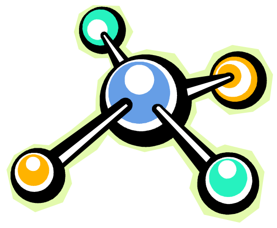
Introduction to Spring

And its use in OpenMRS

EHSDI

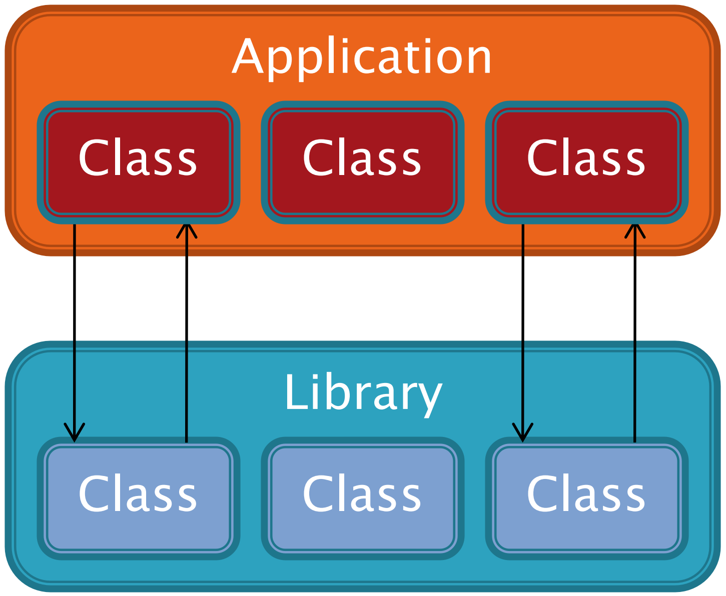
eBuzima

A framework?

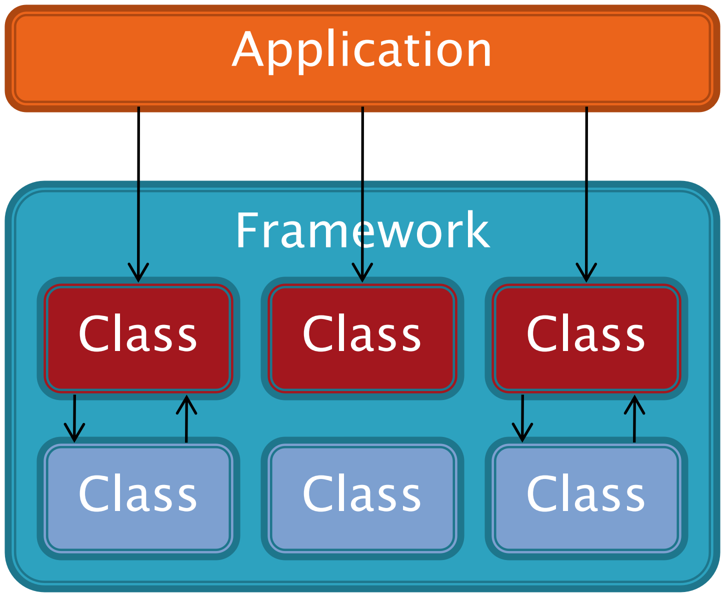


- A software framework provides a structure for an application
- Makes it easier to integrate with other technologies
- We add code to customize the framework for our specific application
- A framework runs our code, rather than our code running it, i.e. the Hollywood Principle: *"Don't call us, we'll call you"*

Library vs Framework



Application maintains control and calls library class methods as it needs



Application registers its classes with the framework and then gives it control

This called **inversion of control**

EHSDI
eBuzima

What is Spring?

- It's Open-source (Apache License)
- Written for .NET as well as Java
- First release in 2003

- Alternative to the Jakarta Struts framework
- Substitute for Enterprise Java Beans
- Designed to be modular – you only have to use the parts you need
- Designed to be easy to test

Spring features

- These are the main features of Spring being used by OpenMRS:

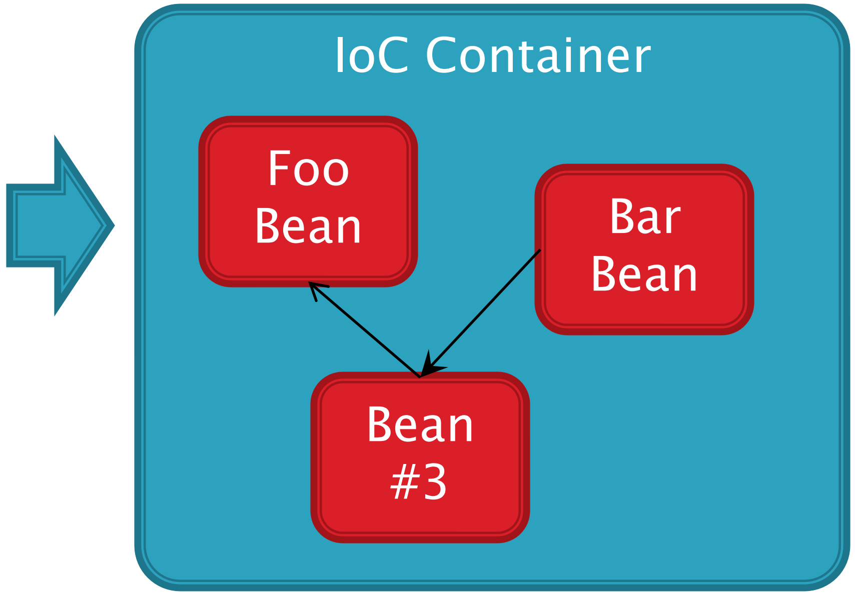
- Inversion of control container
- Aspect-oriented programming
- Data-access framework
- Model-view-controller framework
- Internationalization

Inversion of control container

- Responsible for
  - Creating objects (loads XML bean definitions)
  - Calling initialization methods
  - Configuring objects by wiring them together

```
<beans>
  <bean class="Foo">
    ...
  </bean>
  <bean class="Bar"></bean>
...
</beans>
```

This XML is called the ***application context***



Bean definitions

- Spring uses XML definitions to create beans, e.g.

Java class with bean properties

```
public class Person {
  ...
  public String getSurname() { ... }
  public void setSurname(String surname) { ... }
  public int getAge() { ... }
  public void setAge(int age) { ... }
}
```

XML file with bean definitions

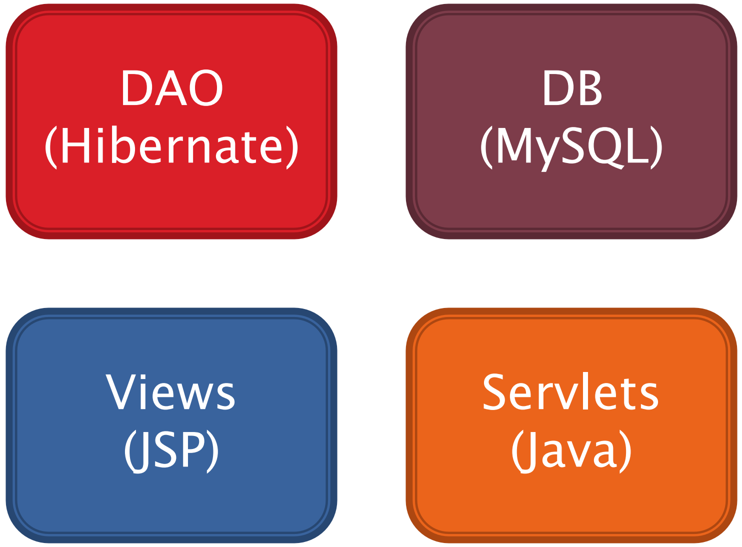
```
<beans>
  <bean id="admin" class="Person">
    <property name="surname" value="Seymour" />
    <property name="age" value="28" />
  </bean>
</beans>
```

Equivalent to...

```
Person admin = new Person();
admin.setSurname("Seymour");
admin.setAge(28);
```

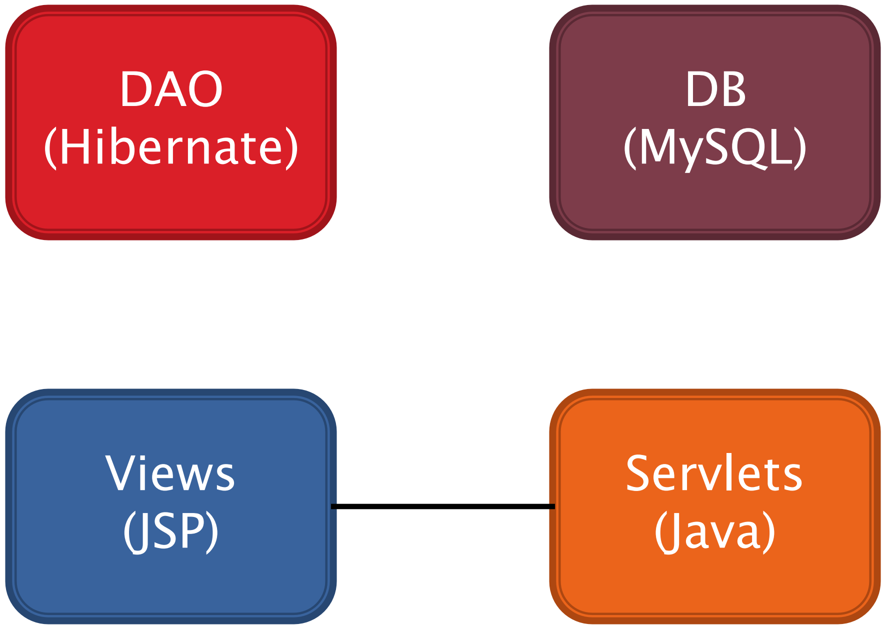
Dependencies

- Spring extracts dependencies into one location – the XML application context
- Supposing we have an typical application with several components...



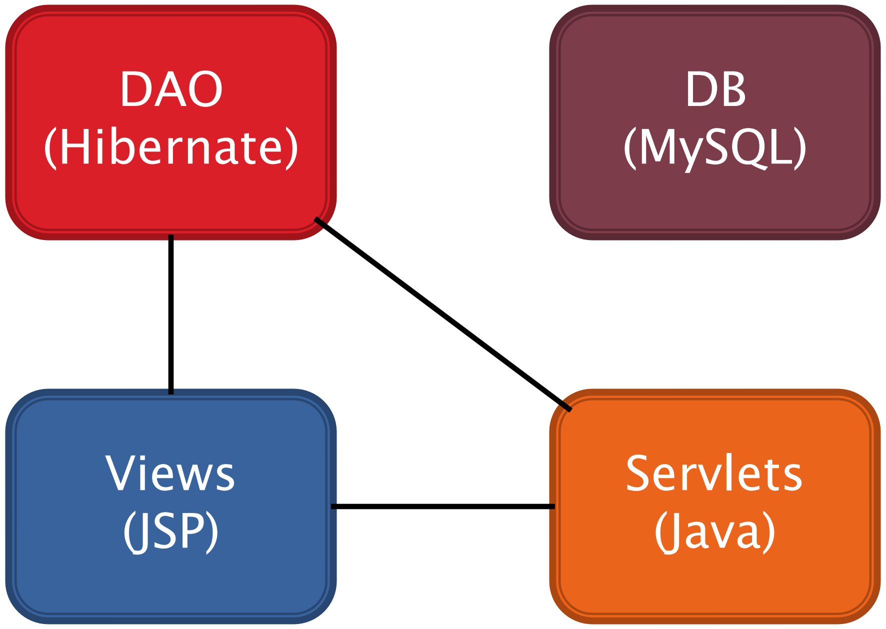
Dependencies

- Specifying JSP paths inside servlets creates a dependency between those two components, i.e. we can't change one without changing the other...



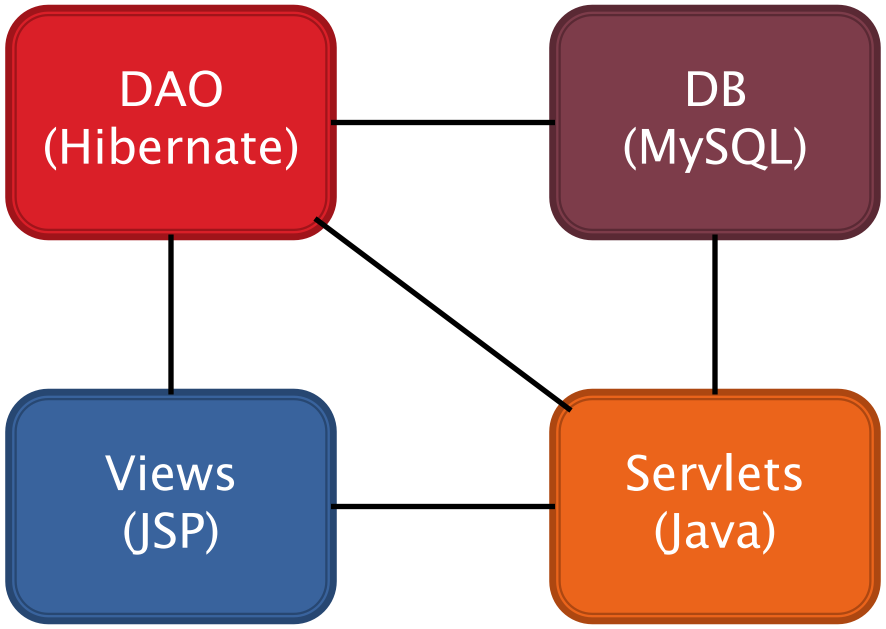
Dependencies

- Putting DAO code into a servlet or JSP creates more dependencies...



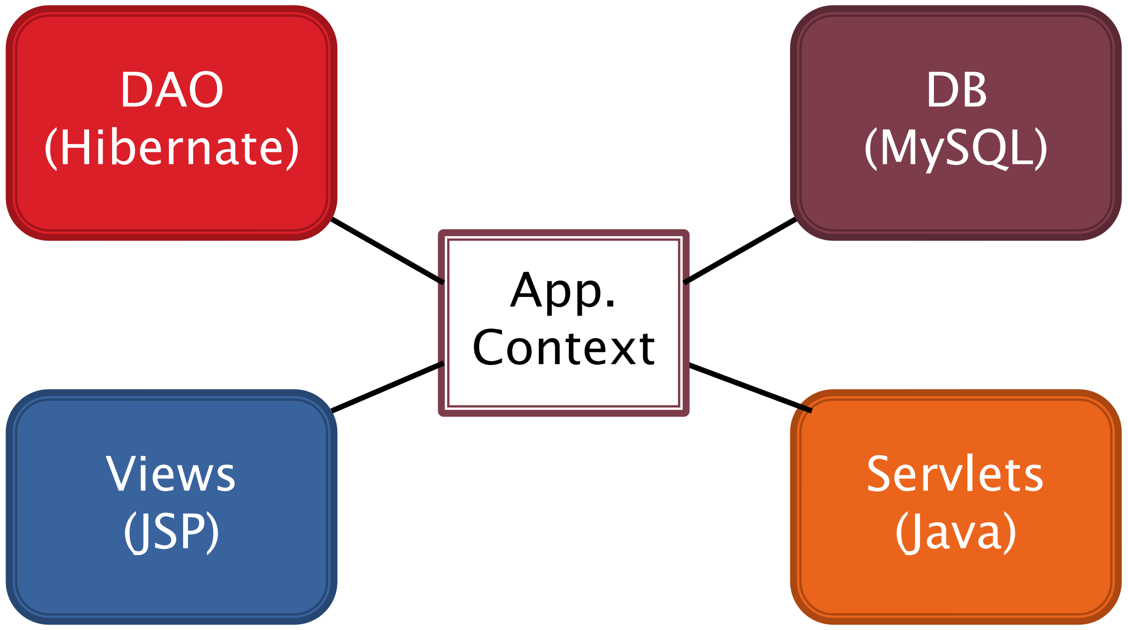
Dependencies

- Putting MySQL specific connection code into the DAO, using SQL in the servlet...

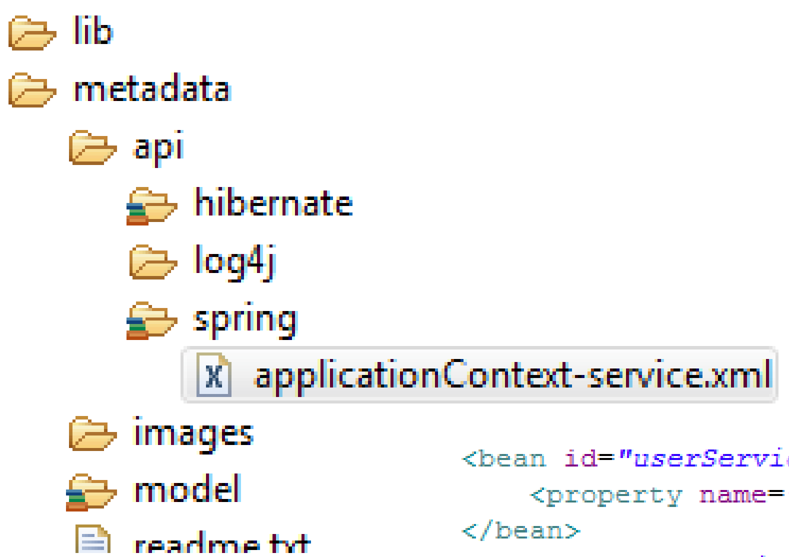


Dependencies

- Spring attempts to keep dependencies in one place – the application context – it connects all the different components as beans...



In OpenMRS…



lib
metadata
api
hibernate
log4j
spring
applicationContext-service.xml
images
model
readme.txt

- Beans are defined in an XML file

```
<bean id="userServiceTarget" class="org.openmrs.api.impl.UserServiceImpl">
    <property name="userDAO"><ref bean="userDAO"/></property>
</bean>
<bean id="obsServiceTarget" class="org.openmrs.api.impl.ObsServiceImpl">
    <property name="obsDAO"><ref bean="obsDAO"/></property>
    <property name="handlers">
        <map>
            <entry>
                <key><value>ImageHandler</value></key>
                <bean class="org.openmrs.obs.handler.ImageHandler"/>
            </entry>
            <entry>
                <key><value>TextHandler</value></key>
                <bean class="org.openmrs.obs.handler.TextHandler"/>
            </entry>
```

EHSDI
eBuzima

Aspect-oriented programming

- A typical application is made up of different *concerns* – areas of functionality, e.g. logging, authorization, data access
- Often these end up mixed in the same methods, e.g.

Logging concern

Authorization concern

```
public List<Patient> getAllPatients() {
  log.debug("Getting all patients");

  if (User.hasPrivilege("View Patients"))
    return null;

  return db.getAllPatients();
}
```

Aspect-oriented programming

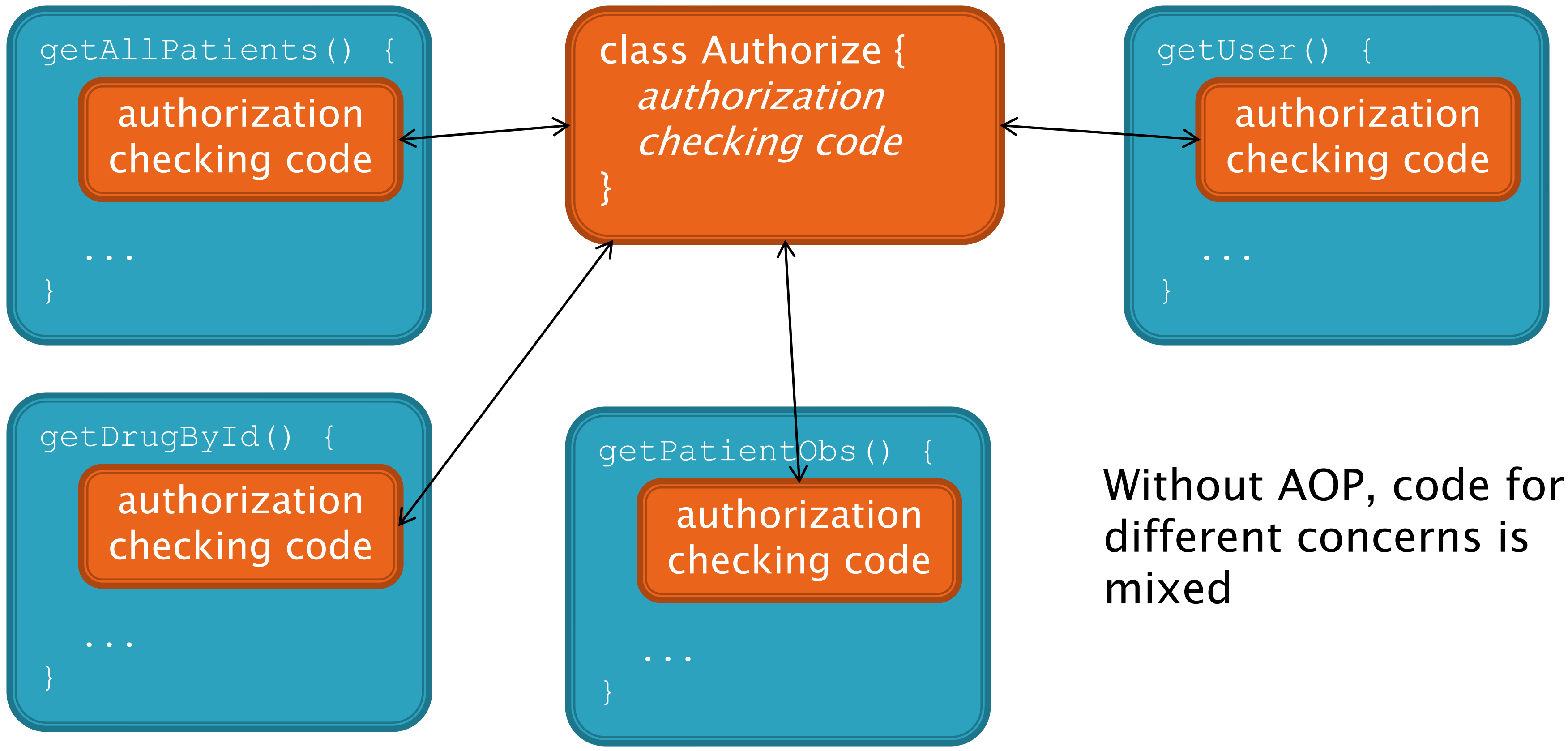
- Spring's AOP support allows us to configure code to be executed before or after multiple methods, e.g.
  - It's possible that many methods will start with an authorization check
  - We can define an annotation and tell Spring to execute a specific method when it encounters that annotation

```
@Authorized("View Patients")
public List<Patient> getAllPatients() {
  log.debug("Getting all patients");

  return db.getAllPatients();
}
```

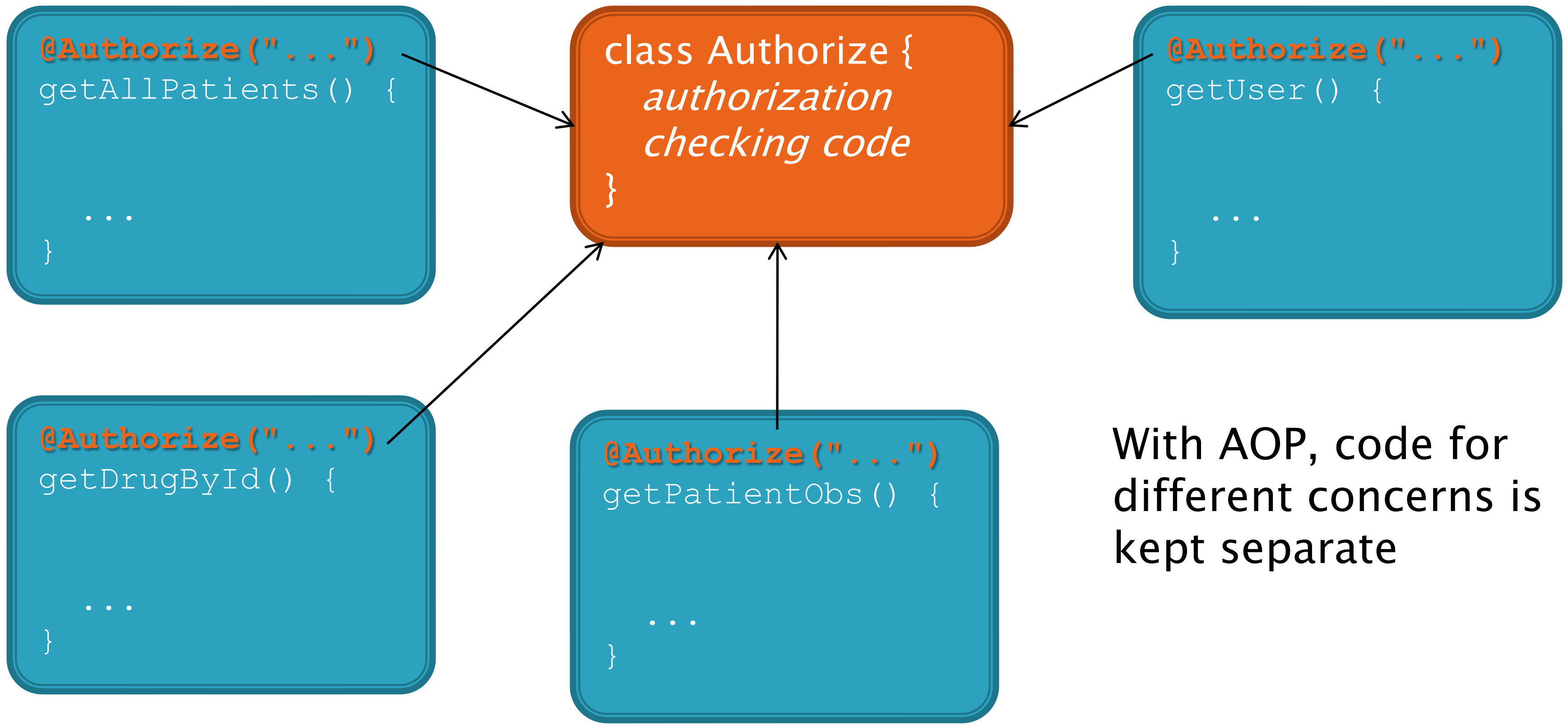
EHSDI
eBuzima

Aspect-oriented programming


getAllPatients() {
authorization checking code
...
}
class Authorize {
authorization checking code
}
getUser() {
authorization checking code
...
}
getDrugById() {
authorization checking code
...
}
getPatientObs() {
authorization checking code
...
}

Without AOP, code for different concerns is mixed

Aspect-oriented programming



```
@Authorize("...")
getAllPatients() {

   ...
}
```

```
class Authorize {
   authorization
   checking code
}
```

```
@Authorize("...")
getUser() {

   ...
}
```

```
@Authorize("...")
getDrugById() {

   ...
}
```

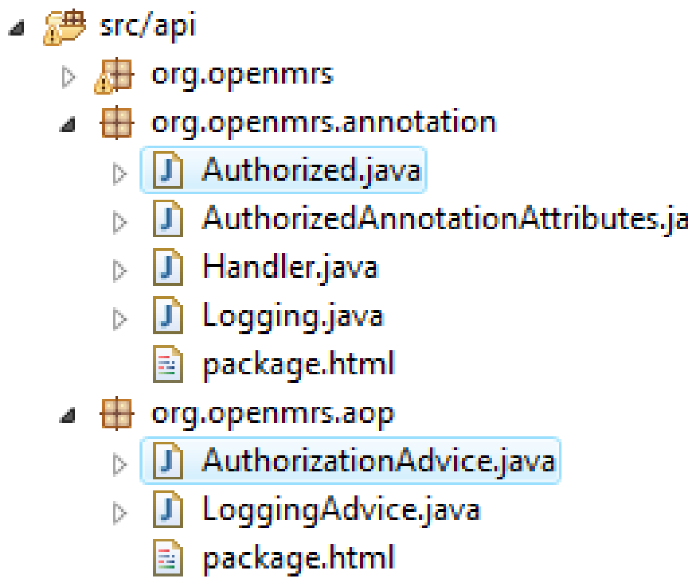
```
@Authorize("...")
getPatientObs() {

   ...
}
```

With AOP, code for different concerns is kept separate

`@Authorize("...")` is called a *join point*

In OpenMRS...



- *Join points* are set using annotations
- *Advice* is concern specific code to be executed at join points
- For example:
  - Authorized annotation executes a method in AuthorizationAdvice

EHSDI
eBuzima

Data-access framework

- Spring is designed to work with many different database frameworks, e.g.
  - Hibernate
  - JDBC
  - JDO
  - Oracle Toplink
- For each of these Spring offers
  - Resource management
  - Transaction management
  - Exception translation

Data-access framework

- Resource management means that Spring can acquire and release the data framework objects for us
- Data connection is configured as a bean, e.g.

```
<bean id="myDataSource" class="org.apache.commons.dbcp.BasicDataSource">
   <property name="driverClassName" value="org.gjt.mm.mysql.Driver" />
   <property name="url" value="jdbc:mysql://localhost/" />
   <property name="defaultCatalog" value="intranet" />
   <property name="username" value="admin" />
   <property name="password" value="test" />
 </bean>
```

Data-access framework

- Even the different data frameworks can be configured in a consistent way, as beans, e.g.

```
<bean id="sessionFactory" class="org.springframework.orm.hibernate3.LocalSessionFactoryBean">
    <property name="dataSource" ref="myDataSource"/>
    <property name="mappingResources">
      <list>
        <value>ehsdi/intranet/api/db/hibernate/User.hbm.xml</value>
        <value>ehsdi/intranet/api/db/hibernate/Role.hbm.xml</value>
        <value>ehsdi/intranet/api/db/hibernate/Privilege.hbm.xml</value>
        <value>ehsdi/intranet/api/db/hibernate/Patient.hbm.xml</value>
      </list>
    </property>
    <property name="hibernateProperties">
      <value>
        hibernate.dialect=org.hibernate.dialect.MySQL5Dialect
        hibernate.show_sql=false
      </value>
    </property>
</bean>
```

Data-access framework

- Transaction management means that transactions can be wrapped around method calls automatically
- For example, using Hibernate without Spring's transaction management, we must explicitly create transactions:

```java
private List<Person> getAllPersons() {
  Session session = HibernateUtil.getSessionFactory().getCurrentSession();
  session.beginTransaction();

  List<Person> persons = session.createCriteria(Person.class).list();
  session.getTransaction().commit();
  return persons;
}
```

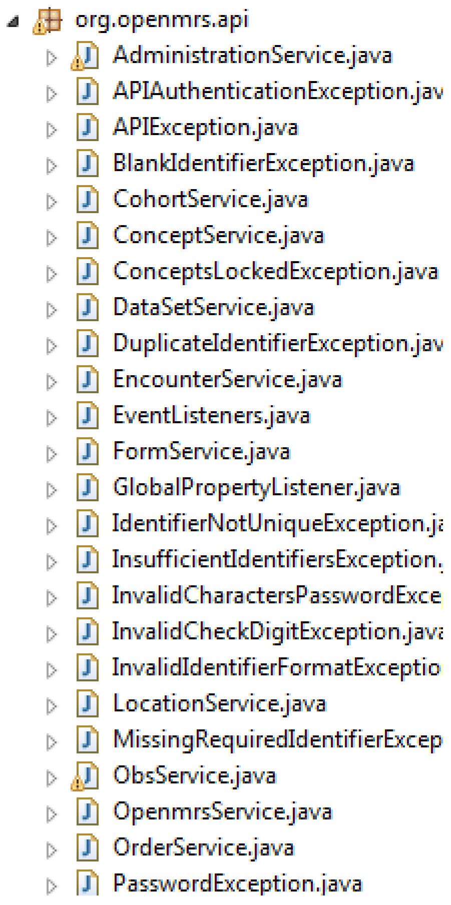
Data-access framework

- Using Spring's annotation driven transaction management, we can have all the methods of a class, wrapped in transactions

```
@Transactional
class PatientService {
  ...
  private List<Person> getAllPersons() {
    List<Person> persons =
            session.createCriteria(Person.class).list();
    return persons;
  }
  ...
}
```

EHSDI
eBuzima

In OpenMRS…



```
org.openmrs.api
  AdministrationService.java
  APIAuthenticationException.jav
  APIException.java
  BlankIdentifierException.java
  CohortService.java
  ConceptService.java
  ConceptsLockedException.java
  DataSetService.java
  DuplicateIdentifierException.jav
  EncounterService.java
  EventListeners.java
  FormService.java
  GlobalPropertyListener.java
  IdentifierNotUniqueException.ja
  InsufficientIdentifiersException.
  InvalidCharactersPasswordExce
  InvalidCheckDigitException.java
  InvalidIdentifierFormatExceptio
  LocationService.java
  MissingRequiredIdentifierExcep
  ObsService.java
  OpenmrsService.java
  OrderService.java
  PasswordException.java
```

- OpenMRS has a service layer
- Contains classes which interact with the data access layer (Hibernate)
- All of the service classes implement an equivalent interface in `org.openmrs.api`
- These service interfaces use the `@Transactional` annotation

EHSDI
eBuzima

In OpenMRS...

Layer	Package / JAR	Example
Services	org.openmrs.api	PatientService
	org.openmrs.api.impl	PatientServiceImpl
DAO	org.openmrs.db	PatientDAO
	org.openmrs.db.hibernate	HibernatePatientDAO
Hibernate	hibernateXX.jar	
JDBC	java.sql	
MySQL	mysql-connector-java-XX.jar	

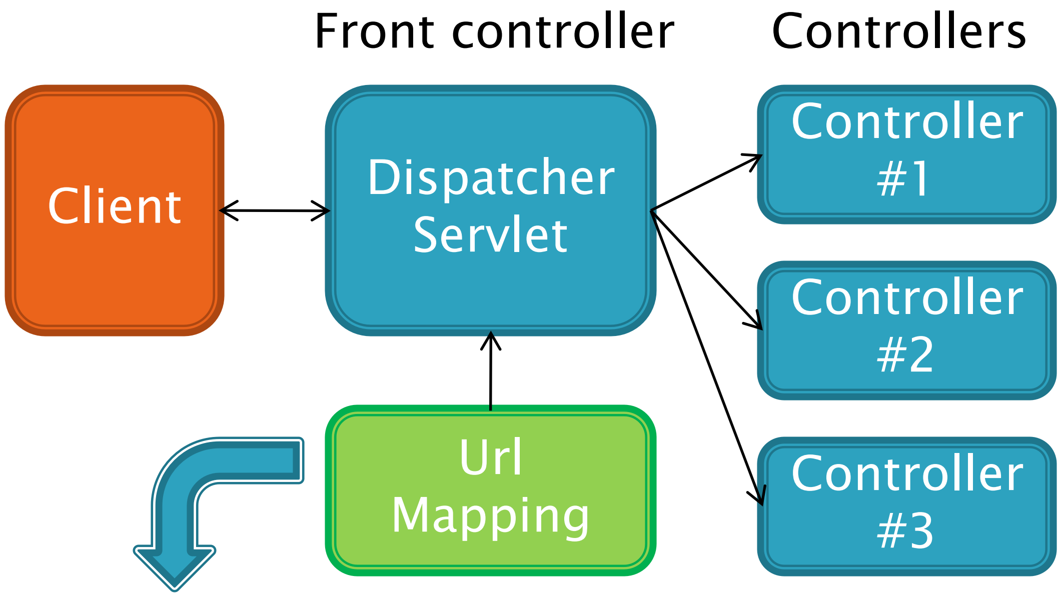
Model–view–controller framework

- Enforces separation of the three components
- Defines a set of interfaces to be implemented
- Not limited to JSP based views

- Controllers are configured as beans like everything else, and have access to other beans
- A controller is not a servlet!

Front controller / URL Mapping

- Spring uses a *front controller* model
- All requests have a single entry point

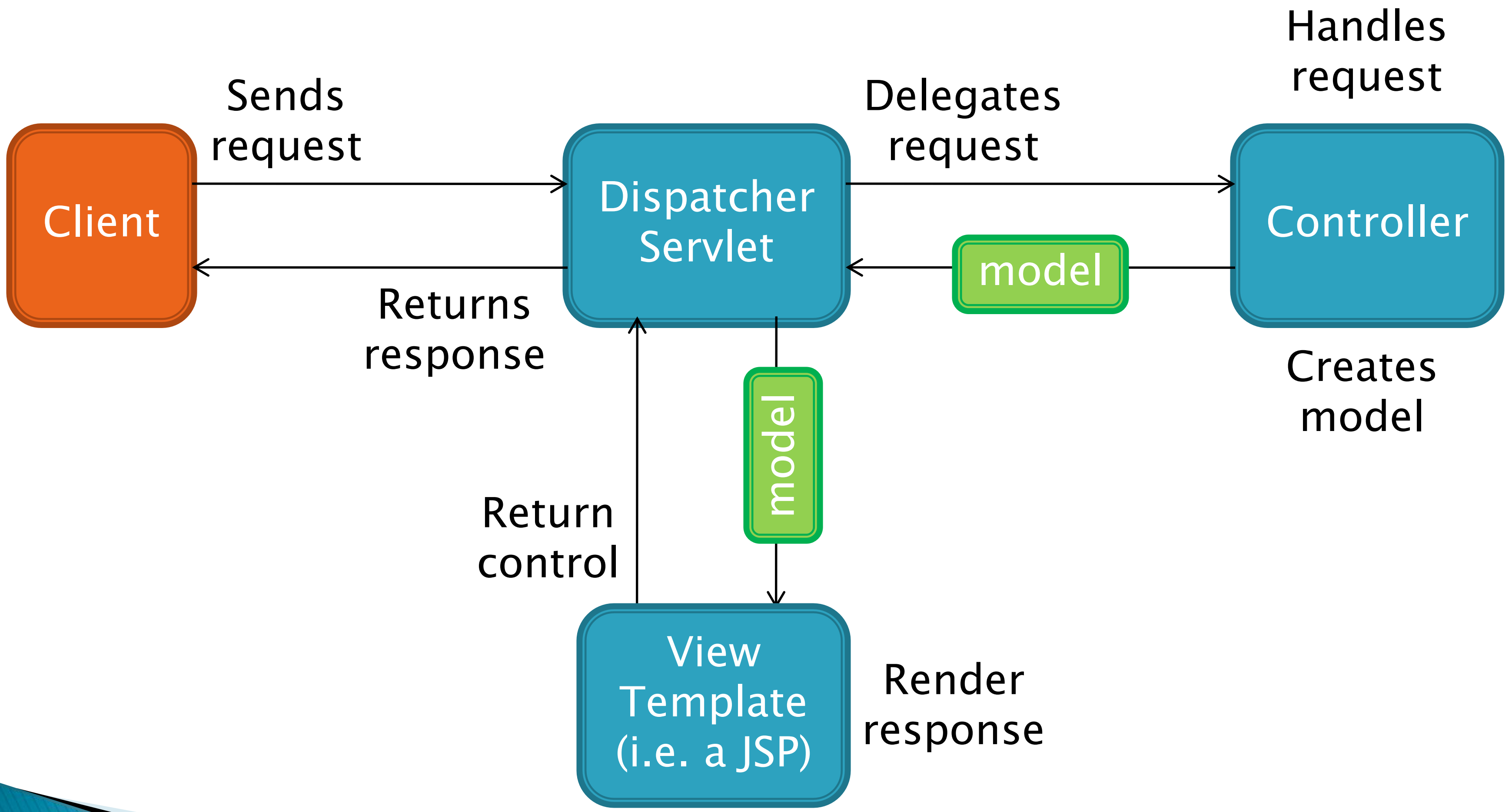


```
<bean id="urlMapping" class="org.springframework...SimpleUrlHandlerMapping">
 <property name="mappings">
  <props>
   <prop key="admin/patients/patient.form">patientForm</prop>
   <prop key="admin/patients/newPatient.form">newPatientForm</prop>
   <prop key="admin/patients/mergePatients.form">mergePatientsForm</prop>
   ...
  </props>
 </property>
</bean>
```

URLs

Controllers

Spring MVC workflow



The Controller interface

- A controller should handle a request by returning a model and a view
- Thus the controller interface is very simple:

```
public interface Controller {
  public ModelAndView handleRequest(
    HttpServletRequest request,
    HttpServletResponse response
  ) throws Exception;
}
```

The View interface

- A view should render a response, given a model
- Thus the view interface is also very simple:

```
public interface View {
  public void render(
    Map model,
    HttpServletRequest request,
    HttpServletResponse response
  ) throws Exception
}
```

EHSDI
eBuzima

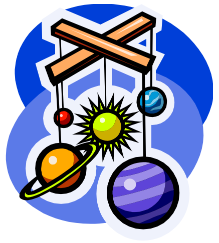
Models

- Models in Spring MVC are usually maps, e.g.

```
Map model = new HashMap();
model.put("patient", patient);
model.put("encounter", encounter);
```

- Map entries automatically become request attributes, accessible in a JSP, e.g.

```
${patient.surname}
${encounter.location}
```



EHSDI
eBuzima

ModelAndView

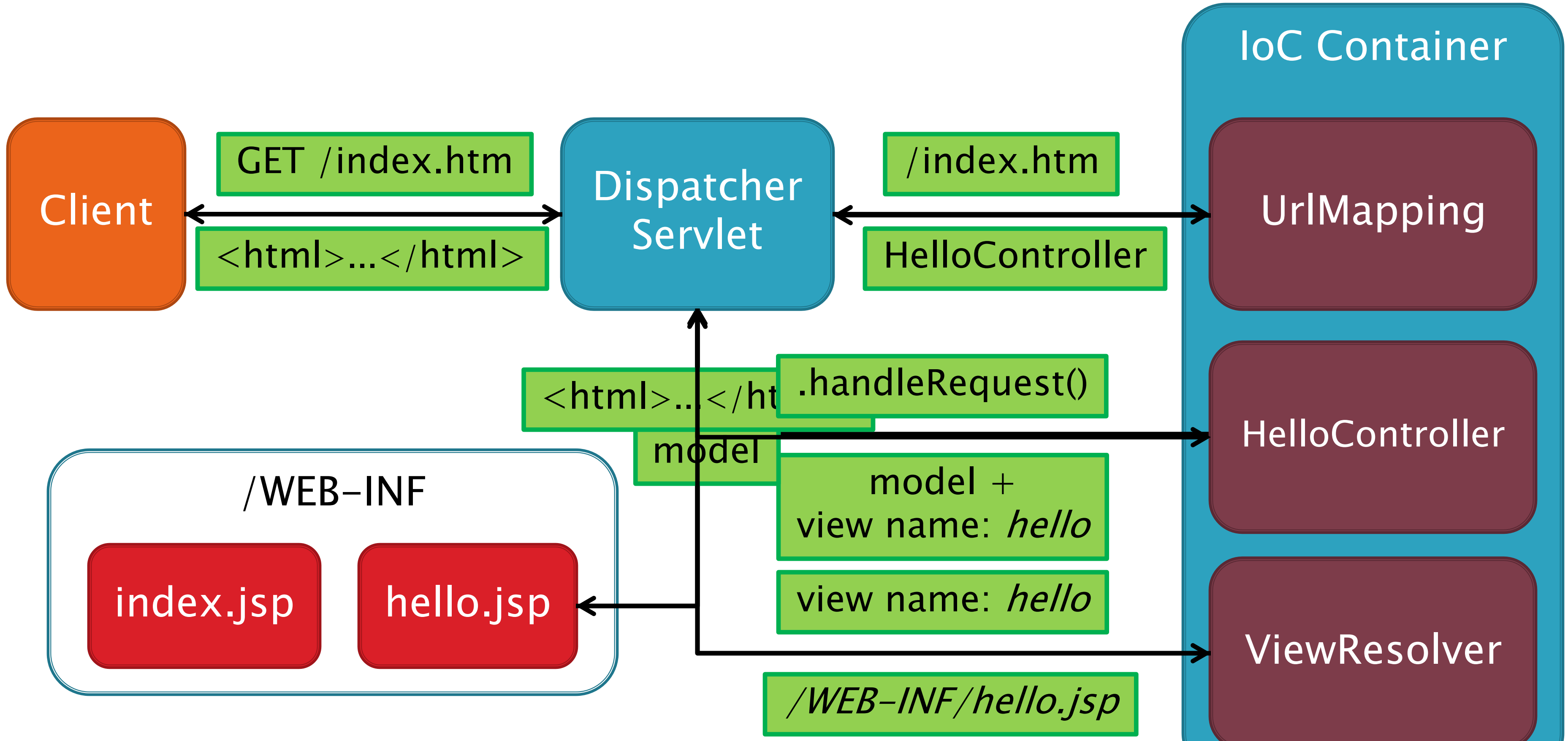
- This is used to pass the model between the controller and the view
- It is constructed by specifying a view and a model
- View can be specified by name, which is then resolved to a JSP by the dispatcher servlet

```
Map model = new HashMap();
model.put("patients", patients);

return new ModelAndView("patientList", model);
```

EHSDI
eBuzima

Example: Spring MVC



Controller classes

- It's unusual to implement Controller directly – usually we extend of one Spring's predefined controller classes, which offer extra functionality, e.g.
  - `AbstractController`
    - Generates cache headers
    - Can be configured to accept/refuse GET or POST
  - `ParameterizableViewController`
    - View name can be configured as bean property

Controller classes

- When using one of these classes, override `handleRequestInternal` instead of `handleRequest`

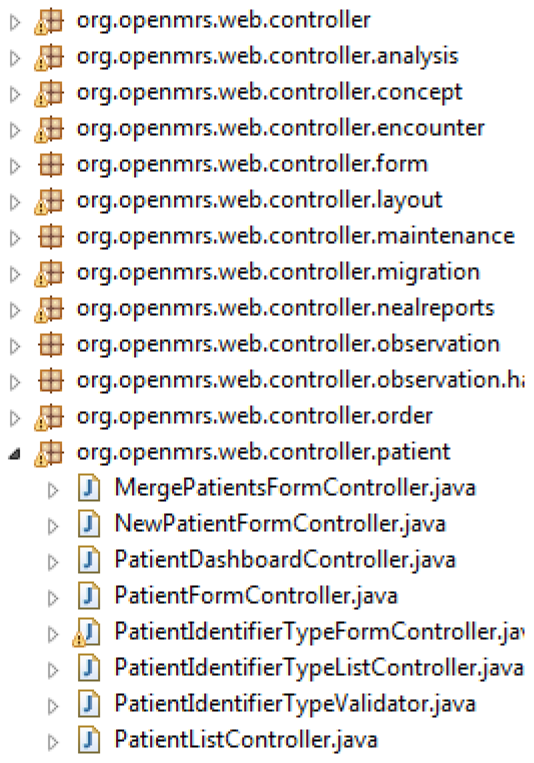
Predefined spring controller class

```
class AbstractController implements Controller {
  ...
  public ModelAndView handleRequest() {
    // Sets up cache headers etc
    ...
    return handleRequestInternal();
  }
  protected abstract ModelAndView handleRequestInternal();
}
```

Our custom controller

```
class HelloController extends AbstractController {
  protected ModelAndView handleRequestInternal() {
    return new ModelAndView("hello");
  }
}
```

In OpenMRS…

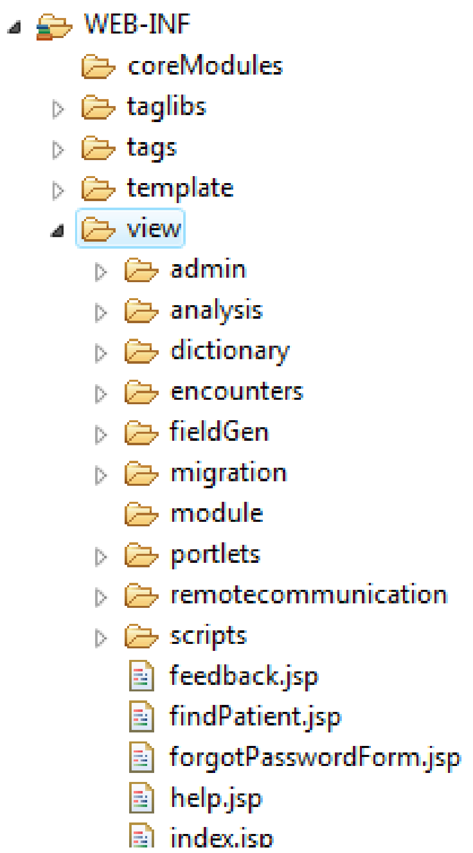


```
org.openmrs.web.controller
org.openmrs.web.controller.analysis
org.openmrs.web.controller.concept
org.openmrs.web.controller.encounter
org.openmrs.web.controller.form
org.openmrs.web.controller.layout
org.openmrs.web.controller.maintenance
org.openmrs.web.controller.migration
org.openmrs.web.controller.nealreports
org.openmrs.web.controller.observation
org.openmrs.web.controller.observation.h
org.openmrs.web.controller.order
org.openmrs.web.controller.patient
    MergePatientsFormController.java
    NewPatientFormController.java
    PatientDashboardController.java
    PatientFormController.java
    PatientIdentifierTypeFormController.ja
    PatientIdentifierTypeListController.java
    PatientIdentifierTypeValidator.java
    PatientListController.java
```

- Controller classes are in subpackages of `org.openmrs.web.controller`
- These are instantiated as beans in `WEB-INF/openmrs-servlet.xml`
- This file also defines a URL mapping bean, which maps URLs to controllers

EHSDI
eBuzima

In OpenMRS…


WEB-INF
coreModules
taglibs
tags
template
view
admin
analysis
dictionary
encounters
fieldGen
migration
module
portlets
remotecommunication
scripts
feedback.jsp
findPatient.jsp
forgotPasswordForm.jsp
help.jsp
index.jsp

- Views are JSP files, stored in *web/WEB-INF/view*

References

- Websites
  - http://www.springsource.org/
  - http://en.wikipedia.org/wiki/Spring_Framework
  - http://www.theserverside.com/tt/articles/article.tss?l=SpringFramework