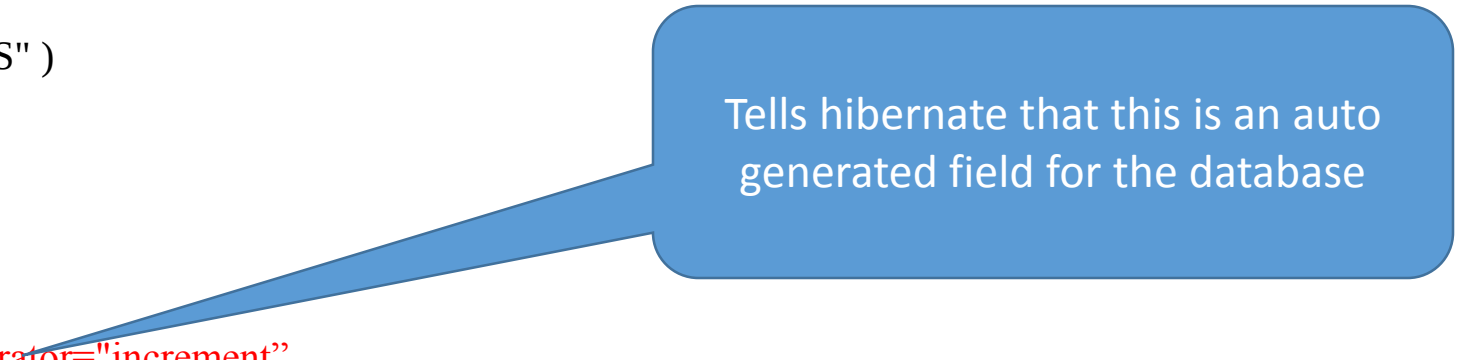
HIBERNATE ANNOTATION EXAMPLE (2)

```
@Entity
@Table( name = "EVENTS" )
public class Event {
   private Long id;
…
    @Id
    @GeneratedValue(generator="increment”
    @GenericGenerator(name="increment", strategy = "increment")
    public Long getId() {  return id;    }

    @Temporal(TemporalType.TIMESTAMP)
    @Column(name = "EVENT_DATE")
    public Date getDate() { return date;    }
    public void setDate(Date date) { this.date = date;    }
…
}
```

Tells hibernate this goes into the EVENTS table

HIBERNATE ANNOTATION EXAMPLE (3)

```
@Entity
@Table( name = "EVENTS" )
public class Event {
   private Long id;
…
    @Id
    @GeneratedValue(generator="increment”
    @GenericGenerator(name="increment", strategy = "increment")
    public Long getId() {  return id;    }

    @Temporal(TemporalType.TIMESTAMP)
    @Column(name = "EVENT_DATE")
    public Date getDate() { return date;    }
    public void setDate(Date date) { this.date = date;    }
…
}
```



HIBERNATE ANNOTATION EXAMPLE (4)

```
@Entity
@Table( name = "EVENTS" )
public class Event {
   private Long id;
…
    @Id
    @GeneratedValue(generator="increment”
    @GenericGenerator(name="increment", strategy = "increment")
    public Long getId() {  return id;    }

    @Temporal(TemporalType.TIMESTAMP)
    @Column(name = "EVENT_DATE")
    public Date getDate() { return date;    }

    public void setDate(Date date) { this.date = date;    }
…
}
```

Note that you don’t need any annotations on the actual private fields or setters if you use the standard JavaBean pattern. The getter defines it.

Hibernate Annotation Example (5)

```
@Entity
@Table( name = "EVENTS" )
public class Event {
…
   private String title;

   public String getTitle() {  return title;    }

   public void setTitle(String title) {         this.title = title;    }
…
}
```

Also note that this is automatically stored with a column name of “title” so we didn’t have to add any annotations

JPA ANNOTATION

- Became standard
  - Came after Hibernate annotations
- Works almost like Hibernate annotations
  - Requires “META-INF/persistence.xml” file
    - Defines data source configuration
    - HibernatePersistence provider
      - Auto-detects any annotated classes
      - Auto-detects any hbm.xml class mapping files
        - (later slides)
      - Allows explicit class loading for mapping
- Annotation syntax
  - Same as Hibernate
  - Hibernate has a few extensions (see docs)

JPA 2 XML

- JPA 2 XML
  - like Hibernate's hbm.xml discussed later
- Separate from code unlike in-line annotations

```xml
<?xml version="1.0" encoding="UTF-8" ?>
<entity-mappings xmlns="http://java.sun.com/xml/ns/persistence/orm"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     xsi:schemaLocation="http://java.sun.com/xml/ns/persistence/orm ;
http://java.sun.com/xml/ns/persistence/orm_1_0.xsd"
    version="1.0">
    <description>My First JPA XML Application</description>
    <package>entity</package>
    <entity class="entity.Employee" name="Employee">
        <table name="EMPLOYEETABLE"/>
        <attributes>
            <id name="empId">
                <generated-value strategy="TABLE"/>
            </id>
            <basic name="empName">
                <column name="EMP_NAME" length="100"/>
            </basic>
            <basic name="empSalary">
            </basic>
        </attributes>
   </entity>
</entity-mappings>
```

Hibernate *.hbm.xml Mappings

- For each class
  - Define ClassName.hbm.xml
    - ClassName not required convention
  - Usually stored in same package
    - I like the convention of
      - src/main/java (actual .java source)
      - src/main/resources (any configuration files, et cetera)
      - src/main/hibernate (I put all of my .hbm.xml here)
        - Matching folder/package structure to src/main/java
  - Optionally multiple classes in one .hbm.xml possible
    - Not common convention though
- Becoming legacy in favor of JPA 2 XML or Annotations

Hibernate *.hbm.xml mappings (2)

```
<?xml version="1.0"?>


<!DOCTYPE hibernate-mapping PUBLIC
        "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
        "http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">

<hibernate-mapping package="org.hibernate.tutorial.hbm">

    <class name="Event" table="EVENTS">
        <id name="id" column="EVENT_ID">
            <generator class="increment"/>
        </id>
        <property name="date" type="timestamp" column="EVENT_DATE"/>
        <property name="title"/>
    </class>

</hibernate-mapping>
```

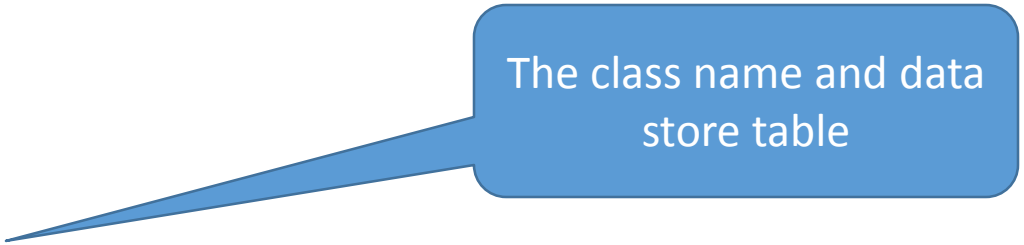
Hibernate *.hbm.xml mappings (3)

```
<?xml version="1.0"?>


<!DOCTYPE hibernate-mapping PUBLIC
        "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
        "http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">

<hibernate-mapping package="org.hibernate.tutorial.hbm">

    <class name="Event" table="EVENTS">
        <id name="id" column="EVENT_ID">
            <generator class="increment"/>
        </id>
        <property name="date" type="timestamp" column="EVENT_DATE"/>
        <property name="title"/>
    </class>

</hibernate-mapping>
```

Define what package for the class(es)

Hibernate *.hbm.xml mappings (4)

```
<?xml version="1.0"?>


<!DOCTYPE hibernate-mapping PUBLIC
        "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
        "http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">

<hibernate-mapping package="org.hibernate.tutorial.hbm">

    <class name="Event" table="EVENTS">
        <id name="id" column="EVENT_ID">
            <generator class="increment"/>
        </id>
        <property name="date" type="timestamp" column="EVENT_DATE"/>
        <property name="title"/>
    </class>

</hibernate-mapping>
```



HIBERNATE *.HBM.XML MAPPINGS (5)

```
<?xml version="1.0"?>


<!DOCTYPE hibernate-mapping PUBLIC
        "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
        "http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">

<hibernate-mapping package="org.hibernate.tutorial.hbm">

    <class name="Event" table="EVENTS">
        <id name="id" column="EVENT_ID">
            <generator class="increment"/>
        </id>
        <property name="date" type="timestamp" column="EVENT_DATE"/>
        <property name="title"/>
    </class>

</hibernate-mapping>
```

The properties of the class to save (getAttribute() or class.attribute style)

HIBERNATE PRIMITIVE TYPES

Mapping type	Java type	ANSI SQL Type
integer	int or java.lang.Integer	INTEGER
long	long or java.lang.Long	BIGINT
short	short or java.lang.Short	SMALLINT
float	float or java.lang.Float	FLOAT
double	double or java.lang.Double	DOUBLE
big_decimal	java.math.BigDecimal	NUMERIC
character	java.lang.String	CHAR(1)
string	java.lang.String	VARCHAR
byte	byte or java.lang.Byte	TINYINT
boolean	boolean or java.lang.Boolean	BIT
yes/no	boolean or java.lang.Boolean	CHAR(1) ('Y' or 'N')
true/false	boolean or java.lang.Boolean	CHAR(1) ('T' or 'F')

DATE TYPES

date	java.util.Date or java.sql.Date	DATE
time	java.util.Date or java.sql.Time	TIME
timestamp	java.util.Date or java.sql.Timestamp	TIMESTAMP
calendar	java.util.Calendar	TIMESTAMP
calendar_date	java.util.Calendar	DATE

BINARY AND LARGE OBJECT TYPES

Mapping type	Java type	ANSI SQL Type
binary	byte[]	VARBINARY (or BLOB)
text	java.lang.String	CLOB
serializable	any Java class that implements java.io.Serializable	VARBINARY (or BLOB)
clob	java.sql.Clob	CLOB
blob	java.sql.Blob	BLOB

Hibernate Configuration

- Need to define
  - Data source (database) connection
    - Engine, URI, credentials, etc
    - Pooling mechanism (Hibernate provides C3P0)
  - Mapping XML definitions or classes with annotations
    - You specify which ones to actually activate
    - Also allows for multiple databases and mappings
    - **Many IDE and Ant tools exist to auto-create .hbm.xml and hibernate.cfg.xml/persistence.xml**
- JPA
  - Provides some auto-discovery at startup
    - Limited to jar files
- Hibernate
  - Has XML or .properties (not used much)
- Approaches
  - XML configuration file
  - Programmatic

HIBERNATE.CFG.XML

- Usually placed in root of the classpath (/hibernate.cfg.xml) for auto-detection.
- You can also programmatically load it in code.

```
<hibernate-configuration>
    <session-factory>
…
    </session-factory>
</hibernate-configuration>
```

<SESSION-FACTORY>

```
<property name="connection.driver_class">org.h2.Driver</property>
<property name="connection.url">jdbc:h2:mem:db1 </property>
<property name="connection.username">sa</property>
<property name="connection.password">123</property>

<property name="connection.pool_size">1</property>

<property name="dialect">org.hibernate.dialect.H2Dialect</property>

<property name="cache.provider_class">org.hibernate.cache.NoCacheProvider</property>

<property name="show_sql">true</property>

<property name="hbm2ddl.auto">create</property>

<mapping class="org.hibernate.tutorial.annotations.Event"/>
```

HIBERNATE.CFG.XML (2)

```
<!-- Database connection settings -->
<property name="connection.driver_class">org.h2.Driver</property>
<property name="connection.url">jdbc:h2:mem:db1</property>
<property name="connection.username">sa</property>
<property name="connection.password"></property>
```

- You define the database settings in this manner
- There are third party library extensions that allow the password inside hibernate.cfg.xml to be encrypted as well
  - Or you could use programmatic configuration instead
- You can’t use system/environment variables inside the config

HIBERNATE.CFG.XML (3)

```
<!-- SQL dialect -->
    <property name="dialect">org.hibernate.dialect.H2Dialect</property>
```

- Depending on the database/data source you use you need to let Hibernate know how to talk to it. Hibernate supports many data sources.

```
    <!-- Drop and re-create the database schema on startup -->
    <property name="hbm2ddl.auto">create</property>
```

- You can have Hibernate:
  - “validate” - Exists and has expected tables/columns; don’t touch data/schema
  - “update” – Create if needed and update tables/columns if class has new fields
  - “create” – Drop any existing and create new (only at start of session)
  - “create-drop” – Same as create but also drop at end of session

Caveats of hbm2ddl.auto

- Some people don't recommend
  - "update" can mess up
    - Doesn't remove renamed columns/attributes
      - Makes new one by default unless you update mapping
    - Migrations not perfect
      - Suppose add not null constraint after the fact
        - Existing nulls would blow up
    - Not recommended for production use anyway
- Hibernate class tools like
  - SchemaExport and SchemaUpdate
  - Useful for getting auto-generated SQL migration/creation scripts
- When set to "create" or "create-drop"
  - hbm2ddl.import_file – allow specifying */path/somefile.sql* to run
  - (used to be hard coded "/import.sql")
  - Needs to be in classpath

HIBERNATE.CFG.XML (4)

<!-- Names the annotated entity class -->

<mapping **class**="org.hibernate.tutorial.annotations.Event"/>

- For each class with annotations you provide a <mapping/> entry

- For each .hbm.xml it looks like this instead
  - <mapping **resource**="org/hibernate/tutorial/domain/Event.hbm.xml"/>
  - Event.hbm.xml is in the classpath
  - No *.hbm.xml is possible, sorry. See persistence.xml for wildcards or programmatic configuration.

PERSISTENCE.XML

- JPA 2 XML method
  - Similar idea to hibernate.cfg.xml
- Additions
  - Allows defining EJB Persistence provider
    - Usually HibernatePersistence suffices
  - jar-file option
    - Allows auto-inclusion of any classes with annotations
      - No need for manual mapping like hibernate.cfg.xml
    - Also auto-includes any .hbm.xml files

Programmatic Configuration

```
Configuration cfg = new Configuration();
cfg.addResource("Item.hbm.xml");
cfg.addResource("Bid.hbm.xml");
```

- Assumes the .hbm.xml are in the classpath
  - This example assumes the root

Programmatic configuration *(2)*

```
Configuration cfg = new Configuration();
cfg.addClass(org.hibernate.auction.Item.class);
cfg.addClass(org.hibernate.auction.Bid.class);
```

- Translates to “/org/hibernate/auction/Item.hbm.xml”
  - Again in classpath
  - Avoids hardcoded filenames (less work than updating hibernate.cfg.xml)
- If class is annotated uses annotation versus xml

Programmatic configuration (3)

```
Configuration cfg = new Configuration()
    .addClass(org.hibernate.auction.Item.class)
    .addClass(org.hibernate.auction.Bid.class)
    .setProperty("hibernate.dialect",
  "org.hibernate.dialect.MySQLInnoDBDialect")
    .setProperty("hibernate.connection.datasource",
  "java:comp/env/jdbc/test")
    .setProperty("hibernate.order_updates", "true");
```

- Database settings can be configured as well
  - Note the “hibernate.” prefix on the options this time

Programmatic Configuration (4)

```
Configuration conf = new Configuration();
conf.configure().setProperty("hibernate.connection.url",
   "jdbc:mysql://"+host+"/"+dbname);

conf.setProperty("hibernate.connection.username", user);

conf.setProperty("hibernate.connection.password", password);
```

- Load /hibernate.cfg.xml as defaults
  - Mappings still defined inside XML
- Override database settings with run-time values

Mapping & Configuration Summary

- Each Java class
  - Annotations or XML mapping file (.hbm.xml)
    - or JPA persistence orm.xml
    - You can mix annotations and XML mappings
- Hibernate XML Configuration
  - Each class/.hbm.xml gets <mapping/> entry
  - Configure database
  - hibernate.cfg.xml
    - or for JPA use persistence.xml
  - Alternative
    - Programmatic configuration
    - Hybrid approach

INITIALIZING HIBERNATE

```
try {
        // Create the SessionFactory from hibernate.cfg.xml
        return new Configuration().configure().buildSessionFactory();
} catch (Throwable ex) {
        // Make sure you log the exception, as it might be swallowed
        System.err.println("Initial SessionFactory creation failed." + ex);
        throw new ExceptionInInitializerError(ex);
}
```

- SessionFactory returned is used later

Saving data

```
Session session = HibernateUtil.getSessionFactory().getCurrentSession();
session.beginTransaction();

Event theEvent = new Event();
theEvent.setTitle(title);
theEvent.setDate(theDate);

session.save(theEvent);

session.getTransaction().commit();
```

- Just start a transaction similar to how you do in databases

LOADING DATA

```
Session session = HibernateUtil.getSessionFactory().getCurrentSession();
session.beginTransaction();  // important even for query
List result = session.createQuery("from Event").list();
session.getTransaction().commit();
return result;
```

- Note the “from Event” which is SQL like
  - Known as **HQL**
  - More complex queries possible
  - See http://docs.jboss.org/hibernate/core/3.6/reference/en-US/html/queryhql.html

HIBERNATE SESSION

- A Session is used to get a physical connection with DB
- The Session object is lightweight
- designed to be instantiated each time an interaction is needed with DB
- Persistent objects are saved and retrieved through a Session object
- Session vs. DB connection
- Every object that is loaded by Hibernate from the database is associated with the session
  - Allowing Hibernate to automatically persist objects that are modified
  - Allowing Hibernate to implement functionality such as **lazy-loading**

Sessions and Threads

- Hibernate sessions are not thread-safe
- you shouldn't pass a Hibernate session into a new thread
- you must not share Hibernate-managed objects between threads
- Spring's transaction management places the Hibernate session in a ThreadLocal variable
  - accessed via the sessionFactory
  - DAOs use that ThreadLocal
  - When you create a new thread you no longer have access to the Hibernate session for that thread

OpenSessionInView Filter

- The SessionInViewFilter
- Opens a Hibernate session which is then available for the entire web request

- The pattern has pros and cons

Querying Data

- Remember don’t use string concatenation to form queries
  - Bad: “from Users where id=” + paramUserId
  - Why? **SQL Injection** Vulnerabilities
    - paramUserId = “1 OR 1=1”
- Use Queries instead
  - List cats = sess.createCriteria(Cat.class)
    .add(Restrictions.like("name", "Fritz%"))
    .add(Restrictions.between("weight", minWeight, max))
    .list();
    - http://docs.jboss.org/hibernate/core/3.6/reference/en-US/html_single/#querycriteria
  - List cats = session.createQuery(
    “from Cat as cat where cat.birthdate < **?**“)
    .setDate(0, date)
    .list();
    - http://docs.jboss.org/hibernate/core/3.6/reference/en-US/html_single/#objectstate-querying

CLOSING CONNECTION

sessionFactory.close();

- Closes resources
  - Connection pools
  - Caches
  - Etc.

Session and Thread Safety

- org.hibernate.Session
  - Don’t use the deprecated class org.hibernate.classic.Session
- **NOT** thread safe
  - Each thread should call SessionFactory.getCurrentSession()
    - Versus .openSession() which has
      - No automatic close management
      - No automatic association to a thread

HQL

- Hibernate Query Language
- Currently a superset of JPQL
- Select, insert, delete, update queries
- SQL-Similar syntax
- With most of its features
  - Join
  - Order
  - Group
  - Aggregations
  - …

HQL (Examples)

- from Cat
- select cat.mate from Cat cat
- select cat.name from DomesticCat cat where cat.name like 'fri%‘
- select avg(cat.weight), sum(cat.weight) from Cat cat

- HQL is OO
- The names are properties (not columns)

FIND BY CRITERIA

```
Criteria criteria =
session.createCriteria(Person.class);
SimpleExpression like =
   Restrictions.like("firstName", "%i%");
Criteria exampleCriteria = criteria.add(like);
List<Person> list = exampleCriteria.list();
for (Person person2 : list) {
 System.out.println(person2.getFirstName());
}
```

RESTRICTIONS

- Eq
- Gt
- Le
- Like
- And
- Or
- …

FIND BY EXAMPLES

```
Person person = new Person();
person.setNationalCode(123);
person.setFirstName("Taghi");
person.setLastName("Taghavi");
Criteria criteria =
    session.createCriteria(Person.class);
Criteria exampleCriteria =
criteria.add(Example.create(person));
List<Person> list = exampleCriteria.list();
for (Person person2 : list) {
     System.out.println(person2.getFirstName());
}
```

Associations Mapping

- One-to-One
- One-to-Many
- Many-to-One
- Many-to-Many

MANY-TO-ONE

```
<class name="Employee" table="EMPLOYEE">
    <meta attribute="class-description"> This clas
    <id name="id" type="int" column="id">
        <generator class="native" />
    </id>
    <property name="firstName" column="first_name"
    <property name="lastName" column="last_name" t
    <property name="salary" column="salary" type="
    <many-to-one name="address" column="address"
    class="Address" not-null="true" />
</class>
```

MANY-TO-ONE (CONT'D)

```
public class Employee{
   private int id;
   private String firstName;
   private String lastName;
   private int salary;
   private Address address;
```

- + getters and setters

MAY-TO-ONE BY ANNOTATION

```
@Entity
@Table(name="cats")
public class Cat {
@ManyToOne
@JoinColumn(name="mother_id")
public Cat getMother() { return mother; }
…
```

ONE-TO-ONE

- similar to many-to-one association
- the column will be set as unique
- E.g. an address associated with a single employee
- Applications?

```
public class Employee{
   private int id;
   private String firstName;
   private String lastName;
   private int salary;
   private Address address;
```

ONE-TO-MANY

```
<class name="Person">
    <id name="id" column="personId">
        <generator class="native"/>
    </id>
    <set name="addresses">
        <key column="personId"
            not-null="true"/>
        <one-to-many class="Address"/>
    </set>
</class>

<class name="Address">
    <id name="id" column="addressId">
        <generator class="native"/>
    </id>
```

ONE-TO-MANY (CONT'D)

```
public class Product {
   private String serialNumber;
   private Set<Part> parts;

   public String getSerialNumber() {return serialNumber; }
   void setSerialNumber(String sn) { serialNumber = sn; }

   public Set<Part> getParts() { return parts; }
   void setParts(Set parts) { this.parts = parts; }
}
```

ONE-TO-MANY BY ANNOTATIONS

```
@Entity
public class Product {
    private String serialNumber;
    private Set<Part> parts;

    @Id
    public String getSerialNumber() {return serialNumber; }
    void setSerialNumber(String sn) { serialNumber = sn; }

    @OneToMany
    @JoinColumn(name="PART_ID")
    public Set<Part> getParts() { return parts; }
    void setParts(Set parts) { this.parts = parts; }
}
```

MANY-TO-MANY

```
@Entity
public class Store {
  @ManyToMany
  public Set<City> getCities() {          ...     }
}


@Entity
public class City {     ... //no bidirectional relationshi
p }
```

Many-to-Many (cont'd)

```
@Entity
public class Store {
    @ManyToMany(cascade = {CascadeType.PERSIST,
    public Set<Customer> getCustomers() {
        ...
    }
}

@Entity
public class Customer {
    @ManyToMany(mappedBy="customers")
    public Set<Store> getStores() {
        ...
    }
}
```

Collection Mapping Types

- java.util.Set
  - mapped with a <set> element, initialized with HashSet
- java.util.SortedSet
  - mapped with a <set> element, initialized with java.util.TreeSet.
  - The **sort** attribute
- java.util.List
  - mapped with a <list> element, initialized with java.util.ArrayList

Collection Mapping Types (Cont'd)

- java.util.Collection
  - mapped with a <bag> or <ibag> element , initialized with ArrayList
- java.util.Map
  - mapped with a <map> element
  - initialized with java.util.HashMap
- java.util.SortedMap
  - mapped with a <map> element
  - initialized with java.util.TreeMap.
  - The **sort** attribute

CASCADE

- Cascades the operation to the children
- Cascade-types:
  - none
  - Save-update
  - Delete
  - Delete-orphan
  - …

EXAMPLE: CASCADE=SAVE-UPDATE

```
<!-- Stock.hbm.xml -->
<set name="stockDailyRecords" cascade="save-update" table="stock_
        <key>
                <column name="STOCK_ID" not-null="true" />
        </key>
        <one-to-many class="com.mkyong.common.StockDailyRecord" />
</set>
```

```
Stock stock = new Stock();
StockDailyRecord stockDailyRecords = new StockDailyRecord();
//set the stock and stockDailyRecords  data

stockDailyRecords.setStock(stock);
stock.getStockDailyRecords().add(stockDailyRecords);

session.save(stock);
```

Saves some stockDailyRecords too

EXAMPLE: CASCADE=DELETE

```
<!-- Stock.hbm.xml -->
<set name="stockDailyRecords" cascade="delete" table="stock_daily
      <key>
            <column name="STOCK_ID" not-null="true" />
      </key>
      <one-to-many class="com.mkyong.common.StockDailyRecord" />
</set>
```

```
Query q = session.createQuery("from Stock where stockCode = :stockCode ");
q.setParameter("stockCode", "4715");
Stock stock = (Stock)q.list().get(0);
session.delete(stock);
```

EXAMPLE: CASCADE=DELETE-ORPHAN

```
<!-- Stock.hbm.xml -->
<set name="stockDailyRecords" cascade="delete-orphan" table="stoc
        <key>
                <column name="STOCK_ID" not-null="true" />
        </key>
        <one-to-many class="com.mkyong.common.StockDailyRecord" />
</set>
```

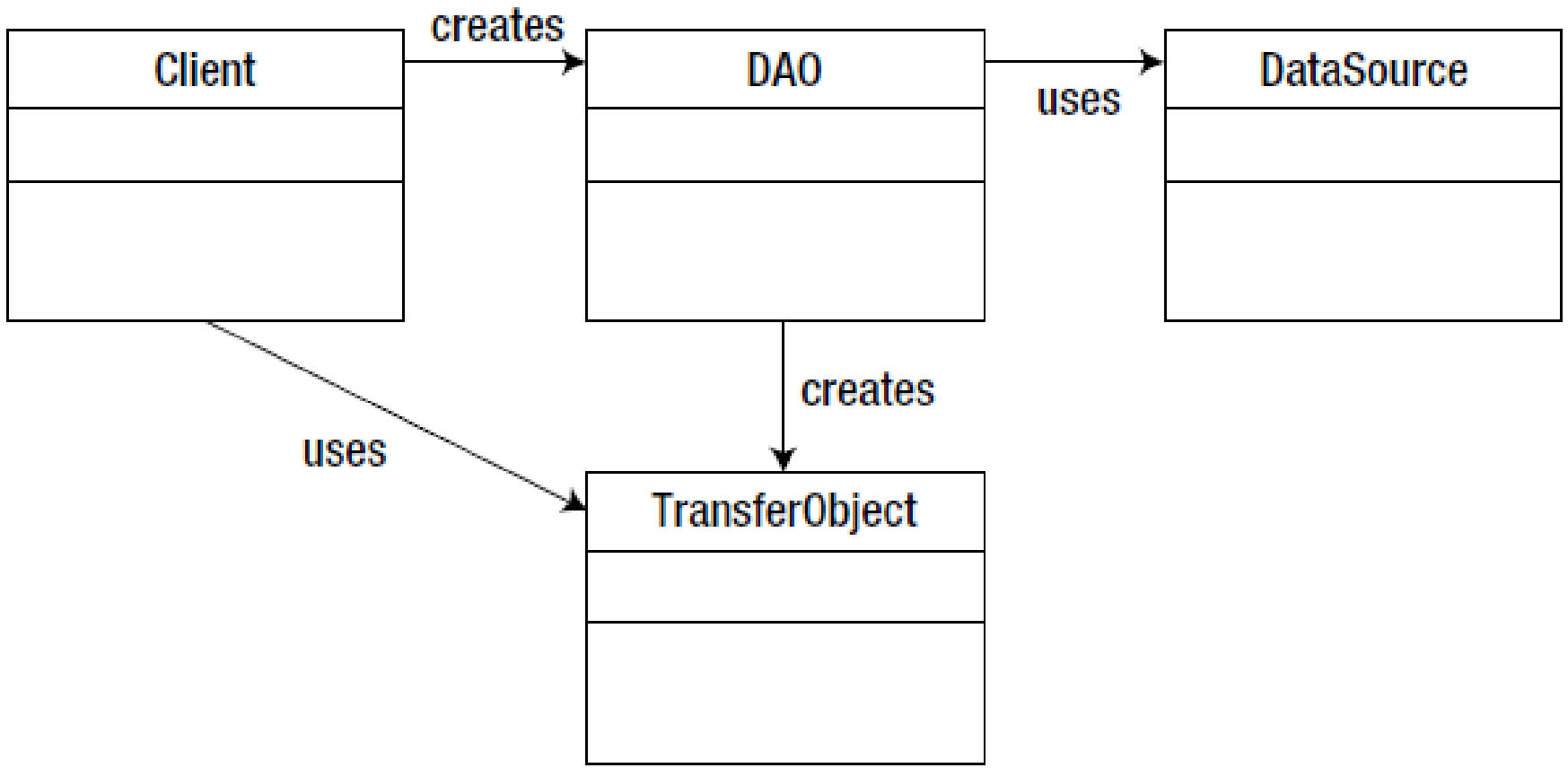
```
StockDailyRecord sdr1 = (StockDailyRecord)session.get(StockDailyRecord.class,
                                        new Integer(56));
StockDailyRecord sdr2 = (StockDailyRecord)session.get(StockDailyRecord.class,
                                        new Integer(57));

Stock stock = (Stock)session.get(Stock.class, new Integer(2));
stock.getStockDailyRecords().remove(sdr1);
stock.getStockDailyRecords().remove(sdr2);

session.saveOrUpdate(stock);
```

Lazy Collection Fetching

- A collection is fetched when the application invokes an operation upon that collection
- This is the default for collections

- lazy=true | false

Data Access Object Pattern (DAO)



DAO

- Abstracts the details of the underlying persistence mechanism
- Hides the implementation details of the data source from its clients
- **Loose coupling** between core business logic and persistence mechanism

DAO: EXAMPLE

```
public interface UserDAO {

   void insertUser(User user);

   User getUserById(int userId);

   User getUser(String username);

   List<User> getUsers();
}
```

HIBERNATE + SPRING

- Springs helps you write hibernate codes
  - better and simpler
- In creating and managing objects (beans)
  - Datasource
  - SessionFactory
  - Sessions
- In Transaction Management

Spring Integration:
1. Write Entity Classes

```
@Entity
@Table(name="USER")
public class User {

  @Id
  @GeneratedValue(strategy = GenerationType.AUTO)
  @Column(name="ID", nullable = false)
  private int id;

  @Column(name="USERNAME", nullable = false)
  private String username;

  @Column(name="NAME", nullable = false)
  private String name;

  public int getId() {
    return id;
  }

  public void setId(int id) {
    this.id = id;
  }
```

Spring Integration:
2. DOA Interfaces

```
public interface UserDAO {

   void insertUser(User user);

   User getUserById(int userId);

   User getUser(String username);

   List<User> getUsers();
}
```

Spring Integration:
3. DAO Implementations

```
public class UserDAOImpl implements UserDAO {

  @Autowired
  private SessionFactory sessionFactory;

  @Override
  public void insertUser(User user) {
    sessionFactory.getCurrentSession().save(user);
  }

  @Override
  public User getUserById(int userId) {
    return (User) sessionFactory.
      getCurrentSession().
      get(User.class, userId);
  }
```

Spring Integration:
4. Service (Manager, Business) Interface

```
public interface UserManager {

  void insertUser(User user);

  User getUserById(int userId);

  User getUser(String username);

  List<User> getUsers();
}
```

Spring Integration:
5. Service Implementation

```
public class UserManagerImpl implements UserManager {

  @Autowired
  private UserDAO userDAO;

  @Override
  @Transactional
  public void insertUser(User user) {
    userDAO.insertUser(user);
  }

  @Override
  @Transactional
  public User getUserById(int userId) {
    return userDAO.getUserById(userId);
  }
```

Spring Integration:
6. Spring Configuration (1)

```
<tx:annotation-driven />
```

SPRING INTEGRATION:
6. SPRING CONFIGURATION (2)

```
<bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource"
  destroy-method="close">
  <property name="driverClassName" value="com.mysql.jdbc.Driver" />
  <property name="url" value="jdbc:mysql://localhost:3306/TEST" />
  <property name="username" value="testuser" />
  <property name="password" value="testpasswd" />
</bean>
```

SPRING INTEGRATION:
6. SPRING CONFIGURATION (3)

```
<bean id="sessionFactory" class="org.springframework.orm.hibernate4.LocalS
  <property name="dataSource" ref="dataSource"></property>
  <property name="hibernateProperties">
    <props>
      <prop
       key="hibernate.dialect">org.hibernate.dialect.MySQL5Dialect</prop>
      <prop key="hibernate.show_sql">true</prop>
    </props>
  </property>
  <property name="packagesToScan" value="com.byteslounge.spring.tx.model"
</bean>
```

Spring Integration:
6. Spring Configuration (4)

```
<bean id="transactionManager"
  class="org.springframework.orm.hibernate4.HibernateTransactionManager"
  p:sessionFactory-ref="sessionFactory">
</bean>
```

Spring Integration:
6. Spring Configuration (5)

- Define beans
  - DAO beans
  - Service beans
- By XML or annotations

Spring Integration:
7. Using the Service (Manager) Beans

```
ApplicationContext ctx =
  new ClassPathXmlApplicationContext("spring.xml");
UserManager userManager =
  (UserManager) ctx.getBean("userManagerImpl");

User user = new User();
user.setUsername("johndoe");
user.setName("John Doe");

userManager.insertUser(user);
```

Generic DAO

- The problem with many DAO implementations
- Similar methods for
  - Load
  - Save, update, delete
  - Search
- The solution?

Generic DAO (cont'd)

```
public class GenericDaoHibernateImpl <T, PK extends Serializable>
    implements GenericDao<T, PK>, FinderExecutor {
    private Class<T> type;

    public GenericDaoHibernateImpl(Class<T> type) {
        this.type = type;
    }

    public PK create(T o) {
        return (PK) getSession().save(o);
    }

    public T read(PK id) {
        return (T) getSession().get(type, id);
    }

    public void update(T o) {
        getSession().update(o);
    }

    public void delete(T o) {
        getSession().delete(o);
    }
```

Hibernate Alternatives

- ORM
  - Enterprise JavaBeans (Entity Beans)
  - Java Data Objects (JDO)
  - Castor
  - Spring DAO
- Other JPA implementations
  - TopLink , OpenJPA, …