



Unit Revision and Study Guide

Semester 2B – 2018, Suzhou
Monash University

Final Unit Examination

Time: 9am-12pm, Friday 24 August 2018

Venue: Room 7211

Suzhou Joint Campus, China

Exam Structure

- 3 Hrs + 10 mins (Reading Time)
- Total Marks: 100 (worth 50% of total assessment)
- Two modules and 4 sections all together – Each section has multiple questions
- Space is provided on the exam paper for you to write
 - The amount of space is an indication of how much you are expected to write
- Please be precise and to the point

Hints

- Do sample exam (one hardcopy per student)
- Work through tutorial exercises
- Refer to Text book chapter references especially for Module 1
- Read more detailed Lecture Topics as well Lecture Slides for Module 2
- Try to answer quiz questions available for most lectures
- Study the code samples in lecture materials
- Almost ALL exam questions taken from the above sources



Module One Revision

Enterprise Applications Development for the Web

Java EE Architecture

- The Java EE platform uses a **distributed multi-tiered** architecture for enterprise applications.
- Application logic is divided into **components** according to function.
- These components are then installed on different machines depending on the tier to which they belong to.
 - **Client-tier** components run on the client machine.
 - **Web-tier** components run on the Web server/Java EE server.
 - **Business-tier** components run on the Java EE server.
 - **Enterprise information system (EIS)-tier** software runs on the EIS server.

Component-based Software

- Java EE applications consist of **components**.
- A Java EE component:
 - is a **self-contained** functional software unit that is assembled into a Java EE application
 - contains its **related** classes and files
 - can **communicate** with other components
- The Java EE specification defines the following components:
 - **Application clients** and applets are components on the client
 - Java Servlet, **JavaServer Faces** (JSF) and JavaServer Pages (JSP) technology components are web components that run on the web server/Java EE server
 - **Enterprise Java Bean** (EJB) components are business components that run on Java EE server

Java EE Clients – Web Clients

- A **Java EE client** is usually either a **web client** or an **application client**
- Web client usually consists of two parts:
 - **Dynamic web pages** containing various types of markup language (e.g. HTML, XML, and etc) that are generated by web components running in the web tier
 - A **web browser**, which renders the pages received from the server
- A web client does **not** usually access database directly, execute complex business rules, or connect to legacy applications.
- The heavyweight operations are **off-loaded** to EJBs running on the Java EE server.

Java EE Clients – Application Clients

- An **application client** runs on a **client** machine.
- It usually has a **graphical user interface** (GUI) created using Swing or Abstract Window Toolkit (AWT) API, but a command line interface is also possible.
- It allows users to handle tasks that require a **richer user interface**.
- Application client **directly** access enterprise beans running in the business tier, but it is capable of opening an HTTP connection to communicate with a servlet running in the web tier if needed

Java EE Containers

- A **runtime** environment that Java EE components run in.
- They provides the **interface** between a component and the low-level, platform-specific functionalities that support the component.
- Example of services provided by Java EE containers include:
 - EJB & servlet **lifecycles**
 - State management
 - **Multithreading**
 - Resource **pooling** and etc.
- Before a component can be executed, it must be **assembled** into a Java EE module and deployed into its container.

What is persistence?

- **Persistence** in relation to application development relates to data being stored **permanently** so it can be reused in the future.
 - Examples include storing data in files or in a database.

Persistence in Java

- Three main approaches:
 - **Serialization**
 - Java Database Connectivity (**JDBC**)
 - Object-relational mapping (**ORM**)

Serialization

- It converts an object into a **sequence** of bits.
- It stores data in a **file**
- It is **easy** to use.
- It must store and retrieve the entire object graph at once.
Hence **not** suitable for large dataset.
- It cannot undo changes that are made to object if an error occurs while information is being updated.

Java Database Connectivity (JDBC)

- It uses **SQL** to retrieve/manage data in a database
- It does not support storing objects due to the **relational** paradigm used. Hence, **mapping** between objects and relational database is necessary.
- Developer must know **both** languages in order to use this framework.
- The mapping process is very **time-consuming** and **error prone**.

Object-Relational Mapping (ORM)

- **Converts** data stored as objects into a relational database structure and vice versa
- Allows developers to focus on **object model** instead of the mapping between object-oriented and relational paradigms.
- Supports **advanced** object-oriented concepts such as inheritance.
- It is not limited to Java.
- Many products available (e.g. Hibernate, ActiveJDBC, MyBaits and etc.)

Java Persistence API (JPA)

- Provided as part of EJB 3.0 specification.
- A specification that provides **standardized** interface for accessing, persisting and managing data.
- It does not have any implementation itself.
- JPA providers (e.g. Hibernate, EclipseLink) develop their own version of **JPA implementation** that meets the requirements of the JPA specification.
- JPA can be **used** in EJBs, web components and Java SE application.

Core Components of JPA

- Object-Relational Mapping (**ORM**)
- Entity **Manager** API
- Java Persistence Query Language (**JPQL**)
- Java **Transaction** API
- **Callbacks & listeners**

Entity Manager

- The center piece of the API **responsible** for persisting, updating, retrieving and deleting entities to/from database.
- Serves as a **bridge** between an **OO program** and the **relational world**.
- It is only an interface and the implementation is provided by **persistence provider** (e.g. EclipseLink).

Obtaining an Entity Manager

The way to obtain an Entity Manager depends on the environment it is being used.

- Application-managed environment (e.g. J2SE application)
 - Via **EntityManagerFactory**
- Container-managed environment (e.g. Application Client, EJB, Web components)
 - Via **Resource injection**

What is Java Server Faces (JSF)?

- JSF is a **server side** component framework or architecture for generating dynamic and interactive web pages.
- **It consists of:** Page Description Languages (**PDLs**, such as JavaServer Pages and Facelets);
- Backbeans or **Managed Beans** to implement business logic;
- Many other supporting elements.
- **Generated** web page content is sent to the **client / browser** for displaying and providing an interface to send data back.

Facelets

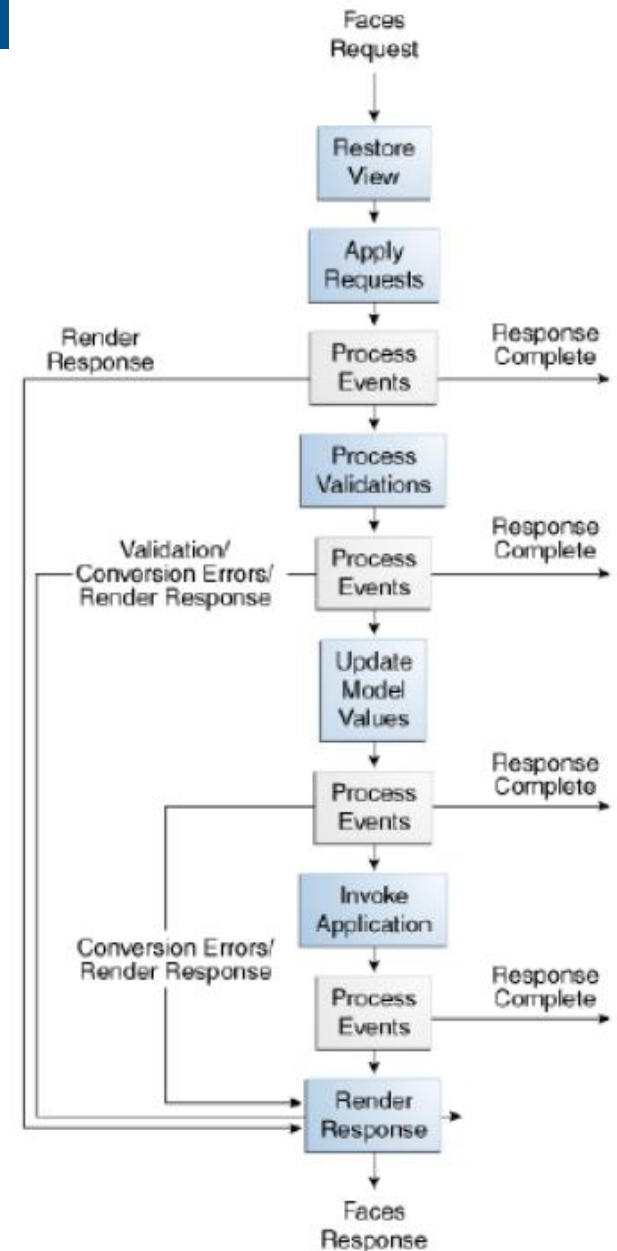
- **Web template system and the default view handler for JavaServer Faces.** Uses XML documents (via XHTML) to declare the components within a view.
- Much easier to define rich web interfaces than the previous JSP approach (depreciated as view handler from JSF 2.0).
- **Facelets has support for all JSF UI components and helps structure the JSF component tree (the view).** Also provides support for templating and use of Expression Language.

Managed Beans

- A **managed bean** is a regular bean registered with JSF. A Java Bean is a convention for a class where all properties are *private*, constructor has *no arguments* and implements *Serializable*.
- Works as the *model* for the JSF user interface components.
- Managed beans are **accessible** from JSF pages.
- JSF **automatically** looks after Managed Beans: Insanitisation (constructors must have no arguments)
- **Lifecycle** (Scoping)
- **Calling** accessor / mutator methods (via Expression Language)

JavaServer Faces Lifecycle

- Two types of requests are handled:
 - *Initial Request*: First request to a page
 - *Postback Request*: Submitting form back to previously rendered view
- Lifecycle comprised of six key phases:
 - Restore View
 - Apply Request Values
 - Process Validations
 - Update Model Values
 - Invoke Application
 - Render Response



Client and Server Validation

- Ideally, you should provide the client (web browser) a method in which it can validate content **before** it is sent to the server
 - Helps the user correct quick **mistakes** or **missing** fields
 - Prevents **many** requests going to the server
 - Using **AJAX** allows server validation messages to be shown to the user **dynamically**
- Most importantly, you should always enforce **server validation** as you **cannot trust** that **clients** will always validate inputs
 - Providing validation for both gives us the best of both approaches

Validation Options in JSF

- We can take a few different approaches in validating data received in JSF
- **Built-in** Validation Components (via JSF Core Library)
 - Example: `<f:validateDoubleRange>` and `<f:validateLength>`
 - See tutorial tasks
- **Managed Bean Validation** Methods / Validator Interface
- **Bean Validation (Java EE 6+)**

Criteria API

- Allows developer to write queries using an **object-oriented** approach.
- Queries written using Criteria API are **type-safe**.
- Most of the errors are discovered at **compile** time.
- Support **everything** that JPQL can do.

Container Managed Entity Manager

- **Container** is responsible for creating and closing the entity manager.
- **Transaction** management is handled by container.
- It is obtained by using `@PersistenceContext`

What are Enterprise JavaBeans (EJBs)?

- Java EE components that implement the **business logic** for enterprise applications.
 - Run in an **EJB container** which operates in a runtime environment such as the Glassfish server.
- Similar to **ManagedBeans**, EJBs have their own specific annotations to designate what kind of functionality should be implemented.
- **Designed to access databases, be called/invoked by local and remote clients or components**

Benefits of EJBs

- **Simplify** development of large-scale enterprise applications by separating **Business-tier** from **Web-tier**.
 - **Client developers** can focus on the web **frontend**.
 - **Application developers** can focus on the web **backend**.
- EJB container has access to **system-level services** such as transactions and security authentication.
 - Enables developers to focus on the **core business** application objectives.
- EJBs are also **portable components** which can be reused in other Java EE applications provided the standard Java libraries have been used.

Using Enterprise JavaBeans

- We can access EJBs using a *no-interface view* or through a defined *business interface*.
 - *No-interface views*: Expose the *public methods* of an EJB implementation to clients.
 - *Business Interface*: A standard *Java Interface* which outlines the business methods of the EJB.
- Aside from the public method implementations, all other EJB method implementations or settings are *hidden* from the client.
- We can obtain a reference to an EJB via *dependency injection* (more in-depth in next lecture) or *JNDI* (Java Naming Directory Interface).

Accessing EJBs via Dependency Injection

- Working with EJBs is best done using **dependency injection** since it is very easy to implement.
- To obtain a reference to the ***no-interface view*** or **local business interface** of an EJB, we use dependency injection via the *javax.ejb.EJB* annotation and specify the EJB implementation class.

@EJB

CalculatorBean calculator;

- Interfaces need to have a **unique name** separate from the Bean.
 - E.g. Calculator, CalculatorLocal, CalculatorRemote.

Accessing EJBs via Dependency Injection

Accessing EJBs via JNDI (1)

Three key namespaces used for lookups:

java:global

- java:global[/app]/module/ejbName[/interfaceName]
- **Portable** way of finding remote EJBs (and when annotations aren't usable such as older Java EE platforms).

java:module

- java:module/ejbName[/interfaceName]
- Used to search for EJB implementations within the **same EJB module**.

java:app

- java:app[/module]/ejbName[/interfaceName]
- Used to search for EJB implementations within the **same application**.

Accessing EJBs via JNDI (2)

- Example: *CalculatorBean* inside *Calculator.war*
 - **Portable JNDI Name:**

`java:module/CalculatorBean`

OR

`java:global/calculator/CalculatorBean`

- **Usage:**

```
Context context = new InitialContext();  
CalculatorBean calc = (CalculatorBean)  
context.lookup("CalculatorBean/sum");
```

- **We will be focusing on the *Injection approach* for this unit.**

Contents of an EJB

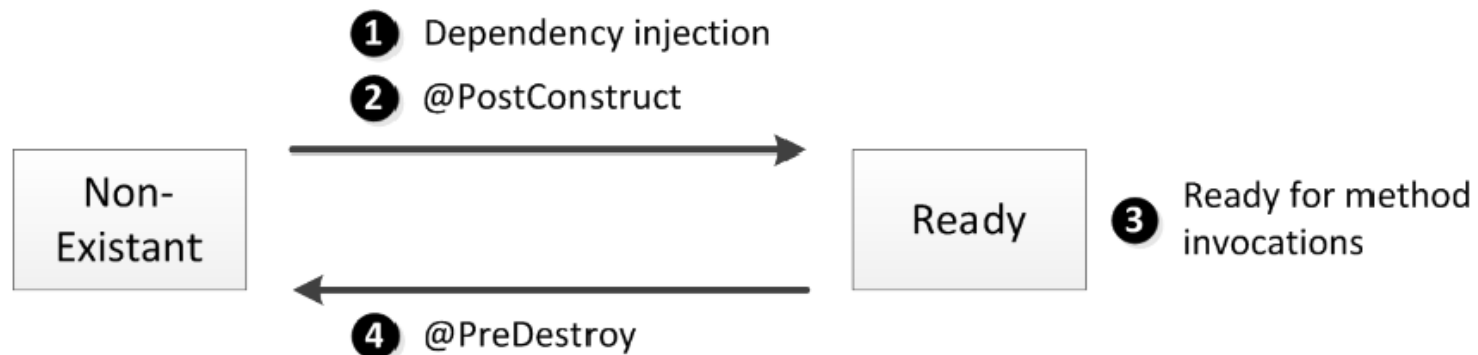
- **EJBs are comprised of the following files:**
 - **Enterprise Bean Class:** Implements the **business methods** of the enterprise bean and any lifecycle callback methods.
 - **Business Interfaces:** Define the **business methods** implemented by the enterprise bean class. Interface is not required if the enterprise bean exposes a local, no-interface view.
 - **Helper Classes:** Other classes needed by the enterprise bean class, such as exception and utility classes.

Session Beans Life Cycle

- Clients do not need to manually create an instance of an EJB since we are getting a reference using **dependency injection (DI)** or through a JNDI lookup.
 - The EJB container has the responsibility of monitoring the life cycle of each Session Bean.
- Very similar to the **life cycle** of our Managed Beans.
- While **Stateless** and **Singleton** Session Beans have the same life cycle, **Stateful** Session Beans require a slightly different cycle due to its potential persistent state.

Life Cycle: Stateless and Singleton

- **Don't maintain** conversational state with a client.
- Stateless is accessed on per-instance basis.
- Singleton provides **concurrent** access to a single instance.

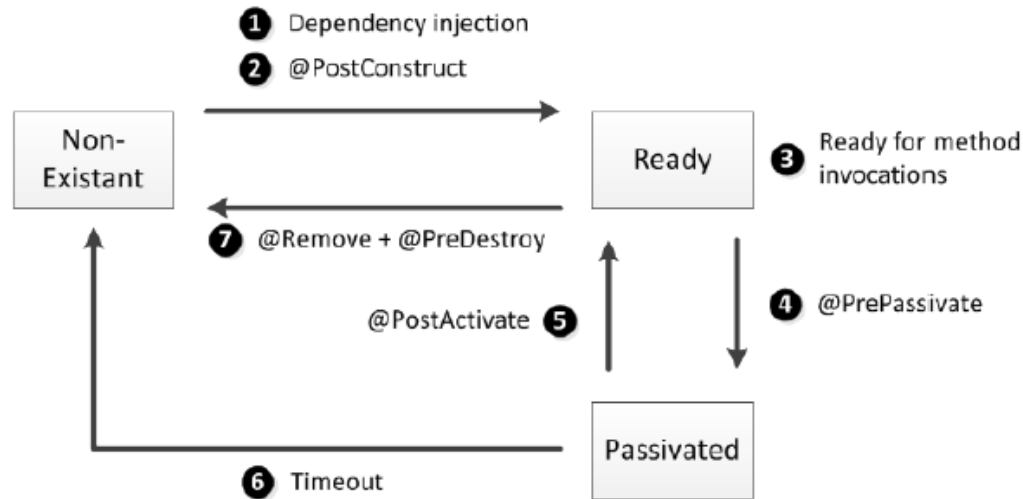


- **Created via Dependency Injection (E.g. @EJB, @Inject)**
@PostConstruct methods are called upon creation.
- **Bean has been created, ready for method invocations.**
@PreDestroy methods called before instance removal.

Life Cycle: Stateful (1)

- Maintains a **conversational state** with the client.
 - The EJB container will generate an instance and assign it to **one client**.
 - Each following request from that client is then sent to that instance.
- Following this approach allows us to gain a **one-to-one** relationship between a client and Stateful Session Bean.
 - Example. 100 users = 100 Stateful Session Beans Ideal for scenarios like a Shopping Cart
- EJB container uses **Passivation** which serialises the bean instance to a storage medium / database when not required. **Activation** is used to bring back an instance into memory.

Life Cycle: Stateful (2)



1. Created via **Dependency Injection** (E.g. @EJB, @Inject)
2. **@PostConstruct** methods are called upon creation.
3. Bean has been created, ready for method invocations.
4. Bean not in use, **@PrePassivate** methods called before serialisation
5. **@PostActivate** methods called for re-initialisation.
6. Timeout has occurred for the bean, remove it entirely.
7. **@Remove** and **@PreDestroy** methods called before removal.

Life-Cycle Callback Annotations

Annotation	Description
@PostConstruct	Marks a method to be invoked immediately after you create a bean instance and the container does dependency injection. This annotation is often used to perform any initialization.
@PreDestroy	Marks a method to be invoked immediately before the container destroys the bean instance. The method annotated with @PreDestroy is often used to release resources that had been previously initialized. With stateful beans, this happens after timeout or when a method annotated with @Remove has been completed.
@PrePassivate	Stateful beans only. Marks a method to be invoked before the container passivates the instance. It usually gives the bean the time to prepare for serialization and to release resources that cannot be serialized (e.g., it closes connections to a database, a message broker, a network socket, etc.).
@PostActivate	Stateful beans only. Marks a method to be invoked immediately after the container reactivates the instance. Gives the bean a chance to reinitialize resources that had been closed during passivation.

RESTful Web Services

- Common approach to web services by using “**stateless**” methods of interaction, commonly via URLs.
- Uses HTTP methods such as **GET** and **POST** to receive and send information to a web service.
- Not really bound by a strict standard that SOAP requires.
 - Very **easy to setup** as a result since you can make your own REST implementation.
 - **Documenting** the API required for almost any “Web 2.0” application since they all take different approaches.

Java Authentication and Authorization Service (JAAS)

- JAAS provides a **set of APIs** that enable various services within the Java EE platform to support **authentication** and enforce **access control** lists (ACL) upon user roles.
- In particular, we will be using this service to integrate **user authentication** into our web applications.

JAAS: Supported Authentication Methods

- The Java EE platform supports the following authentication mechanisms:
 - Basic authentication
 - Form-based authentication
 - Digest authentication
 - Client authentication
 - Mutual authentication
- We will focus on **Basic** and **Form-based** authentication methods for this unit.

Password Hashing

- To store passwords in a more secure manner, we can use **one-way hashing algorithms** to obtain a completely different encrypted value.

Example using MD5:

• 'password' => '5f4dcc3b5aa765d61d8327deb882cf99'

- By storing passwords in this manner, we can **hash user password inputs** and compare them with the stored value in the database.
 - We cannot obtain the value of the password otherwise unless we attempt to **brute-force the password** by generating hashes for each potential password combination

Rainbow Table Attacks

- Unfortunately storing passwords using the previous hash approach is **still not very secure!**
- Majority of hash algorithms are designed for **high performance**.
 - With the computational hardware available today common passwords are **easy to be computed** via **brute-force method**.
- **Rainbow tables** are precomputed key-value pair tables which enable the reverse lookup of hash algorithms against common phrases.
 - These are easily available online and for purchase.

Password Salting

- We can **strengthen stored passwords** by adding random data that is associated with the hashed password.
 - **Salted value** can be **simply added** with the password (password + passwordSalt).
 - **Random data** must be generated for each user for optimum security.
- **Example using MD5:**

```
Password = 'password';
```

```
passwordSalt = 'RN5_43<m84@|]5&66[33';
```


```
password = password + passwordSalt;
```

```
Hashed password
```

```
'password' => '4a935057ff675ffc83a4e46ca4f873dd'
```

Password Salting: Benefits

- By each user having their **own salt** attached with the password they input, we prevent the effectiveness of rainbow table attacks.
 - **Long random data segments** will significantly increase computational requirements to brute-force hashed passwords.
- If your **database is compromised**, the attacker would need to **brute-force** each individual user password with their assigned salt to obtain the true value.
 - Other algorithms can be used which are designed to **take time** to compute (E.g. bcrypt, scrypt) to also prevent the effectiveness of this approach.



Module Two Revision

Internet Applications Development Using ASP.NET

Module Two – Material not on Exam

- Background topics from Day 0
- Topic 5 ASP.NET – C#
- Topic 6.2 ASP.NET - Themes & Skins
- ASP.NET Calendar control
- Detailed database access logic and web service file system handling (just understand basic controls)
- Advanced Data Control features
- Extraordinary namespace knowledge

ASP.NET Server Controls

- **Standard:** Basic controls used on many pages, such as buttons, links, images, lists etc
- **Validation:** Used to validate data input by a user.
- **Navigation:** Used to provide navigation features, such as menus, to allow users to navigate around a web site.
- **Data:** Provide access to data sources such as databases or XML files
- **Login:** Used to provide a login mechanism for users to gain access to a site.

ASP.NET Validation Controls

- **RequiredFieldValidator:** Checks that field has data in it.
- **CompareValidator:** Compares a user entry against a constant value.
- **RangeValidator:** Checks that the user's entry is between specified lower and upper boundaries. Checks ranges within pairs of numbers, alphabetic characters and dates. Boundaries can be constants or values derived from another control.
- **RegularExpressionValidator:** Checks that an entry matches a pattern defined by a regular expression. Allows to check for predictable sequences of characters: phone numbers, email addresses, post codes, etc.
- **CustomValidator:** Checks the user's data entry using validation logic from a custom method written by the developer.

Examine source codes on the following

- Login Control
- Validation Controls
- Navigation Controls
- Data Controls (AccessDataSource)
- GridView Control and its various column types
- Master Page and Content Page formats
- Basic Emailing and File System user interactions
- Other examples from Lectures 4 - 11 (refer to detailed Lecture Topics from each day)

ASP.NET Web Form – Display web page

Steps which need to happen in order for an ASP.NET page to be displayed via a browser.

1. The author writes a set of instructions in a file, saves the file with an .aspx extension and transfers the file to a Virtual Directory on a Web Server.
2. Client requests the web page
3. Web server passes control to another ASP.NET engine
4. HTML is created from instructions and passed back to the Web Server
5. HTML output stream returned to the browser
6. The browser interprets the HTML and displays web page

ASP.NET MVC Web Application

- Role of Models, Views and Controllers in an MVC Application

ASP.NET MVC Web Application

Role of **Models** in an MVC Application:

- Generally a model is a class or set of classes that describes all the business logic and additionally handles data access for an application.
- The model also contains code that defines its relationship with other models and defines the data validation rules to be used when adding or updating data.

ASP.NET MVC Web Application

Role of **Views** in an MVC Application:

- Views are the outputs or responses that are sent back to the user once a request is processed.
- They basically consist of markup (like HTML) code with embedded .NET code, but they can also be other forms of output like XML, PDF documents etc depending on the situation.
- Views can be thought of as the presentation layer of an application and ideally should be as "dumb" as possible.

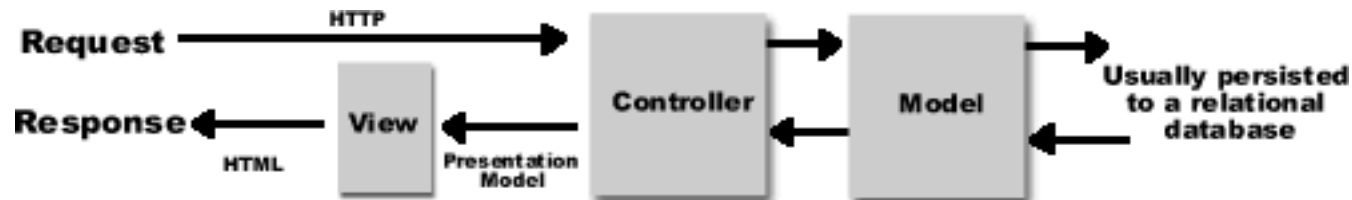
ASP.NET MVC Web Application

Role of **Controllers** in an MVC Application:

- Controllers control the application flow or logic of the application. Each web request is directed to a particular controller where the user input is accepted. The controller logic then decides what response is to be generated.
- The controller logic normally contains calls to models to access data, and also other functionalities like access control checks etc. Lastly, the controller passes the response (output) to the view.
- Controllers can be thought of as the control logic layer of the application. As mentioned above, the model should have all the business logic of an application. The controllers are used just to delegate the actions to the models and they should be "light". This design philosophy is sometimes referred to as "fat models and thin controllers".

Request Handling in the MVC Scenario

- In the MVC scenario, we have
 - SQL queries in the Model
 - HTML and other graphic elements in the View
 - logic in the Controller
- The Controller calls and fetches data from the Model. It then loads the data and passes it to the Views, which sends the results to the user. The following diagram shows how all these components work together:



- The request is sent to the controller, with the user entered/selected data (POST/GET data).
- The controller processes the request, and calls the model to access the data.
- The model responds to the controllers call, by sending or storing data.
- The controller then sends the output data to the view.
- The view outputs the data in the proper format.



Good luck and all the very best 😊