DATA FLOW

Layers

Three layers should be enough for everybody (based on the Separation of Concerns Principle)

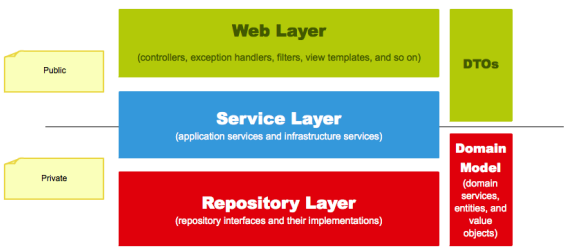
- If think about the responsibilities of a web application, we notice that a web application has the following “concerns”:

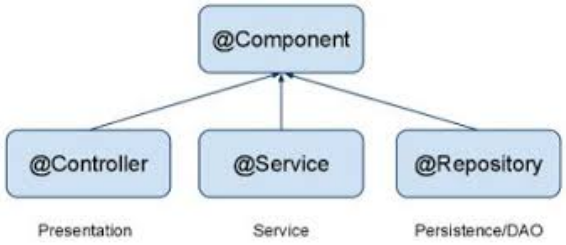
  It needs to process the user’s input and return the correct response back to the user.
  It needs an exception handling mechanism that provides reasonable error messages to the user.
  It needs a transaction management strategy.
  It needs to handle both authentication and authorization.
  It needs to implement the business logic of the application.
  It needs to communicate with the used data storage and other external resources.

We can fulfil all these concerns by using “only” three layers. These layers are:

- The **web layer** is the uppermost layer of a web application. It is responsible of processing user’s input and returning the correct response back to the user. The web layer must also handle the exceptions thrown by the other layers. Because the web layer is the entry point of our application, it must take care of authentication and act as a first line of defense against unauthorized users.
- The **service layer** resides below the web layer. It acts as a transaction boundary and contains both application and infrastructure services. The application services provides the public API of the service layer. They also act as a transaction boundary and are responsible of authorization. The infrastructure services contain the “plumbing code” that communicates with external resources such as file systems, databases, or email servers. Often these methods are used by more than a one application service.
- The **repository layer** is the lowest layer of a web application. It is responsible for communicating with the used data storage.

The components that belong to a specific layer can use the components that belong to the same layer or to the layer below it.





Spring layers

- **Controller**: responsible for displaying views
- **Service**: responsible for business logic
- **Repository**: responsible for data storage and retrieval (DAO - Data Access Objects)

Annotations

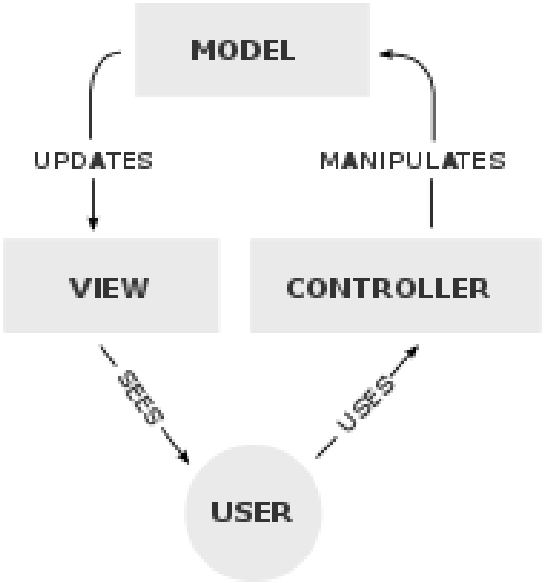
- **@Component** - generic stereotype for any Spring-managed component. We can use @Component across the application to mark the beans as Spring's managed components**.**
- **@Service, @Controller / RestController, @Repository and @Entity** - all "just" specializations of @Component for more specific use cases (@Component could be used instead, but these make the code more readable)
- **@(Rest)Controller** (stereotype for consumer/web layer) - responsible for triggering the creation of a (REST API) controller
- **@Service** (stereotype for service layer) - beans marked by ~ hold the business logic; responsible for triggering the creation of a service
- **@Repository** (stereotype for persistence layer) - responsible for triggering the creation of DAOs in our application. Also to catch persistence-specific exceptions and rethrow them as one of Spring's unified unchecked exceptions.
- **@Entity** (stereotype for persistence layer) - responsible for triggering the creation of an entity in the database

Service

- The difference between @Component and @Service annotation is that it will make your code more readable.
- A service component is where all your DAOs come together and have the business logic.
- For example: we are using a service that communicates between a controller and a database, like saving an entity.
- For dependency injection we create an interface for a service first then make the implementation (we haven't been doing this).

MVC

Stands for **Model - View - Controller**. It is an architectural pattern commonly used for developing user interfaces that divides an application into three interconnected parts.



Components

- The **model** is the central component of the pattern. It is the application's dynamic data structure, independent of the user interface. It directly manages the data, logic and rules of the application.
- A **view** can be any output representation of information, such as a chart or a diagram. Multiple views of the same information are possible, such as a bar chart for management and a tabular view for accountants.
- The third part or section, the **controller**, accepts input and converts it to commands for the model or view.

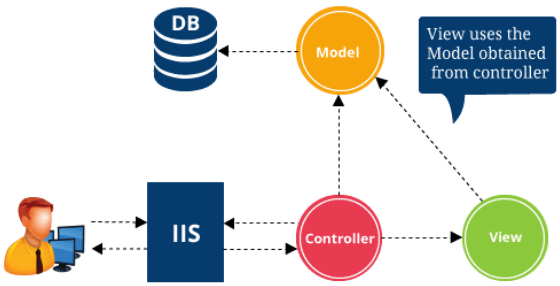
Interactions

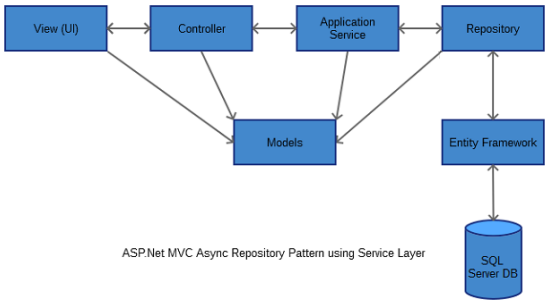
- In addition to dividing the application into three kinds of components, the model–view–controller design defines the interactions between them.
- The model is responsible for managing the data of the application. It receives user input from the controller.
- The view means presentation of the model in a particular format.
- The controller responds to the user input and performs interactions on the data model objects. The controller receives the input, optionally validates it and then passes the input to the model.

COMPONENTS

See above

Communication b/w LAYERS & COMPONENTS





Example

- We want to get an Item at the GET /{id} endpoint. In the GET /{id} mapped controller we are using an ItemService as a Service, that has an implementation. The implementation contains a code snippet which uses ItemRepository which is a Repository to get the item by its ID. Then it returns with an Item model filled with the information of the previously requested item. It returns to the controller which will display the Item.

Dependency Injection & Inversion of Control

DI

- **What is a dependency?** Whenever Class A relies on (=uses) methods, and thus instances, of another Class B, this relationship represents a dependency. In this context, Class A can be considered a Client and Class B a Service.
- **What is dependency injection?** It's the technique used to decouple the creation and the usage of the dependency. That is, the instance of Class B used by Class A will NOT be created in/by Class A - but it will be injected therein. DI is a middleman - takes Class A's order for an instance of Class B, makes sure it's created and provides it to Class A.
- **What does DI look like in practice?** Without DI:

```
class Car{
  private Wheels wheel = new MRFWheels();
  private Battery battery = new ExcideBattery();
  ...
```

```
}
```

There are basically 3 types of DI.
1) Constructor injection - the dependencies are provided through a class constructor
2) Setter injection - the client exposes a setter method that the injector uses to inject the dependency
3) Interface injection - the dependency provides an injector method that will inject the dependency into any
client passed to it. Clients must implement an interface that exposes a setter method that accepts the dependency

```
class Car{
  private Wheels wheel;
  private Battery battery;

  /*Somewhere in our codebase we instatiate the objects required by this class.
  */

  // Constructor Based
  Car(Wheel wh, Battery bt) {
    this.wh = wh;
    this.bt = bt;
  }

  // Setter Based
  void setWheel(Wheel wh){
    this.wh = wh;
  }
  ...
  ...
  // Rest of code
}
```

- **DI in Spring -** Starting with Spring 2.5, the framework introduced a new style of Dependency Injection driven by @Autowired Annotations. This annotation allows Spring to resolve and inject collaborating beans into your bean. @Autowired can be used directly on properties (fields), on setters methods or on

constructors (we have been using the first and the third method, and encouraged to rely on constructor injection).

- **What is DI good for, why do we need DI?** Helps you to follow SOLID's dependency inversion and single responsibility principles. The goal of SOLID design principles is to improve the reusability of your code. They also aim to reduce the frequency with which you need to change a class. Dependency injection supports these goals by decoupling the creation of the usage of an object. That enables you to replace dependencies without changing the class that uses them. It also reduces the risk that you have to change a class just because one of its dependencies changed.

(Optional) **Alternative to DI to achieve dependency inversion**

You can introduce interfaces to break the dependencies between higher and lower level classes. If you do that, both classes depend on the interface and no longer on each other.

That principle improves the reusability of your code and limits the ripple effect if you need to change lower

level classes. But even if you implement it perfectly, you still keep a dependency on the lower level class. The

interface only decouples the usage of the lower level class but not its instantiation. At some place in your code, you need to instantiate the implementation of the interface. That prevents you from replacing the implementation of the interface with a different one.

The goal of the dependency injection technique is to remove this dependency by separating the usage from

the creation of the object. This reduces the amount of required boilerplate code and improves flexibility.

IoC

- **Inversion of Control is a principle** in software engineering by which the control of objects or portions of a program is transferred to a container or framework. It's most often used in the context of object-oriented programming.
- By contrast with traditional programming, in which our custom code makes calls to a library, IoC enables a framework to take control of the flow of a program and make calls to our custom code. To enable this, frameworks use abstractions with additional behavior built in. If we want to add our own behavior, we need to extend the classes of the framework or plugin our own classes.
- **The advantages of this architecture are:**
    - decoupling the execution of a task from its implementation
    - making it easier to switch between different implementations
    - greater modularity of a program
    - greater ease in testing a program by isolating a component or mocking its dependencies and allowing

components to communicate through contracts

- Dependency injection is a pattern through which to implement IoC, where the control being inverted is the setting of object's dependencies. The act of connecting objects with other objects, or "injecting" objects into other objects, is done by an assembler rather than by the objects themselves.

- **The Spring IOC container -** An IoC container is a common characteristic of frameworks that implement IoC.
- In the Spring framework, the IoC container is represented by the interface ApplicationContext. The Spring container is responsible for instantiating, configuring and assembling objects known as beans, as well as managing their lifecycle.
- The Spring framework provides several implementations of the ApplicationContext interface — ClassPathXmlApplicationContext and FileSystemXmlApplicationContext for standalone applications, and

WebApplicationContext for web applications.

- In order to assemble beans, the container uses configuration metadata, which can be in the form of XML configuration or **annotations**.
- Dependency Injection in Spring can be done through constructors, setters or fields.