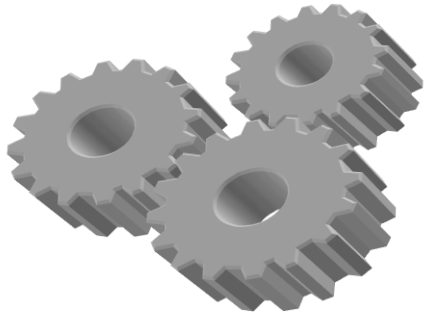
Module 16: Hibernate Applications

CS544: Enterprise Architecture

Integrating Hibernate

- In this module we will discuss integrating Hibernate in a more real world application
  - Use HibernateUtil
    - to provide a singleton session factory
  - Avoid session per operation by:
    - Thread Local / Current Session Pattern
      - Session per Service Level request
    - Open Session in View Pattern
      - Session per request (from the browser)



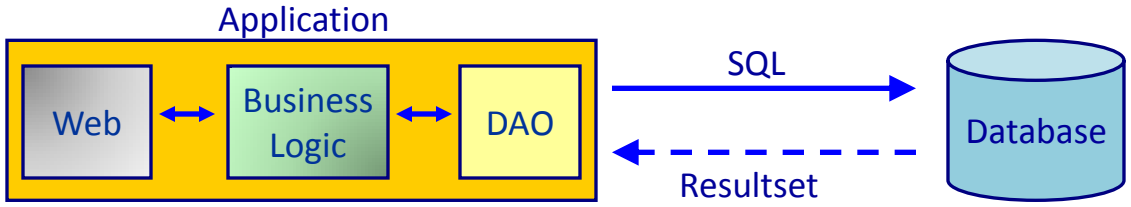
Examples so Far

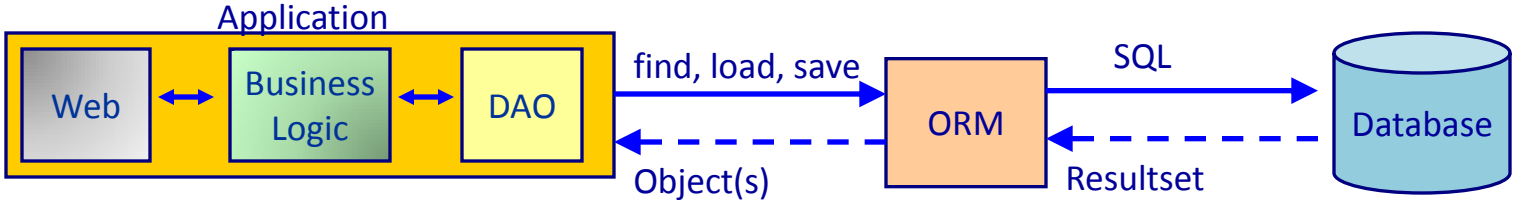
```
public class Application {
    private static SessionFactory sessionFactory;

    private static SessionFactory sessionFactory = new Configuration()
        .configure().buildSessionFactory();

    public static void main(String[] args) {
        Session session = null;
        Transaction tx = null;

        try {
            session = sessionFactory.openSession();
            tx = session.beginTransaction();

            Employee employee = new Employee();
            employee.setFirstname("Frank");
            employee.setLastname("Brown");
            session.persist(employee);

            tx.commit();
        } catch (HibernateException e) {
            tx.rollback();
            e.printStackTrace();
        } finally {
            if (session != null)
                session.close();
        }
    }
}
```

Unrealistic use of Hibernate. Hibernate only used inside public static void main()

Seperation of Concerns
Data Access Objects

- Hibernate will most likely be used in DAOs
- DAOs are a common design pattern
  - Separates business logic from data access logic
  - Used both with and without an ORM solution





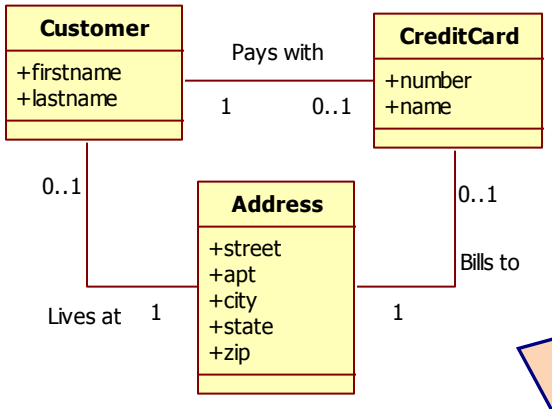
Problems and Solutions

1. Examples start the SessionFactory every time
   - – Takes a lot of time, application only needs it once
   - ✔ Create a SessionFactory Singleton (HibernateUtil)
2. Examples open close the Session/TX ever time
   - – Session Per Operation Anti-Pattern
   - – Non transactional (not atomic) behavior
   - ✔ The thread local pattern fixes these issues
3. Thread local links session to the transaction
   - ■ Transaction demarcation belongs at the service level
   - – Session is closed in View: LazyInitializationException
   - ✔ Open Session in View pattern fixes LazyInitialization

Example Application

Application Layers

- View Layer:
  - addCustomer.jsp, customer.jsp, customers.jsp, updaCustomer.jsp, error.jsp
- Control Layer:
  - ViewAllCustomers, ViewCustomer, AddCustomer, ViewUpdCustomer, UpdCustomer
- Service Layer:
  - Customer Service
- Business Layer:
  - Customer, Address, CreditCard
- Persistence Layer:
  - CustomerDAO, AddressDAO, CreditCardDAO



Example Application to keep track of customers, their creditcards, shipping and billing addresses

Hibernate specific code should only be needed here

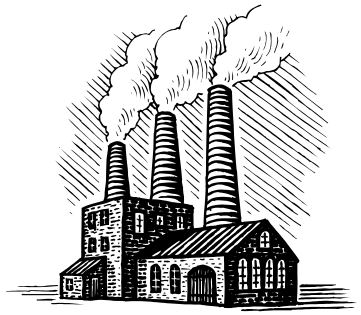
Hibernate Applications:

HIBERNATE UTIL

Session Factory

- The Session Factory is expensive to create
  - Doing so essentially ‘starts’ Hibernate
  - We generally want to start it only once

- We need a session factory whenever we use Hibernate (to make Sessions)
  - It needs to be available everywhere
  - We need a SessionFactory singleton!



HibernateUtil

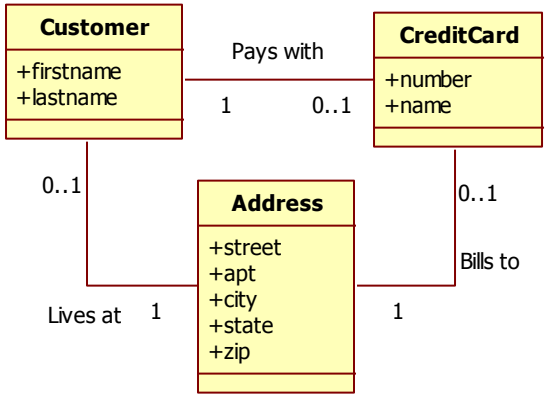
- The HibernateUtil class creates a SessionFactory and makes it available as a Singleton

```
public class HibernateUtil {
  private static final SessionFactory sessionFactory;

  static {
    try {
      // Create the SessionFactory from hibernate.cfg.xml
      Configuration configuration = new Configuration();
      configuration.configure();
      ServiceRegistry serviceRegistry =
          new StandardServiceRegistryBuilder().applySettings(
                        configuration.getProperties()).build();
      sessionFactory = configuration.buildSessionFactory(serviceRegistry);
    } 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);
    }
  }

  public static SessionFactory getSessionFactory() {
    return sessionFactory;
  }
}
```

Example Application

Application Layers

- View Layer:
  - addCustomer.jsp, customer.jsp, customers.jsp, updaCustomer.jsp, error.jsp
- Control Layer:
  - ViewAllCustomers, ViewCustomer, AddCustomer, ViewUpdCustomer, UpdCustomer
- Service Layer:
  - Customer Service
- Business Layer:
  - Customer, Address, CreditCard
- Persistence Layer:
  - CustomerDAO, AddressDAO, CreditCardDAO, **HibernateUtil**



HibernateUtil is added to the persistence Layer

Using only HibernateUtil

- We could make code similar to examples so far
  - Retrieving the SessionFactory whenever needed

```
public class AddressDAO {

  public void create(Address addr) {
    Session session = null;
    Transaction tx = null;
    try {
      session = HibernateUtil.getSessionFactory().openSession();
      tx = session.beginTransaction();
      session.saveOrUpdate(addr);
      tx.commit();
    } catch (Exception e) {
      tx.rollback();
      e.printStackTrace();
    } finally {
      session.close();
    }
  }

  ...
```

HibernateUtil allows us to use a Single SessionFactory for all our DAO classes

But if we copy-paste Hibernate code like our examples (and many examples in the online docs) into each DAO method...

Bad Integration

- You can create a fully functioning application using Hibernate this way
  - But you'll have plenty of problems:

1. Code Problems:
   - Lots of nearly identical code in DAO methods
   - Essentially just copy / pasted

2 Session Problems

- DAOs open and close a session per operation
  - Session Cache is never used
  - Retrieved objects are immediately detached
  - No way to automatically load related entities
    - Need to separately load entities – similar to JDBC Code
    - Not doing so just creates Lazy initialization exceptions

- This is such a bad way of using Hibernate it is called the Session-per-Operation anti-pattern

3 Transaction Problems

- Because a new session is opened and closed each time, a new tx is also opened and closed
  - Transaction only spans a single operation
  - Transaction never spans a unit of work
  - Essentially creating non-transactional behavior

4 Exception Problems

- Exception is handled inside the lowest layer
  - DOAs are located in the persistence layer
    - This implementation prints a stacktrace to the log
    - The application user is left in the dark

  - Exceptions are handled best in the control layer
    - Allowing the controller to chose a different view
    - Clearly informing the user that an error has occurred
      - Perhaps even giving advice on how to prevent or fix it

Hibernate Applications:

THREAD LOCAL SESSION PATTERN

Longer Running Session

- One way to solve the Session Per Operation problem could be to create a session in the controller and pass it into every method call

- This could work but would require us to:
  - Change the methods on all Service and DAO classes
  - Thereby tightly coupling Hibernate to multiple application layers, making our application inflexible

Current Session

- Hibernate can provide a Thread 'current session'
  - So that we don't have to pass the Session around

```
<hibernate-configuration>
  <session-factory>
    ...

    <!-- Use the thread local session pattern -->
    <property name="current_session_context_class">thread</property>

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

Configuration Property in Hibernate.cfg.xml

- Simply get the current session:

```
Session session = sessionFactory.getcurrentSession()

// Combined with HibernateUtil
Session session = HibernateUtil.getSessionFactory().getCurrentSession();
```

Scope of the ‘Current Session’

- The first time getCurrentSession() is called a new Session is created and returned
- Any sequential calls return the same session
- On Transaction.commit() the session is flushed and closed automatically
- If another call to getCurrentSession() is made after commit, a new Session is created
- The Transaction and the Session therefore have roughly the same scope

Unit of Work Transactions

- Create Transactions that span a unit of work
  - Using HibernateUtil and the ThreadLocal

```
public class CustomerService {
  private CustomerDAO customerDao = new CustomerDAO();
  private AddressDAO addressDao = new AddressDAO();
  private CreditCardDAO ccDao = new CreditCardDAO();
  private SessionFactory sf = HibernateUtil.getSessionFactory();

  public void addNewCustomer(Customer cust, Address shipAddr, CreditCard cc,
                             Address billAddr) {
    cc.setAddress(billAddr);
    cust.setShipAddress(shipAddr);
    cust.setCreditCard(cc);

    Transaction tx = sf.getCurrentSession().beginTransaction();
    addressDao.create(shipAddr);
    addressDao.create(billAddr); // does nothing if ship == bill
    ccDao.create(cc);
    customerDao.create(cust);
    tx.commit();
  }

  ...
```

Transaction per Service method

With ThreadLocal all these DAO methods will use the same session

Controller Exception Handling

```
public class ViewAllCustomers extends HttpServlet {
  private static final long serialVersionUID = 1L;

  public void doGet(HttpServletRequest req, HttpServletResponse resp)
  throws ServletException, IOException {

    try {
      CustomerService custServ = new CustomerService();
      req.setAttribute("customers", custServ.getAll());

      // forward to customers page
      req.getRequestDispatcher("customers.jsp").forward(req, resp);

    } catch (RuntimeException ex) {
      HibernateUtil.getSessionFactory().getCurrentSession()
            .getTransaction().rollback();
      ex.printStackTrace();

      // forward to error page
      req.setAttribute("exception", ex);
      req.getRequestDispatcher("error.jsp").forward(req, resp);
    }
  }
}
```

Handling the Exception here allows us to notify the user

DAO using getCurrentSession()

- Using ThreadLocal Pattern, TX per Service Method, Controller Exception Handling

```
public class AddressDAO {
  private SessionFactory sf = HibernateUtil.getSessionFactory();

  public void create(Address addr) {
    sf.getCurrentSession().persist(addr);
  }

  public Address get(int id) {
    return (Address) sf.getCurrentSession().get(Address.class, id);
  }

  public void update(Address addr) {
    sf.getCurrentSession().saveOrUpdate(addr);
  }

  public void delete(Address addr) {
    sf.getCurrentSession().delete(addr);
  }
}
```

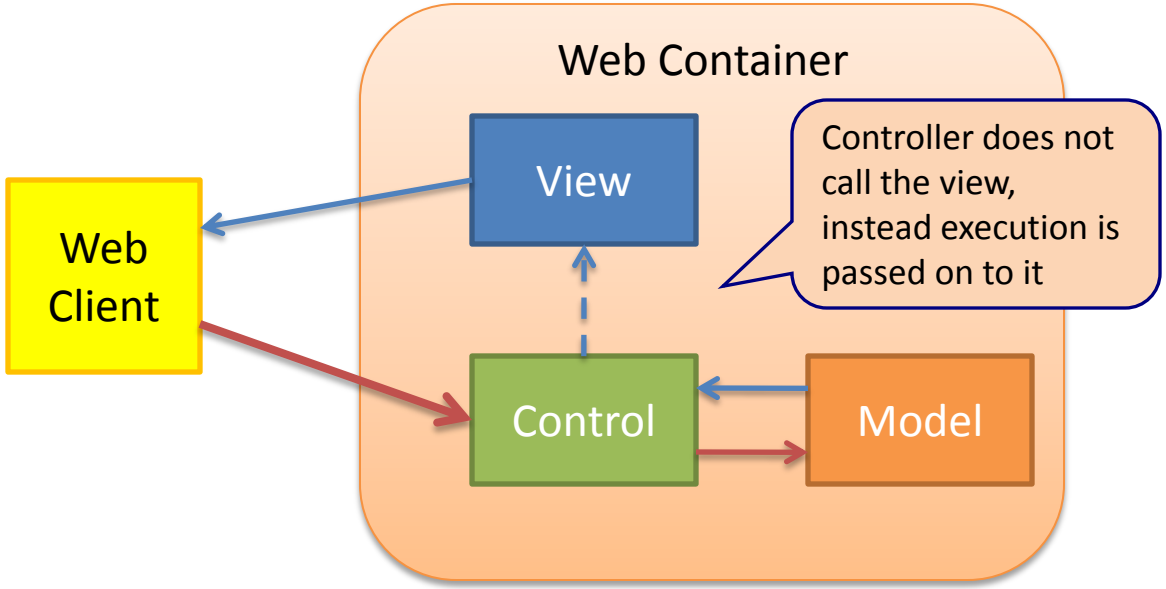
Interesting Points

1. Controllers Service classes still contain Hibernate Specific Transaction Code
   - Can be fixed by using JTA transactions instead

2. Need to eagerly load related data so that the view has all the data it needs for rendering
   - View can not use automatic loading of related objects
   - Transactions are committed after each service level method, causing the session to be closed
   - Even if we put the transaction around the entire controller we would still have the same problem
     - Transaction would still commit before view rendering

Reason to Eager Load

- Controller is what starts executing ‘our’ code
  - Controller does not call the view, it finishes and delegates control to it
  - All data needed by the view has to be loaded before access to the session is finished



Eager Loading

- We discussed two forms of Eager Loading during the Optimization Module
  - Always eager join mapping
    - Has many issues, not the best for this situation
  - Eager Join Fetch Queries
    - If we use these in our DAO methods the relation will always be loaded when that method is used
    - Might retrieve data when not needed

- There is also a third form of eager Loading:
  - **Hiberante.Initialize()** – manually initialize a proxy
  - Very clearly shows the programmer's intention

Service Level Hibernate.Initialize()

```
public class CustomerService {
  ...

  public Customer getCust(int custId) {
    Transaction tx = sf.getCurrentSession().beginTransaction();
    Customer cust = customerDao.get(custId);

    // make sure associated entities are loaded
    Hibernate.initialize(cust.getShipAddress());
    Hibernate.initialize(cust.getCreditCard());
    Hibernate.initialize(cust.getCreditCard().getAddress());

    tx.commit();
    return cust;
  }

  ...
}
```

Used inside TX since we can only initialize proxies of managed objects

More Hibernate Specific Code

Could have also just used cust.getShipAddress() without initialize() but our intention would not be clear

What if some calls to getCust() don't need all these related objects?

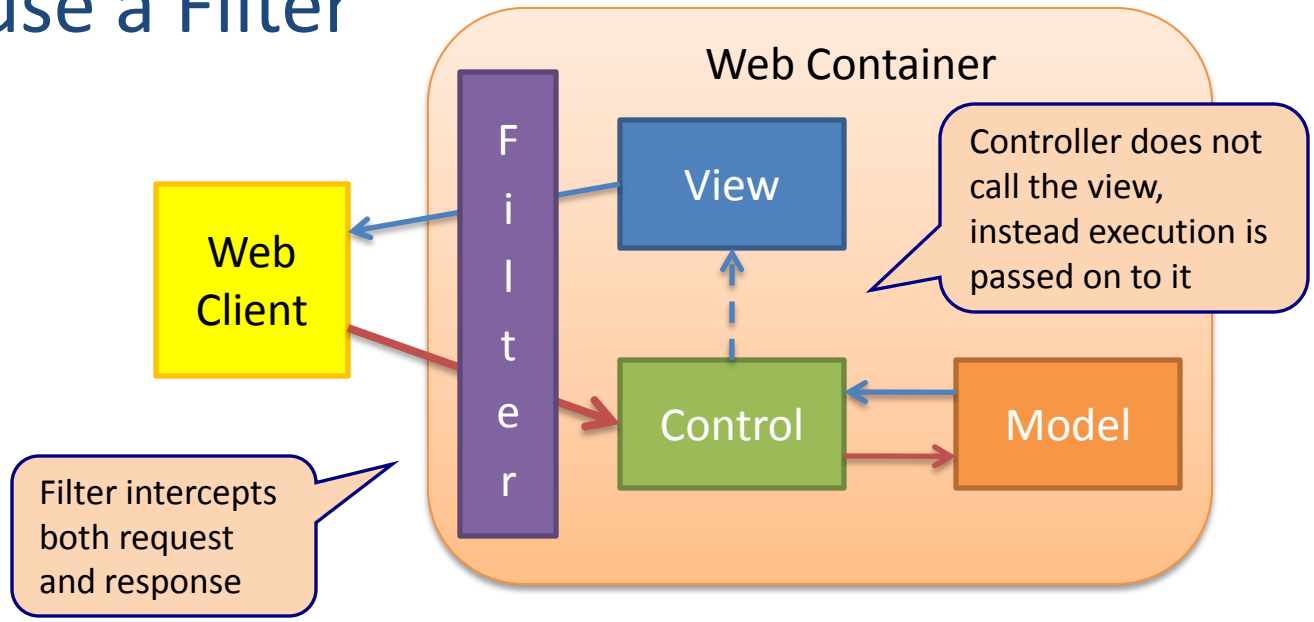
- Despite these interesting points:
  - ThreadLocal Session Pattern provides reasonable integration of Hibernate into a Web Application

Hibernate Applications:

OPEN SESSION IN VIEW PATTERN

Filter

- In order to keep the session open during the execution of the controller and the view we use a Filter



- Opens the transaction before the controller begins, closes it after the view finishes

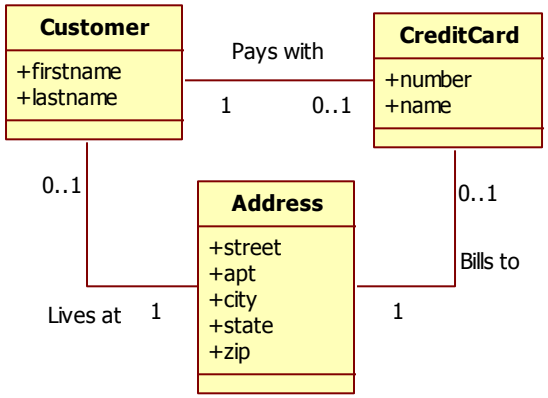
Example Application

Application Layers

- View Layer:
  - addCustomer.jsp, customer.jsp, customers.jsp, updaCustomer.jsp, error.jsp, **OpenSessionFilter**

OpenSessionFilter is added to the View Layer

- Control Layer:
  - ViewAllCustomers, ViewCustomer, AddCustomer, ViewUpdCustomer, UpdCustomer
- Service Layer:
  - Customer Service
- Business Layer:
  - Customer, Address, CreditCard



- Persistence Layer:
  - CustomerDAO, AddressDAO, CreditCardDAO, HibernateUtil

Web.xml

```
<web-app>
  ...

  <filter>
    <filter-name>OpenSessionInView</filter-name>
    <filter-class>example.filter.OpenSessionInView</filter-class>
  </filter>

  <filter-mapping>
    <filter-name>OpenSessionInView</filter-name>
    <url-pattern>/*</url-pattern>
  </filter-mapping>

  ...

  <error-page>
    <exception-type>org.hibernate.HibernateException</exception-type>
    <location>/hibernateError.jsp</location>
  </error-page>
</web-app>
```

OpenSessionInView filter mapped to all requests

Custom error page selection done in web.xml

OpenSessionInView Filter

```
public class OpenSessionInView implements Filter {
  private SessionFactory sf;

  public void init(FilterConfig arg0) throws ServletException {
    sf = HibernateUtil.getSessionFactory();
  }
  public void destroy() {}

  public void doFilter(ServletRequest req, ServletResponse resp,
        FilterChain chain) throws IOException, ServletException {
    Transaction tx = null;
    try {
      tx = sf.getCurrentSession().beginTransaction();
      chain.doFilter(req, resp);
      tx.commit();
    } catch(RuntimeException ex) {
      try {
        ex.printStackTrace();
        tx.rollback();
      } catch(RuntimeException rbEx) {
        System.out.println("Could not rollback transaction " + rbEx);
        rbEx.printStackTrace();
      }
      throw ex;
    }
  }
}
```

TX and Session now span the entire Request

Most Open Session in View filter implementations do not throw an exception.

We added this to be able to use the custom error page configured in the web.xml

Service Methods

- No longer contains Hibernate.Initialize()
- No longer contains TX code

```
public class CustomerService {
  ...

  public void addNewCustomer(Customer cust, Address shipAddr, CreditCard cc,
                             Address billAddr) {
    cc.setAddress(billAddr);
    cust.setShipAddress(shipAddr);
    cust.setCreditCard(cc);

    addressDao.create(shipAddr);
    addressDao.create(billAddr);
    ccDao.create(cc);
    customerDao.create(cust);
  }

  public Customer getCust(int custId) {
    return customerDao.get(custId);
  }
  ...
}
```

No longer contains TX code, if a controller calls more than one Service method the Transaction spans more than a single unit of works

No longer contains Hibernate.Initialize() code, view automatically loads related objects

DAO

- Same DAOs as used for Thread Local Pattern

```
public class AddressDAO {
  private SessionFactory sf = HibernateUtil.getSessionFactory();

  public void create(Address addr) {
    sf.getCurrentSession().persist(addr);
  }

  public Address get(int id) {
    return (Address) sf.getCurrentSession().get(Address.class, id);
  }

  public void update(Address addr) {
    sf.getCurrentSession().saveOrUpdate(addr);
  }

  public void delete(Address addr) {
    sf.getCurrentSession().delete(addr);
  }
}
```

Controller

- No longer catches the exception, rolls back the transaction, or redirects to the error page

```
public class ViewCustomer extends HttpServlet {
  private static final long serialVersionUID = 1L;

  public void doGet(HttpServletRequest req, HttpServletResponse resp)
                    throws ServletException, IOException {
    int custId = Integer.parseInt(req.getParameter("custId"));

    CustomerService custServ = new CustomerService();
    Customer cust = custServ.getCust(custId);
    req.setAttribute("cust", cust);

    // forward to view customer page
    req.getRequestDispatcher("customer.jsp").forward(req, resp);
  }
}
```

Transaction Rollback for all controllers now managed in single location in the Filter

Selecting the correct error page configured in a single location in the web.xml

Removing the try catch from controllers, removing the need to copy-and-paste

View

- JSP can now dynamically load related objects

```
<table>
  <caption>Shipping Address</caption>
  <tr>
    <th>Street:</th>
    <td>${cust.shipAddress.street}</td>
  </tr>
  <tr>
    <th>Apt:</th>
    <td>${cust.shipAddress.apt}</td>
  </tr>
  <tr>
    <th>City:</th>
    <td>${cust.shipAddress.city}</td>
  </tr>
  <tr>
    <th>State:</th>
    <td>${cust.shipAddress.state}</td>
  </tr>
  <tr>
    <th>Zip:</th>
    <td>${cust.shipAddress.zip}</td>
  </tr>
</table>
```

Even though the controller only loaded the customer object, the view can access **cust.shipAddress.street**

Open Session in View Issues

- The Open Session in View Pattern cleanly solves almost all our Integration issues
  - Only one Session Per Request
  - Exceptions are handled cleanly in one place
  - Transactions can span too many operations (no longer atomic)
    - We will see next module how Spring Transaction Demarcation can solve this
    - Spring also provides other features that aide development

Problems and Solutions

Problem	Solution
Multiple Session Factories	Hibernate Util
Session Per Operation Anti Pattern	Thread Local
Eagerly load (initialize) data for view	Open Session in View Pattern
Cannot have multiple transactions	Spring – next module

Active Learning

- What problems arise when we use the session per operation anti-pattern?

- Why do we have to pre-cache (initialize) data before giving it to the view when using the thread local pattern?

Module Summary

- In this module we discussed Integrating Hibernate into a more typical architecture
  - We discussed problems with Session Management, Transactions, and Exception Handling
- HibernateUtil provides a single SessionFactory
- SessionPerOperation should be avoided at all costs
- ThreadLocal can provide SessionPerRequest but with some problems for view rendering
- OpenSessionInView solves view issues by keeping the session open during view rendering