WEB524

WEB PROGRAMMING ON WINDOWS

WEEK 2 – LECTURE 1

PERSISTING DATA

Textbook

From now on, you will learn about the week's topics in these class notes. You should continue to use the textbook as a supplemental source. Use the table-of-contents and the index to locate the topic of interest.

The textbook has code examples that directly use persistent storage. This means the controllers interact with the data context object directly. In this course we will not do that! Instead, we will use *view model* classes. Unfortunately, the textbook does not work with the *view model* class topic which is critical concept that you will learn and use in this course.

Resources

- Please navigate and study any links contained in this presentation.
- For information on the Web App Project Template please read the document accompanying these notes on Blackboard.
- Code Example: Get All Get One

Previous Week

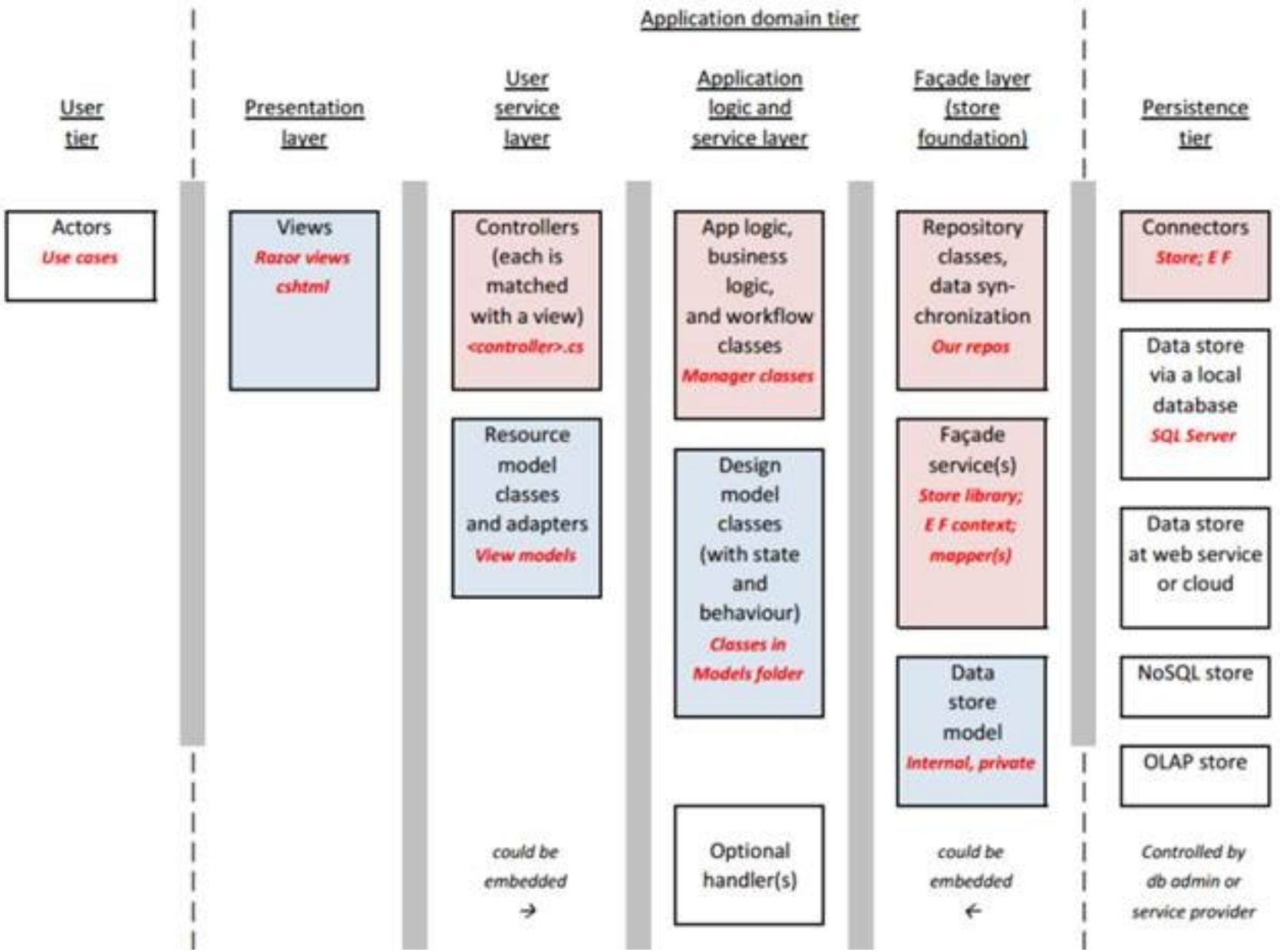
- The key objective for this course is to *create interactive web apps that are scalable yet somewhat complex*.
- During the first week of the course, we were introduced to:
  - The Microsoft web development stack, ASP.NET, C#, and Visual Studio.
  - The model-view-controller (MVC) design pattern.
  - Round-trip data to-and-from a browser.
  - Collections.
  - String <-> number conversion.

What's Next?

- Your professor will give you a *project template* that includes a pre-configured, ready-to-use database – also known as a *persistent store* – and all the program objects that enable us to interact with it.
- Over the next few weeks, you will use the template to focus on learning how to create interactive web apps without worrying about data storage.
- Later in the course, you will learn to create your own data store. You will be able to support new projects and new data entities.

System Design Guidance

- In semester 4, we began to use an *architectural model* to guide us as we create larger and more complex apps.
- This model presents a “layered” architecture.
- Try to locate your source code modules within the architectural model.


Application domain tier
User tier
Presentation layer
User service layer
Application logic and service layer
Façade layer (store foundation)
Persistence tier
Actors
Use cases
Views
Razor views
cshtml
Controllers (each is matched with a view)
<controller>.cs
Resource model classes and adapters
View models
App logic, business logic, and workflow classes
Manager classes
Design model classes (with state and behaviour)
Classes in Models folder
Optional handler(s)
Repository classes, data syn-chronization
Our repos
Façade service(s)
Store library; E F context; mapper(s)
Data store model
Internal, private
Connectors
Store; E F
Data store via a local database
SQL Server
Data store at web service or cloud
NoSQL store
OLAP store
could be embedded
could be embedded
Controlled by db admin or service provider

System Design Guidance

- Actors interact with your web app's views which are defined by cshtml source code files in your projects Views folder.
- Views get data from (or deliver data to) controllers.
- A *view model* class describes the shape of the data object(s) coming in to or leaving the controllers. *(more on this in a moment)*
- In this lesson, we will be introduced to many of the pink and blue boxes:
  - Data store, as a locally-stored database (within our app).
  - Façade service, provided by the Entity Framework.
  - *Design model* classes, which describe the data in the persistent store.
  - Manager class, for app and business logic.

Persistent Data Storage

- Remember from last week, Visual Studio includes an on-demand web server to host our web apps.
  - It does a great job to serve the needs of software developers but couldn't hold up in a production environment.
- Visual Studio also includes an on-demand *instance*-based SQL Server database engine.
  - When you run a web app that uses a data store, Visual Studio starts the web server and the database server engine.

Persistent Data Storage

- Our web apps will use a relational database to persist data. The database will be located in the *App_Data* folder.
- The *Web.config* settings file includes a *connection string* that is used to connect to and access the database.
- When you get to the point that you want to deploy your web app on a publicly-available host, you can configure a *host*-based database server engine as your data store.
- We will learn more about the host-based SQL Server later on.

Microsoft Entity Framework

- In our web app, we will never interact directly with the database.
- Instead, we use a *façade service* called the Entity Framework (EF).
- It is an object-relational mapper (aka ORM).
- It enables C# programmers to work with objects *in memory* and persist them to a relational database management system.
- Most ASP.NET MVC web apps will use the Entity Framework.

Microsoft Entity Framework

- A combination of two components:

1. *"Data context"*: a class that acts as a gateway to the data store.
2. One or more "*design model classes*" that describe the entities in the data store.

- Place the classes in a folder called "Entity Models". This is not a convention.

EF Data Context

- The data context class has a rich set of functionality.
  - It includes properties for every entity that's present in the data store.
  - It supports the typical range of data store operations, including querying and data modification (add, change, delete).

EF Design Model Classes

- *Design model* classes describe the entities in the data store.
- They are simple classes consisting mostly of properties and often constructors.
- The Entity Framework maps a class to a table in the database. Each property is a column in a table.
- Relationships in a relational database management system are defined by properties called *Associations*.
- Optionally, you can create a "class diagram" in the models folder. A class diagram is a Visual Studio object that enables you to visually *diagram* (and design/edit) your classes.

Manager class (app and business logic)

- Our controllers NEVER work directly with the data store.
- We create a "manager" class in the *Controllers* folder to handle data service operations. This approach offers at least two benefits:
  - A task that is defined in the Manager can be used by any controller (if appropriate).
  - The Manager promotes the use of a layered system architecture, for safe and efficient coding.
- The Manager class has a reference to the *data context*. It has methods that can be called by controllers (e.g. "get all products", "add supplier", etc.).

Manager class (app and business logic)

- Another design feature is that data sent to and delivered from Manager methods are based on *view model* classes.
- We NEVER leak information or details about the data store.
- It is also possible that a Manager method can accept or deliver value types (i.e. int, double) or nothing. For completeness, *parameter* or *return types* in Manager methods must be one of the following:
  - empty parameter list, or null return type
  - a value type (see "Value Types" in this document)
  - a view model object
  - a view model collection

View Model Classes

- A *view model* class is used when interacting with the user. It is typically used for three purposes.
  1. Sending data to a view, for displaying/viewing.
  2. Sending data to a view, to provide initial data and settings for an HTML Form.
  3. Receiving data that was POSTed from a view that has an HTML Form.
- A view model class is customized for the use case.
- It takes on the exact shape of the data needed to fulfill the use case.
- A good view model will simplify coding and understanding. It uses property names and types that match those in design model classes plus other properties *(as required by the use case)*.

Why bother with View Model Classes?

- We follow some system architecture rules when building an app. The most important rule is:

  We NEVER allow user-interaction code in controllers to get access to the app's design. This includes the storage model (i.e. the database). Instead, we ALWAYS use a layered approach.

- To accomplish this, we need other classes that define the data that the user interacts with.
- Some advantages to using view model classes:
  - The way our app works with data is more flexible and adaptable.
  - Our code is safer because user interaction data is customized.

How to create View Model Classes

- *View model* classes should be written in code files located in the Models folder.
- Create a separate code file to hold each view model class needed for each entity or entity group.
- As an example, assume that you have a *Suppliers* and *Products* business domain model. You should create a code file named “SupplierViewModel.cs” and another named “ProductViewModel.cs”.
- To get started:
  - Create a class and copy the properties you need from the design model class.
  - Customize your class to match the use case.
- You can also use inheritance when creating view model classes.

What classes do you write?

- The answer is *totally* and *completely dependent* upon your app's use cases.
- There are some common and obvious use case patterns for data management:
  - Get all items
  - Get one item
  - Add new item
  - Edit existing item
  - Delete item

How to name View Model Classes

- Use a composite name – <entity><use case>ViewModel.

- Examples:
  - **ProductAddViewModel** - a class that includes the properties needed to fulfill the “add new” use case for a product object.
  - **ProductBaseViewModel** - a class that is used to display a product object.

Typical View Model Classes

- Following are some suggestions for writing view model classes.
- Notice the use of the entity's name in the classes. Notice also the inheritance chain.
- You do not have to write all these classes.
- Conversely, you may need more classes to handle some use cases. Don't be afraid of doing that.
- Keep in mind our example of the *Suppliers* and *Products* business domain model. A supplier can have zero or more products, and a product is always linked to one supplier.

<entity>List

- A class that can be used in the user interface as a simple lookup list.
- For example, a drop-down list.
- For all kinds of users.
- Include the Id property and one or two descriptive properties.

```
public class ProductListViewModel { }
```

<entity>AddForm

- A class that defines the data for an HTML Form.
- Designed for an HTML Form.
- Includes the initial (or default) data.
- Includes any other content used to help build the HTML Form. For example, the items that will appear in an item-selection list.
- Do NOT include the Id property.

```
public class ProductAddFormViewModel { }
```

<entity>Add

- A class that describes the data needed to create a new object.
- For all kinds of users.
- To be used for add-new tasks.
- Include properties that must be present in an add-new task.
- Do NOT include the Id property.

```
public class ProductAddViewModel { }
```

<entity>Base

- A class used for get-some and get-one tasks.
- For public or trusted users.
- Adds the Id property to the '<entity>Add' class (and any other useful/required properties).

```
public class ProductBaseViewModel
   : ProductAddViewModel { }
```

<entity>BaseWithAssociatedObject

- For get-some and get-one tasks.
- For public or trusted users.
- Adds the associated Supplier property to enable richer data display.

```
public class ProductBaseWithSupplierViewModel
   : ProductBaseViewModel { }
```

<entity>EditForm

- A class that defines the data for an HTML Form.
- For partially-trusted users.
- Includes all properties that you allow to be edited.
- May include other properties needed by the HTML Form.

```
public class ProductEditFormViewModel { }
```

<entity>Edit

- For update-item tasks.
- For partially-trusted users.
- Includes all properties that you allow to be edited.

```
public class ProductEditViewModel { }
```

<entity>Full

- To deliver all (most) properties of an object.
- For trusted users.
- Includes all (most) properties that you allow to be viewed or edited.
- Can include computed or temporary properties.

```
public class ProductFullViewModel
   : ProductBaseViewModel { }
```

Two kinds of classes!

- Problem: Design model and view model.
- Solution? AutoMapper

- Objects in the data store are defined by *design model* classes.
- We have just learned that we should NEVER allow user-interaction code to work with these objects.
- You must work with objects defined by *view model* classes.
- How do we go back-and-forth between these kinds of objects?

Mapping

- We use a library named AutoMapper to help with this task.
- AutoMapper was created by Jimmy Bogard, see his blog post.
- Convention-based mapper: When it maps between objects it simply maps/assigns values if both the *property type* and *property name* match.
- Easy.
- Non-matching properties are ignored.
- Static and Instance API: In 4.2.1 version (or later) AutoMapper provides two APIs: a static and an instance API.
- In this semester of the WEB524 course you are requested to use the instance API in your all assignments. Failure to do so will result in a big penalty on the assignment grade(s).
- You can add AutoMapper to your app using NuGet Package Manager.