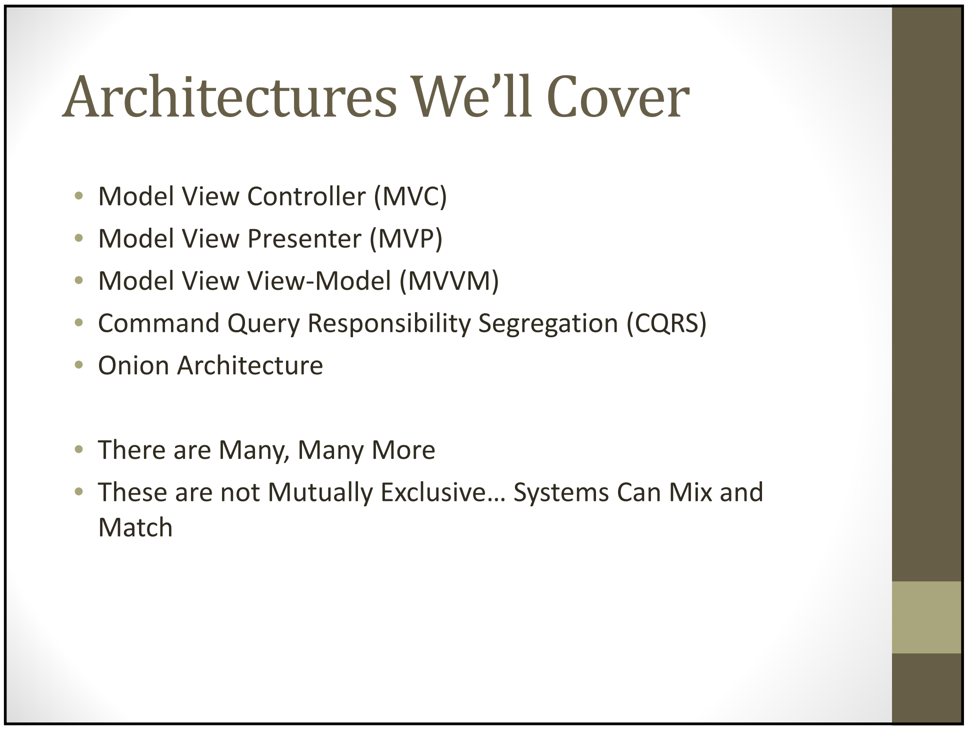
Software Engineering

CSC440/640
Prof. Schweitzer
Week 10

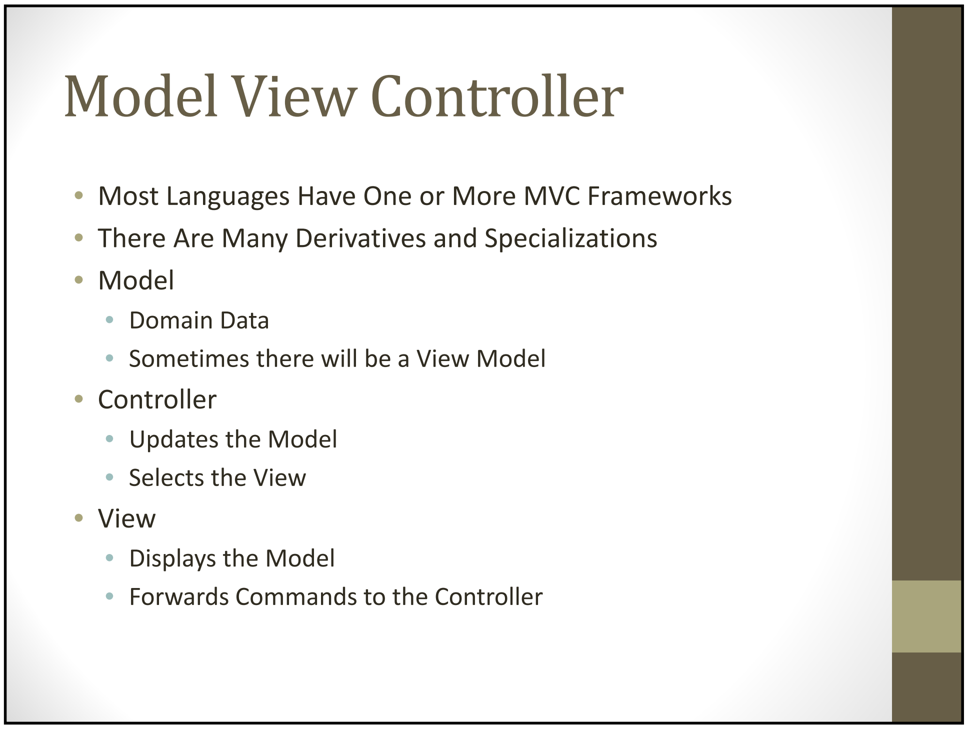
Architectural Jam Session



Architectures We'll Cover

- Model View Controller (MVC)
- Model View Presenter (MVP)
- Model View View-Model (MVVM)
- Command Query Responsibility Segregation (CQRS)
- Onion Architecture

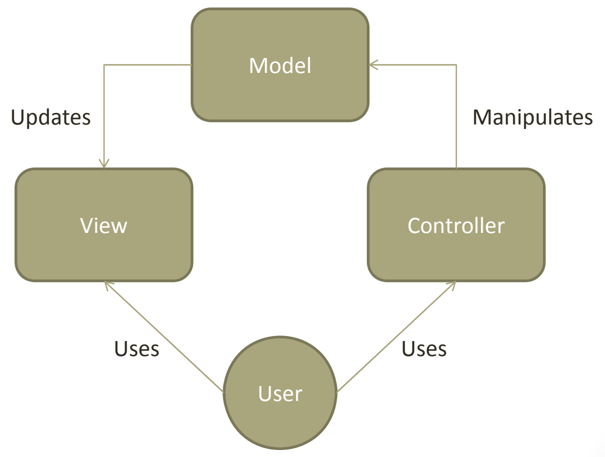
- There are Many, Many More
- These are not Mutually Exclusive… Systems Can Mix and Match



Model View Controller

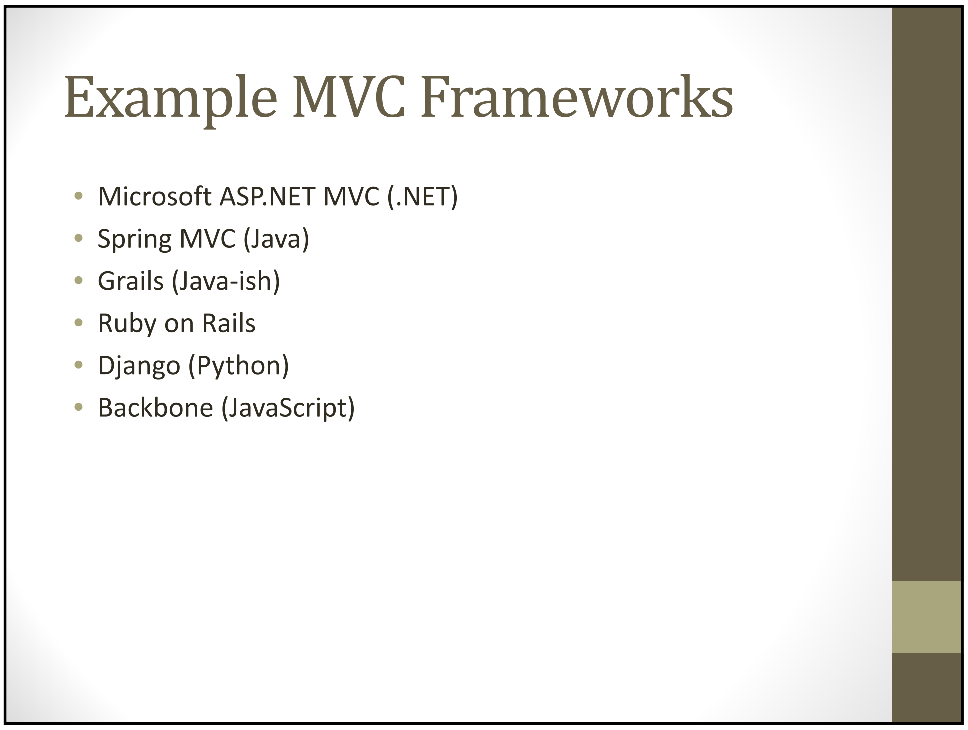
- Most Languages Have One or More MVC Frameworks
- There Are Many Derivatives and Specializations
- Model
  - Domain Data
  - Sometimes there will be a View Model
- Controller
  - Updates the Model
  - Selects the View
- View
  - Displays the Model
  - Forwards Commands to the Controller

MVC Diagram



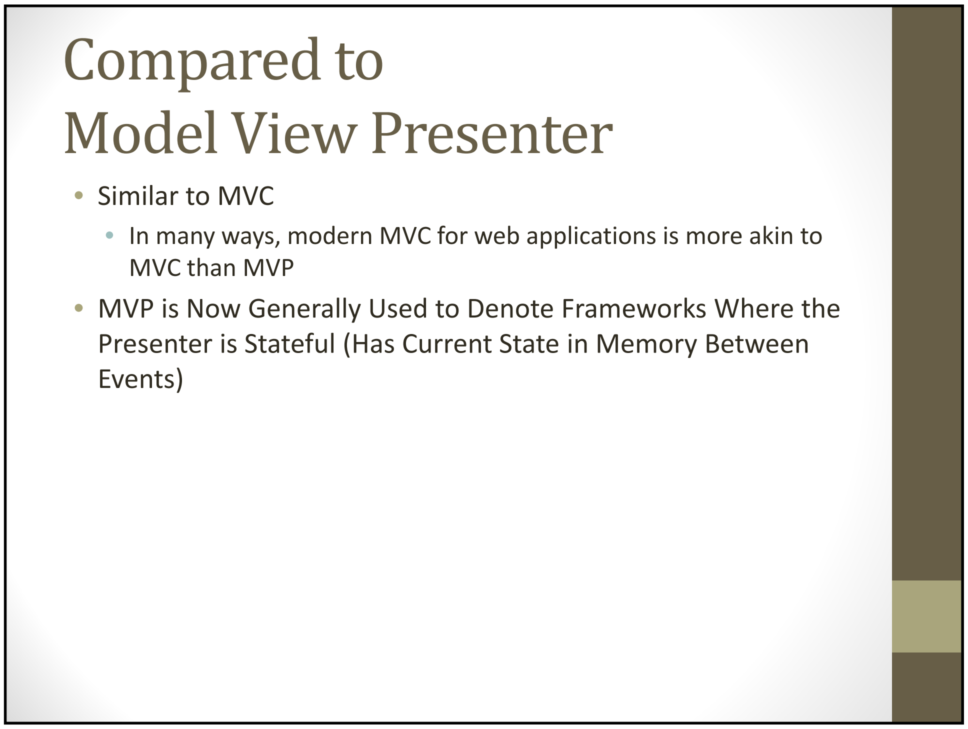
MVC and the Web

- Ideal for Stateless Systems
  - No State in Memory Kept Been Subsequent Operations
- Extremely Popular in Web Development
- Typically a URL (Route) will Map to a Function on a Controller
  - http://www.domain.com/Controller/Function/Argument
  - MVC Frameworks Primary Purpose is to Bind Routes to Controllers
  - Works well with REST API Methodologies
- Controller can load session and query for model objects and show appropriate view
- Views Can Be Many Things
  - HTML (Web Page)
  - JSON (REST)
  - XML (SOAP)



Example MVC Frameworks

- Microsoft ASP.NET MVC (.NET)
- Spring MVC (Java)
- Grails (Java-ish)
- Ruby on Rails
- Django (Python)
- Backbone (JavaScript)



Compared to Model View Presenter

- Similar to MVC
  - In many ways, modern MVC for web applications is more akin to MVC than MVP
- MVP is Now Generally Used to Denote Frameworks Where the Presenter is Stateful (Has Current State in Memory Between Events)

Web Service Technologies

- Web Services are HTTP Remote Procedure Calls – Calling a Function on the Web
- SOAP
  - Simple Object Access Protocol
  - XML Format for Making Remote Procedure Calls
  - XML Structure Handles Everything Including Function Names, Parameters, Types, and Security Objects
- REST
  - Representational State Transfer
  - Data can be formatted using Simple XML or JSON
  - Uses Standard HTTP VERBS as a base
  - Uses Standard HTTP Security

Model View View-Model

- Derived from MVC
- Introduced by Martin Fowler
- Targets UI Frameworks Which Support Data Binding
- Model
  - Purely consists of the domain model
- View-Model
  - A model which represents the fields being show in the view
  - Also contains Business Logic
- View
  - UI Elements Which Display the View-Model. Ideally this is expressed purely in Markup (i.e. XAML or HTML)
- Binder
  - Framework Which Updates the View and View-Model when events occur

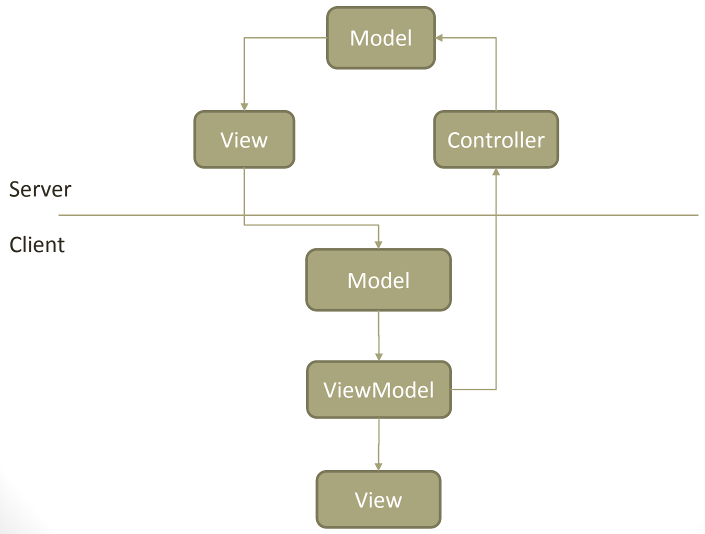
MVVM Frameworks

- Ideal for Stateful Systems
- Example Frameworks:
  - AngularJS (JavaScript)
    - Unfortunately, they claim to implement MVC
  - KnockoutJS (JavaScript)
  - Aurelia (JavaScript)
  - WPF* (.NET)
  - Silverlight* (.NET – Deprecated)
- Not Well Supported in Java
  - Lots of Small Frameworks, But Nothing Widely Used
- * XAML Doesn't Force MVVM, but the Data Binding functionality is what allows it to work.

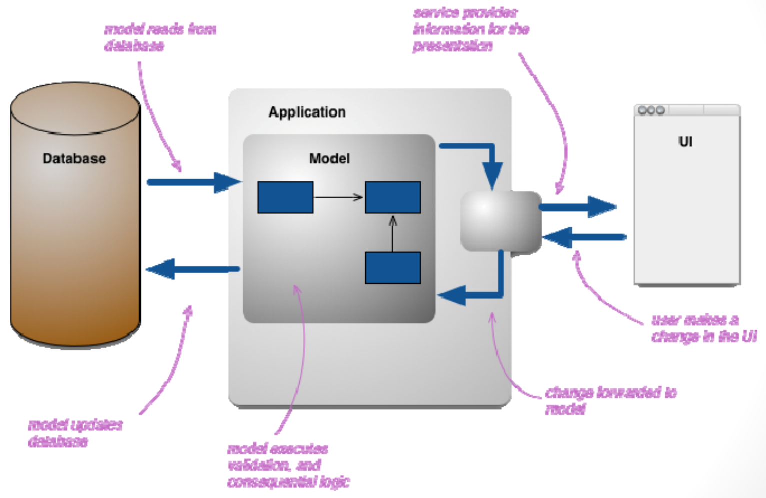
MVVM and the Web

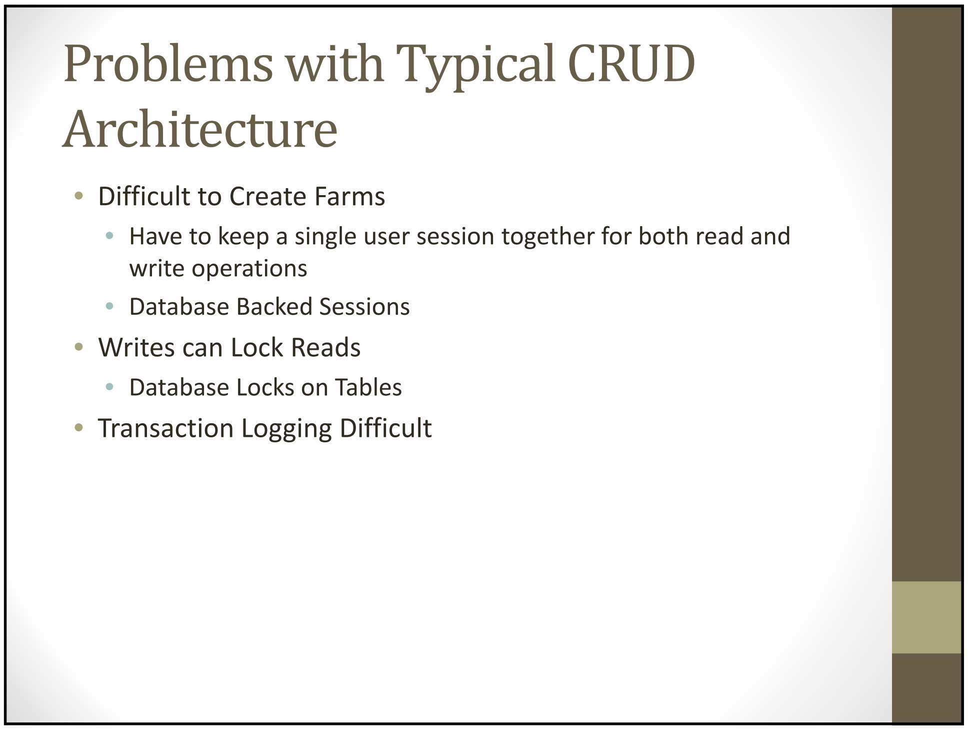
- Many Web Based Applications Now Combine MVVM and MVC
- Model-View-ViewModel is used on the Client Side
  - JavaScript application
  - Model treats Web Services as a Database
  - The JavaScript Application is Stateful
- Model-View-Control is used Server Side
  - The View is the JSON output that eventually is consumed by the Model of the JavaScript Application
  - The Web Service is Stateless

MVVM and the Web (2)



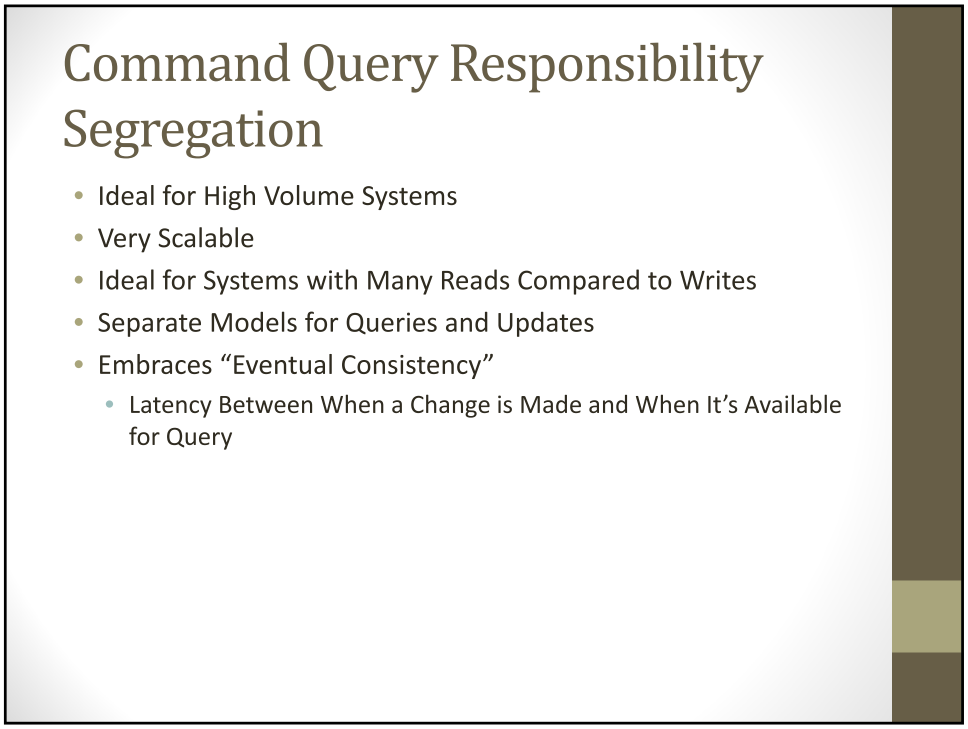
Typical CRUD Architecture


model reads from database
service provides information for the presentation
Application
Database
Model
UI
user makes a change in the UI
change forwarded to model
model updates database
model executes validation, and consequential logic



Problems with Typical CRUD Architecture

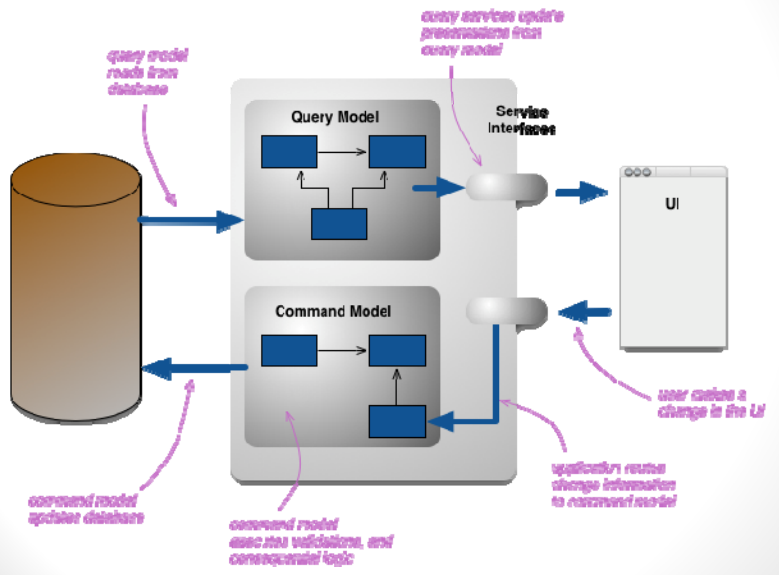
- Difficult to Create Farms
  - Have to keep a single user session together for both read and write operations
  - Database Backed Sessions
- Writes can Lock Reads
  - Database Locks on Tables
- Transaction Logging Difficult



Command Query Responsibility Segregation

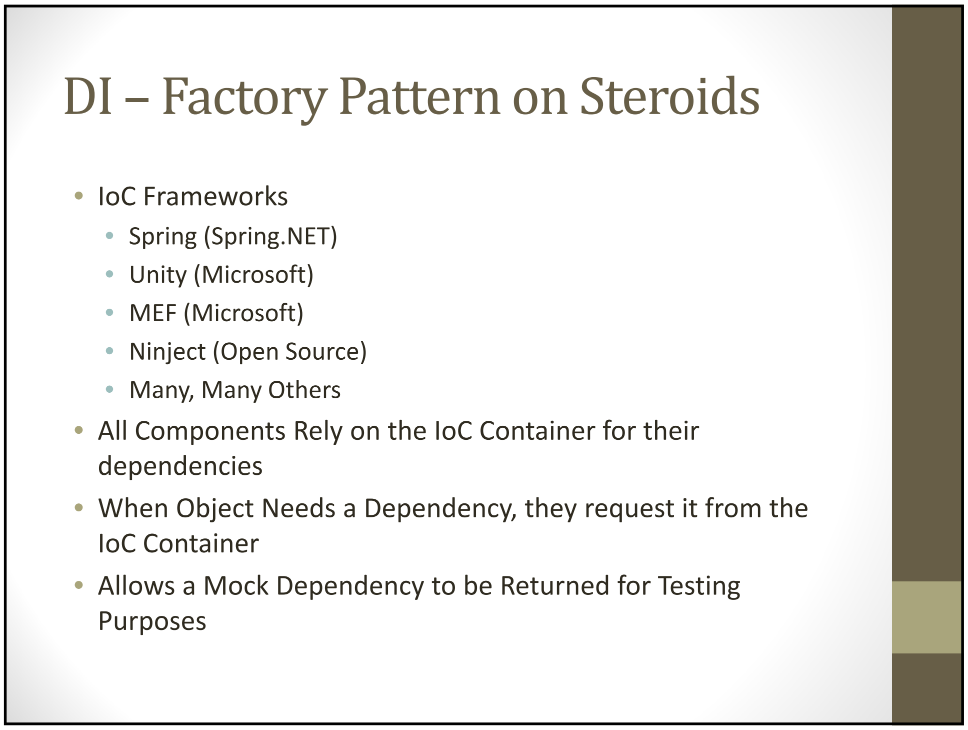
- Ideal for High Volume Systems
- Very Scalable
- Ideal for Systems with Many Reads Compared to Writes
- Separate Models for Queries and Updates
- Embraces “Eventual Consistency”
  - Latency Between When a Change is Made and When It’s Available for Query

CQRS Architecture


query model reads from database
query services update presentations from query model
Query Model
Service Interfaces
UI
Command Model
user makes a change in the UI
application routes change information to command model
command model updates database
command model executes validations, and consequential logic

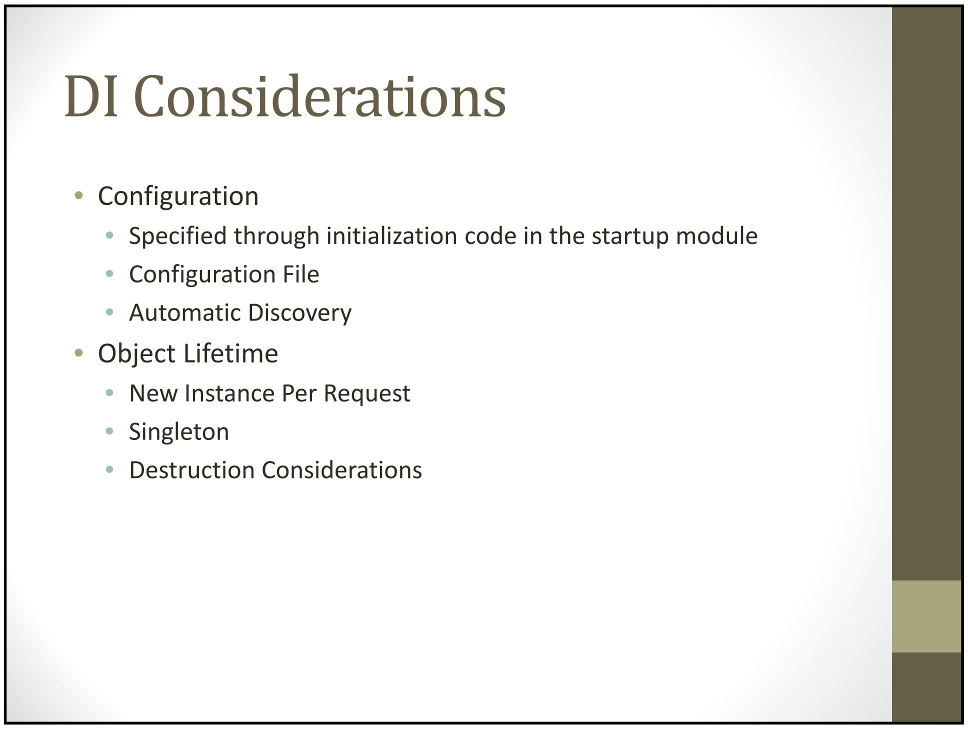
Dependency Injection

- Goal: Remove hard coded dependencies from a project
- Typical Scenario:
  - UI Specifies Business Layer Objects
  - Business Layer Specifies Database Objects
- Problem:
  - What is Testing Business Layer – How specify a different Database for testing purposes?
- Solution:
  - Inversion of Control/Dependency Injection
- Wiring of Consumers and Services is Removed from the Consumer



DI – Factory Pattern on Steroids

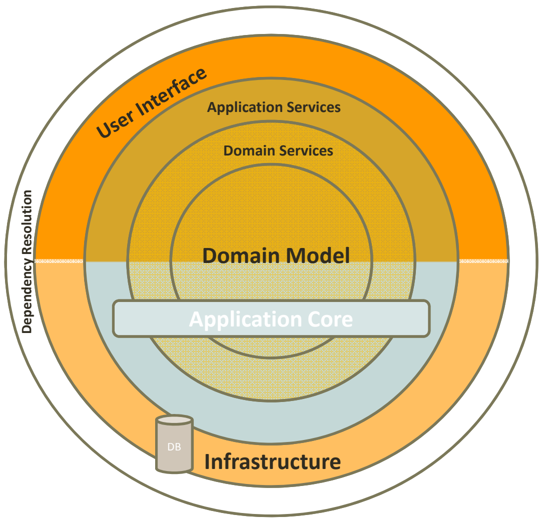
- IoC Frameworks
  - Spring (Spring.NET)
  - Unity (Microsoft)
  - MEF (Microsoft)
  - Ninject (Open Source)
  - Many, Many Others
- All Components Rely on the IoC Container for their dependencies
- When Object Needs a Dependency, they request it from the IoC Container
- Allows a Mock Dependency to be Returned for Testing Purposes



DI Considerations

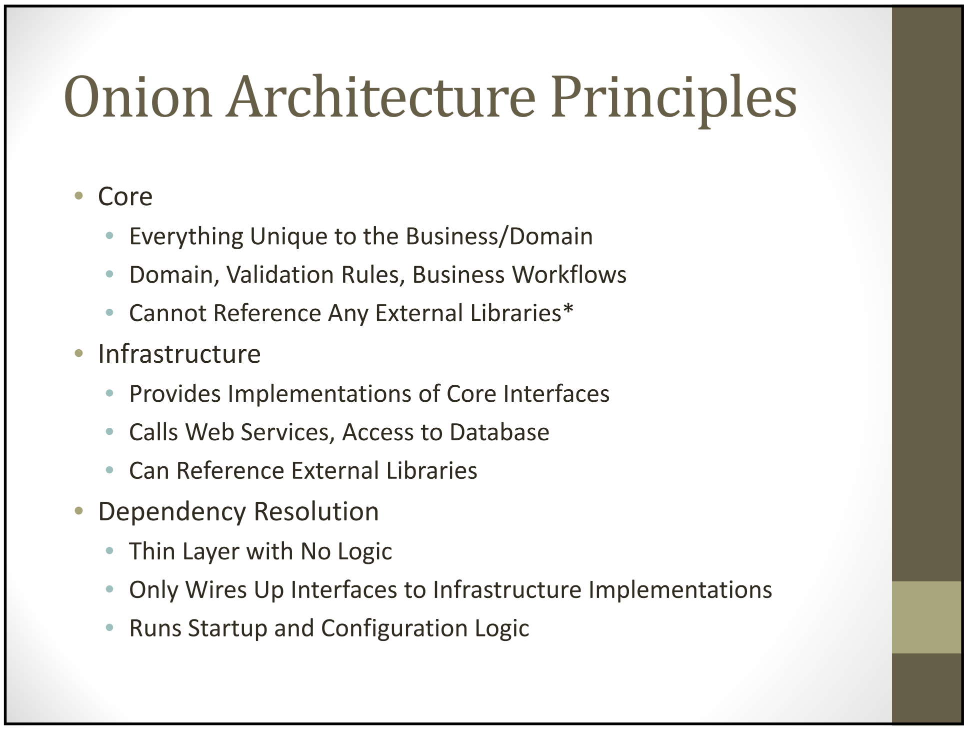
- Configuration
  - Specified through initialization code in the startup module
  - Configuration File
  - Automatic Discovery
- Object Lifetime
  - New Instance Per Request
  - Singleton
  - Destruction Considerations

Onion Architecture



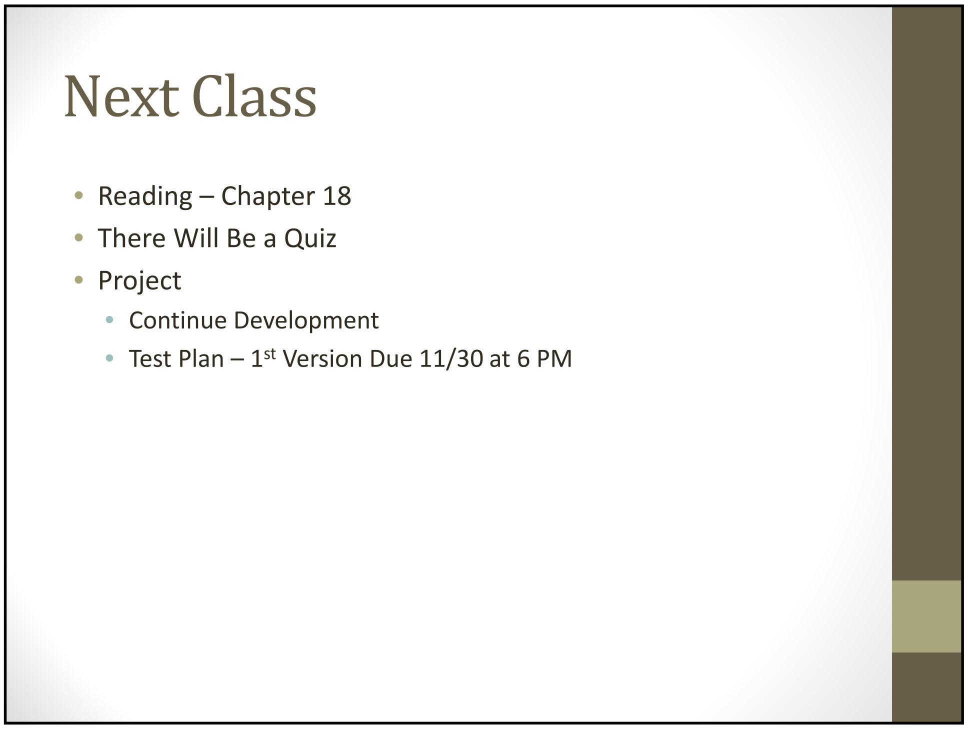
Onion Architecture Basics

- Relies heavily on the Dependency Injection Principle
- The Core of the Onion contains
  - Domain Models
  - Interfaces That Represent Business Services
- Everything Else is Considered Outside the Core and Is Injected
  - Database Repositories
  - External Services (3rd Party Components)
  - Application Services (Logging, Business Logic, etc.)
  - …
- Code that Exists Outside the Core Can Directly Reference Anything in the Layers Below
- The Core Layers Have to Have Dependencies Injected to Gain Access to Them



Onion Architecture Principles

- Core
  - Everything Unique to the Business/Domain
  - Domain, Validation Rules, Business Workflows
  - Cannot Reference Any External Libraries*
- Infrastructure
  - Provides Implementations of Core Interfaces
  - Calls Web Services, Access to Database
  - Can Reference External Libraries
- Dependency Resolution
  - Thin Layer with No Logic
  - Only Wires Up Interfaces to Infrastructure Implementations
  - Runs Startup and Configuration Logic



Next Class

- Reading – Chapter 18
- There Will Be a Quiz
- Project
  - Continue Development
  - Test Plan – 1st Version Due 11/30 at 6 PM