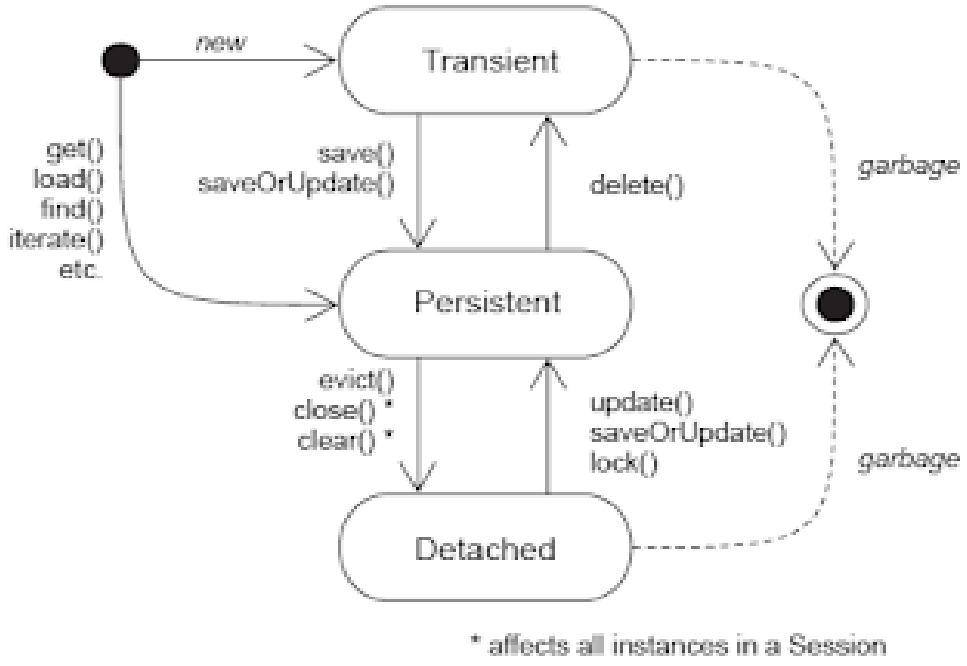
Hibernate – Entity LifeCycle

We know that Hibernate works with plain Java objects (POJO). In raw form (without hibernate specific annotations), hibernate will not be able to identify these java classes. But when these POJOs are properly annotated with required annotations then hibernate will be able to identify them and then work with them e.g. store in the database, update them, etc. These POJOs are said to mapped with hibernate.

1. Entity Lifecycle States

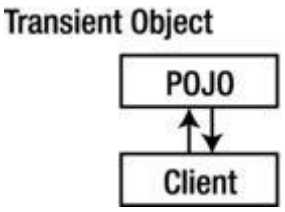
Given an instance of a class that is mapped to Hibernate, it can be in any one of four different persistence states (known as hibernate entity lifecyle states):

1. **Transient**
2. **Persistent**
3. **Detached**
4. **Removed**



1.1. Transient

Transient entities exist in heap memory as normal Java objects. Hibernate does not manage transient entities or persist changes done on them.

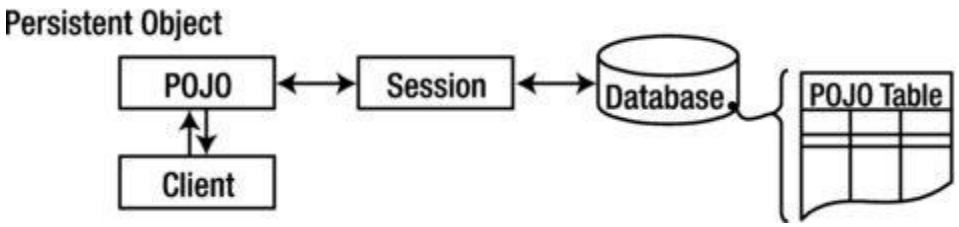


Transient objects are independent of Hibernate

To persist the changes to a transient entity, we would have to ask the hibernate session to save the transient object to the database, at which point Hibernate assigns the object an identifier and marks the object as being in persistent state.

1.2. Persistent

Persistent entities exist in the database, and Hibernate manages the persistence for persistent objects.

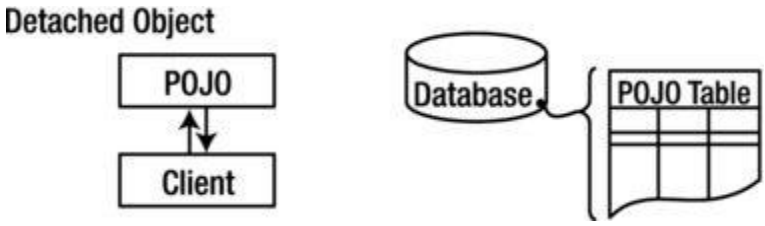


Persistent objects are maintained by Hibernate

If fields or properties change on a persistent object, Hibernate will keep the database representation up to date when the application marks the changes as to be committed.

1.3. Detached

Detached entities have a representation in the database, but changes to the entity will not be reflected in the database, and vice-versa. This temporary separation of the entity and the database is shown in the image below.



Detached objects exist in the database but are not maintained by Hibernate

A detached entity can be created by closing the session that it was associated with, or by evicting it from the session with a call to the session's `evict()` method.

One reason you might consider doing this would be to read an object out of the database, modify the properties of the object in memory, and then store the results some place other than your database. This would be an alternative to doing a deep copy of the object.

In order to persist changes made to a detached object, the application must re-attach it to a valid Hibernate session. A detached instance can be associated with a new Hibernate session when your application calls one of the `load()`, `refresh()`, `merge()`, `update()`, or `save()` methods on the new session with a reference to the detached object.

After the method call, the detached entity would be a persistent entity managed by the new Hibernate session.

1.4. Removed

Removed entities are objects that are being managed by Hibernate (persistent entities, in other words) that have been passed to the session's `remove()` method.

When the application marks the changes held in the session as to be committed, the entries in the database that correspond to removed entities are deleted.

Now let's not note down the take-aways from this tutorial.

2. Conclusion

1. **Newly created POJO object will be in the transient state. Transient entity doesn't represent any row of the database i.e. not associated with any session object. It's plain simple java object.**
2. **Persistent entity represents one row of the database and always associated with some unique hibernate session. Changes to persistent objects are tracked by hibernate and are saved into database when commit call happen.**
3. **Detached entities are those who were once persistent in the past, and now they are no longer persistent. To persist changes done in detached objects, you must re-attach them to hibernate session.**
4. **Removed entities are persistent objects that have been passed to the session's `remove()` method and soon will be deleted as soon as changes held in the session will be committed to database.**

Spring 5 Introduction :

Advantages of Spring:

The benefits of Spring Framework include:

- **Plain Old Java Object — Developers call this POJO. The reason this is so beneficial is that it means developers do not have to use a server or any other enterprise container. This makes the entire framework extremely lightweight, which is a significant advantage when developing web applications.**
- **Flexible Configurations — Developers have the option to choose either XML or Java-based annotations for configuration purposes. Having such an option makes the jobs of developers a lot simpler.**

- **The AOP Module — Developers can have different compilation units or a separate class loader.**
- **Testing is Easier — The Spring Dependency injection helps developers insert test data.**

While there are advantages of Spring framework, there are some disadvantages as well, and they are listed below.

Disadvantages of Spring Framework:

The cons of Spring are:

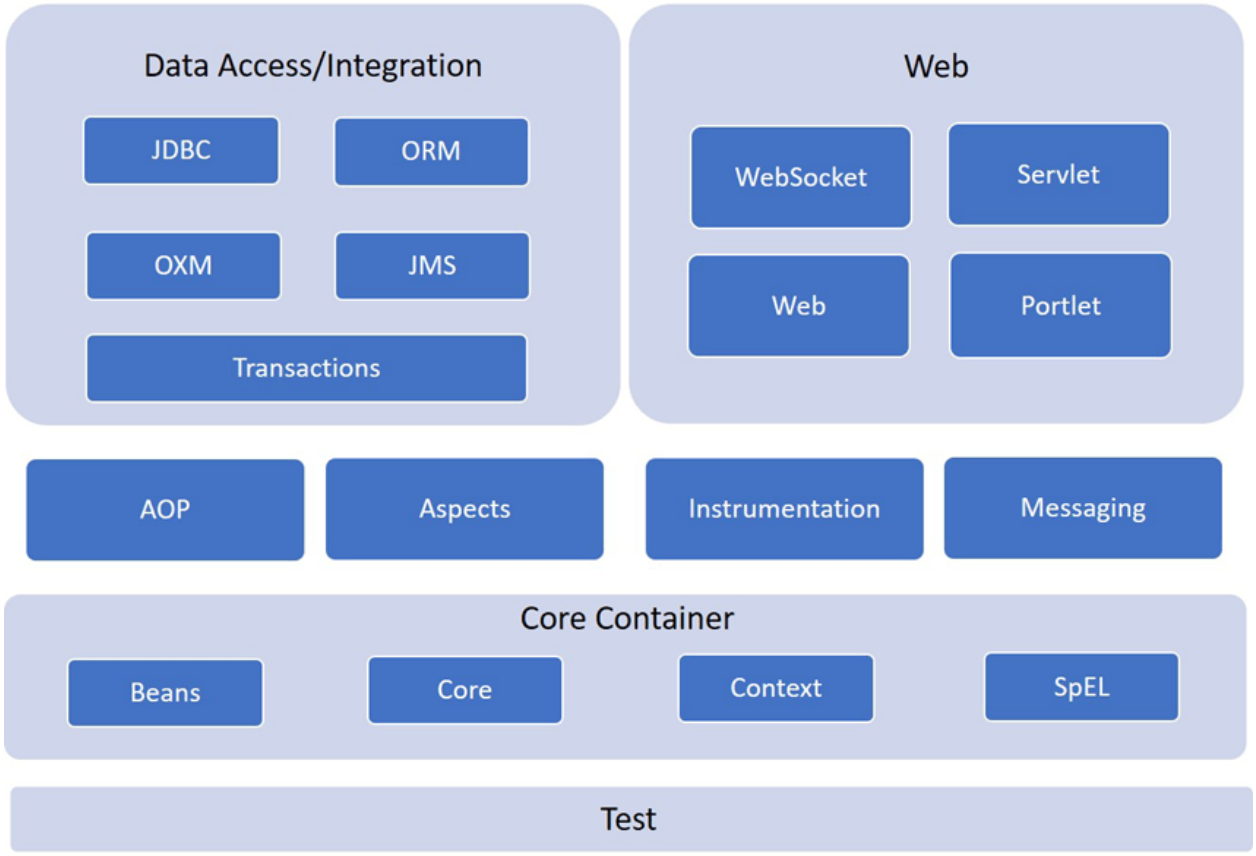
- **Complexity — The Spring framework has a lot of variables and complications. Therefore, you should only use it if you have an experienced**

team of developers who have used this framework before. The learning curve will be difficult, so if you or your team does not have a lot of development experience, it would be better to choose something else.

- **Parallel Mechanisms — One of the biggest advantages of Spring is that it gives developers a wide array of options, but this could also be a disadvantage because it causes confusion. Developers have to know which features will be useful, and making the wrong decisions could lead to significant delays.**
- **No Specific Guidelines — Within the Spring documentation, it says nothing about dealing with threats such as XSS or cross-site request forgery. With this in mind, you and your team**

will need to figure out ways on how to stop hackers from infiltrating your application yourself.

Spring Framework Architecture :



Aspect Oriented Programming (AOP) compliments OOPs in the sense that it also provides modularity. But the key unit of modularity is aspect than class.

AOP breaks the program logic into distinct parts (called concerns). It is used to increase modularity by **cross-cutting concerns**.

A **cross-cutting concern** is a concern that can affect the whole application and should be centralized in one location in code as possible, such as transaction management, authentication, logging, security etc.

The Spring Framework recommends you to use Spring AspectJ AOP implementation over the Spring 1.2 old style dtd based AOP implementation because it provides you more control and it is easy to use.

There are two ways to use Spring AOP AspectJ implementation:

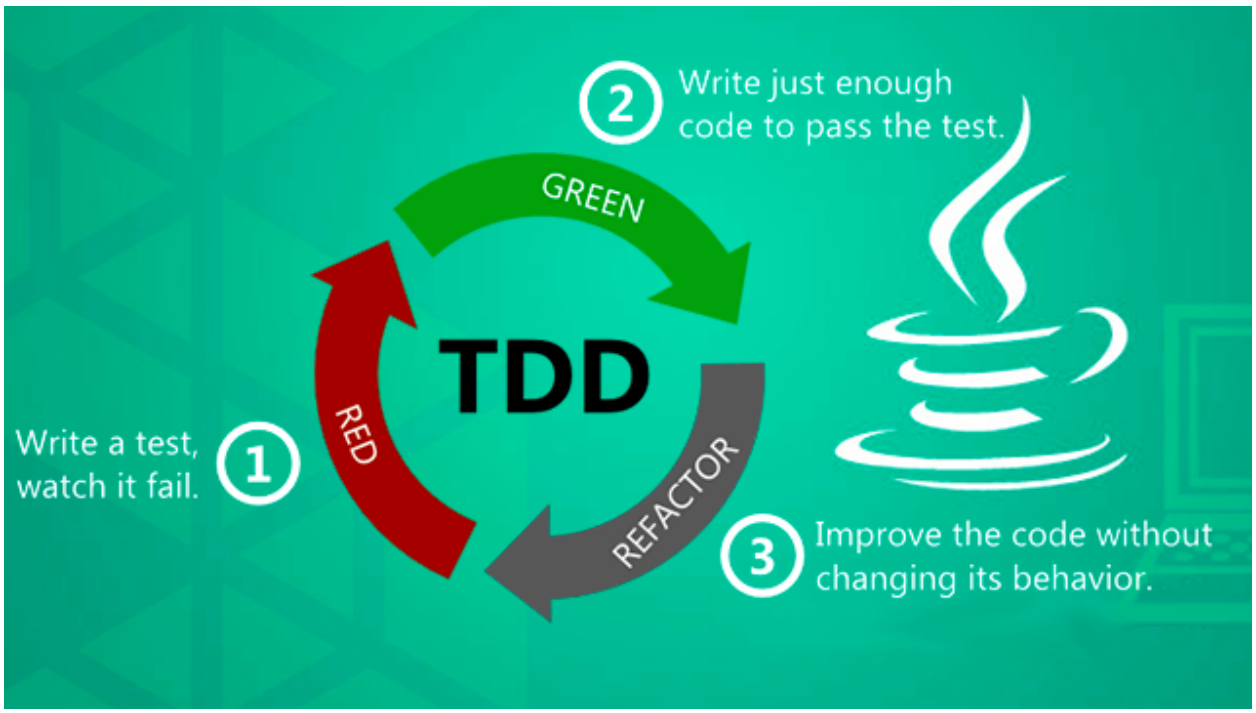
Whatsapp
Email
Alerts

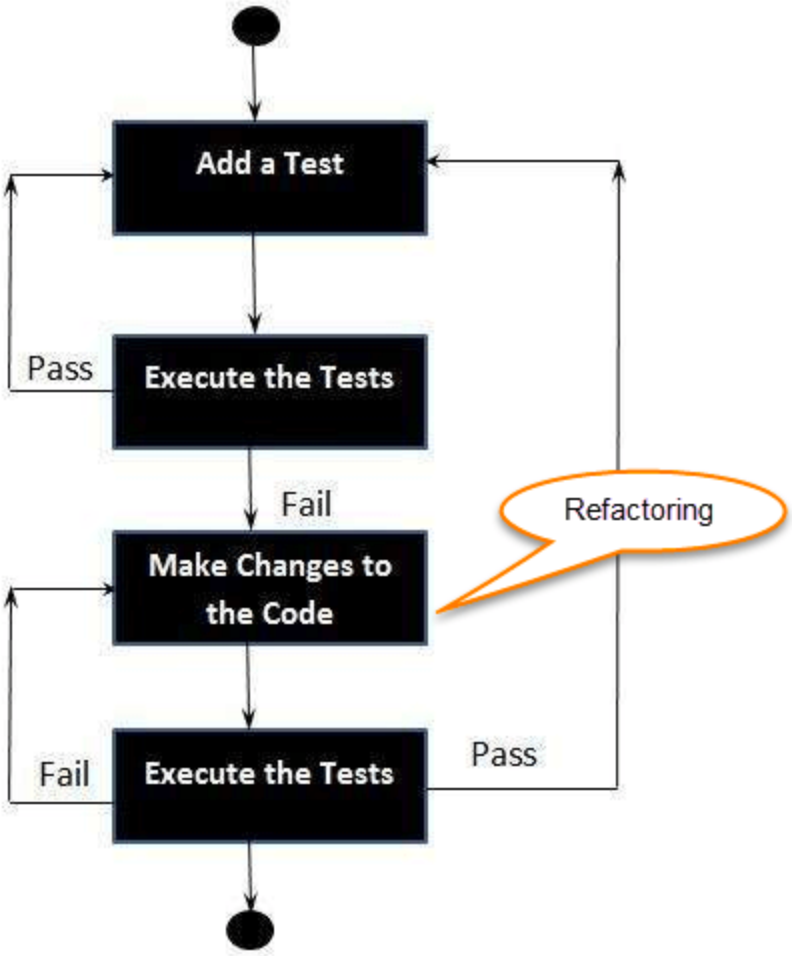
Code optimization tool : PMD

Test Driven Development:

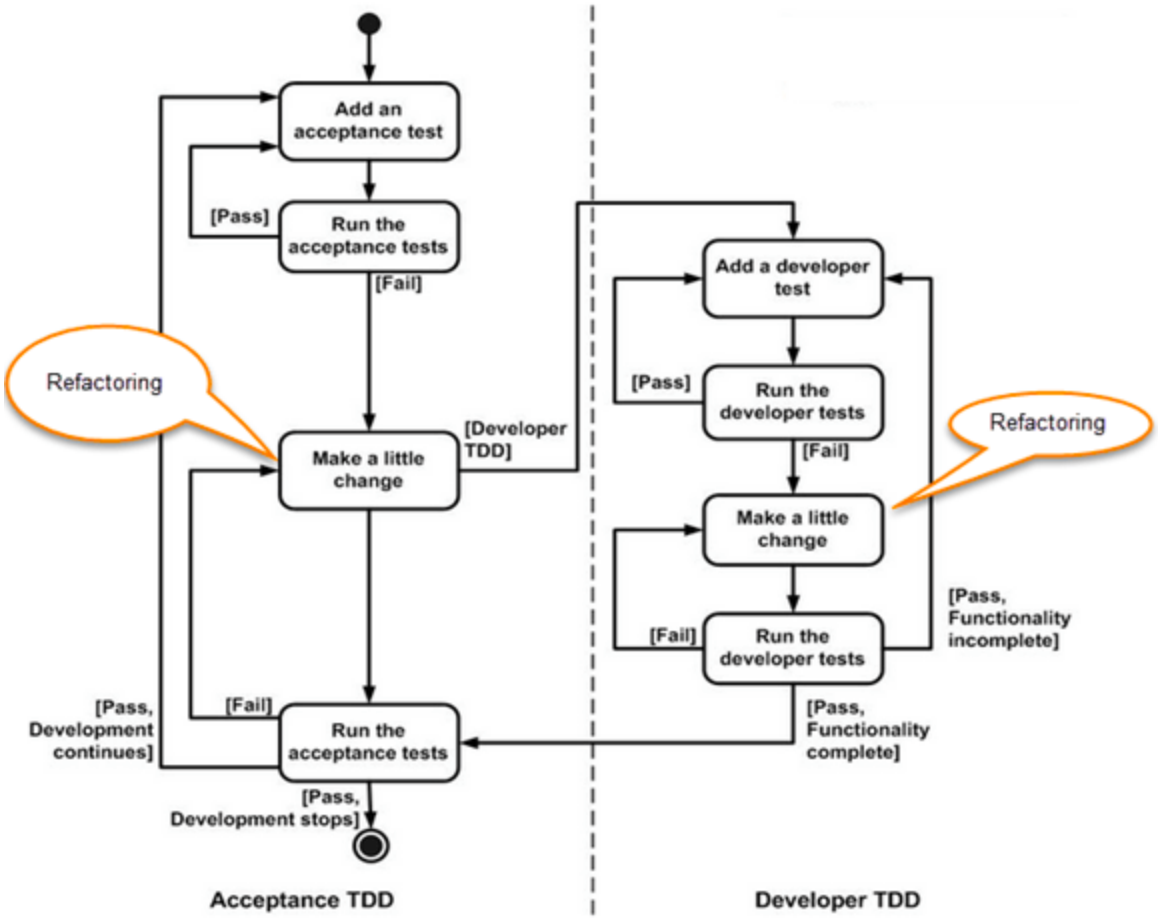
Test Driven Development (TDD) is a software development approach in which test cases are developed to specify and validate what the code will do. In simple terms, test cases for each functionality are created and tested first and if the test fails then the new code is written in order to pass the test and make code simple and bug-free.

Test-Driven Development starts with designing and developing tests for every small functionality of an application. TDD instructs developers to write new code only if an automated test has failed. This avoids duplication of code. The full form of TDD is Test-driven development.





Pass. Development Stops



Unit Testing With Mockito

Basic Simple Demo code for using mockito with JUnit –

1. Production Code

```
public class TestService {
 public int getUniqueId() {
  return 43;
 }
}
```

2. Testing Code

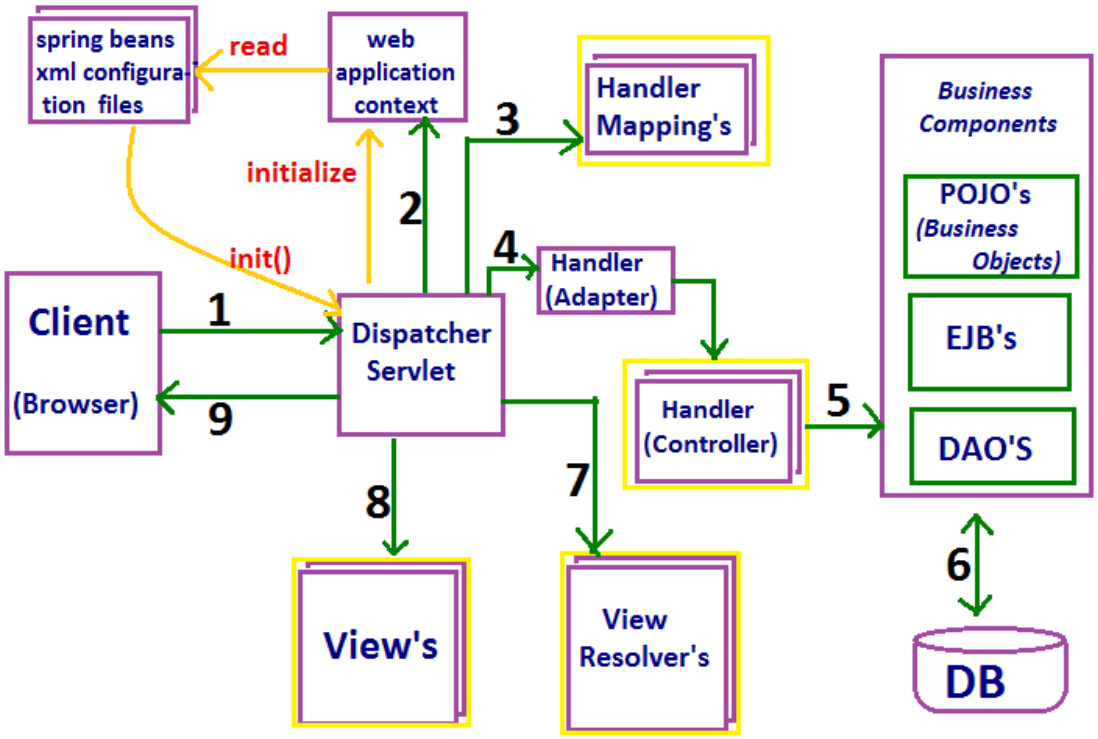
```
public class JUnitServiceTestExample {

 @Test
 public void testGetInt() {
  // create mock
  TestService test = Mockito.mock(TestService.class);

  // define return value for method getUniqueId()
  when(test.getUniqueId()).thenReturn(43);

  // use mock in test....
  assertEquals(test.getUniqueId(), 43);
 }
}
```

Spring Boot:



SCOPE	DESCRIPTION
`singleton` (default)	Single bean object instance per spring IoC container

`prototype`	Opposite to singleton, it produces a new instance each and every time a bean is requested.
`request`	A single instance will be created and available during complete lifecycle of an HTTP request. Only valid in web-aware Spring `ApplicationContext`.
`session`	A single instance will be created and available during complete lifecycle of an HTTP Session. Only valid in web-aware Spring `ApplicationContext`.
`application`	A single instance will be created and available during complete lifecycle of `ServletContext`. Only valid in web-aware Spring `ApplicationContext`.

`websocket`	A single instance will be created and available during complete lifecycle of `WebSocket`. Only valid in web-aware Spring `ApplicationContext`.

1.1. singleton scope

`singleton` is default bean scope in spring container. It tells the container to create and manage only one instance of bean class, per container. This single instance is stored in a cache of such singleton beans, and all subsequent requests and references for that named bean return the cached instance.

Example of singleton scope bean using Java config –

```
@Component
//This statement is redundant - singleton is default scope
@Scope("singleton")  //This statement is redundant
public class BeanClass {

}
```

Example of singleton scope bean using XML config –

```
<!-- To specify singleton scope is redundant -->
<bean id="beanId" class="com.howtodoinjava.BeanClass" scope="singleton" />
//or
<bean id="beanId" class="com.howtodoinjava.BeanClass" />
```

1.2. prototype scope

`prototype` scope results in the creation of a new bean instance every time a request for the bean is made by application code.

You should know that destruction bean lifecycle methods are not called `prototype` scoped beans, only initialization callback methods are called. So as developer, you are responsible for clean up prototype-scoped bean instances and any resource there hold.

Java config example of prototype bean scope –

```
@Component
@Scope("prototype")
public class BeanClass {
}
```

XML config example of prototype bean scope –

```
<bean id="beanId" class="com.howtodoinjava.BeanClass" scope="prototype" />
```

As a rule, you should prefer to use the `prototype` scope for all stateful beans and the `singleton` scope for stateless beans.

To use beans in the request, session, application and websocket scopes, you need to register the `RequestContextListener` or `RequestContextFilter`.

1.3. request scope

In `request` scope, container creates a new instance for each and every HTTP request. So, if server is currently handling 50 requests, then container can have at most 50 individual instances of bean class. Any state change to one instance, will not be visible to other instances. These instances are destructed as soon as the request is completed.

Java config example of request bean scope –

```
@Component
@Scope("request")
public class BeanClass {
}

//or

@Component
@RequestScope
public class BeanClass {
}
```

XML config example of request bean scope –

```
<bean id="beanId" class="com.howtodoinjava.BeanClass" scope="request" />
```

Read More: How web servers work?

1.4. session scope

In `session` scope, container creates a new instance for each and every HTTP session. So, if server has 20 active sessions, then container can have at most 20 individual instances of bean class. All HTTP requests within single session lifetime will have access to same single bean instance in that session scope.

Any state change to one instance, will not be visible to other instances. These instances are destructed as soon as the session is destroyed/end on server.

Java config example of session bean scope –

```
@Component
@Scope("session")
public class BeanClass {
}

//or

@Component
@SessionScope
public class BeanClass {
}
```

XML config example of session bean scope –

```
<bean id="beanId" class="com.howtodoinjava.BeanClass" scope="session" />
```

1.5. application scope

In `application` scope, container creates one instance per web application runtime. It is almost similar to `singleton` scope, with only two differences i.e.

1. `application` scoped bean is singleton per `ServletContext`, whereas `singleton` scoped bean is singleton per `ApplicationContext`. Please note that there can be multiple application contexts for single application.
2. `application` scoped bean is visible as a `ServletContext` attribute.

Java config example of application bean scope –

```
@Component
@Scope("application")
public class BeanClass {
}

//or

@Component
@ApplicationScope
public class BeanClass {
}
```

XML config example of application bean scope –

```
<bean id="beanId" class="com.howtodoinjava.BeanClass" scope="application"
/>
```

1.6. websocket scope

The WebSocket Protocol enables two-way communication between a client and a remote host that has opted-in to communication with client. WebSocket Protocol provides a single TCP connection for traffic in both directions. This is specially useful for multi-user applications with simultaneous editing and multi-user games.

In this type of web applications, HTTP is used only for the initial handshake. Server can respond with HTTP status 101 (switching protocols) if it agrees – to handshake request. If the handshake succeeds, the TCP socket remains open and both client and server can use it to send messages to each other.

Java config example of websocket bean scope –

```
@Component
@Scope("websocket")
public class BeanClass {
}
```

XML config example of websocket bean scope –

```
<bean id="beanId" class="com.howtodoinjava.BeanClass" scope="websocket" />
```

Please note that `websocket` scoped beans are typically singletons and live longer than any individual WebSocket session.

2. Custom thread scope

Spring also provide a non-default `thread` scope using class `SimpleThreadScope`. To use this scope, you must use register it to container using `CustomScopeConfigurer` class.

```
<bean
class="org.springframework.beans.factory.config.CustomScopeConfigurer">
    <property name="scopes">
        <map>
            <entry key="thread">
                <bean
class="org.springframework.context.support.SimpleThreadScope"/>
            </entry>
        </map>
    </property>
</bean>
```

Every request for a bean will return the same instance within the same thread.

Java config example of thread bean scope –

```
@Component
@Scope("thread")
public class BeanClass {
}
```

XML config example of thread bean scope –

```
<bean id="beanId" class="com.howtodoinjava.BeanClass" scope="thread" />
```

Microservices:

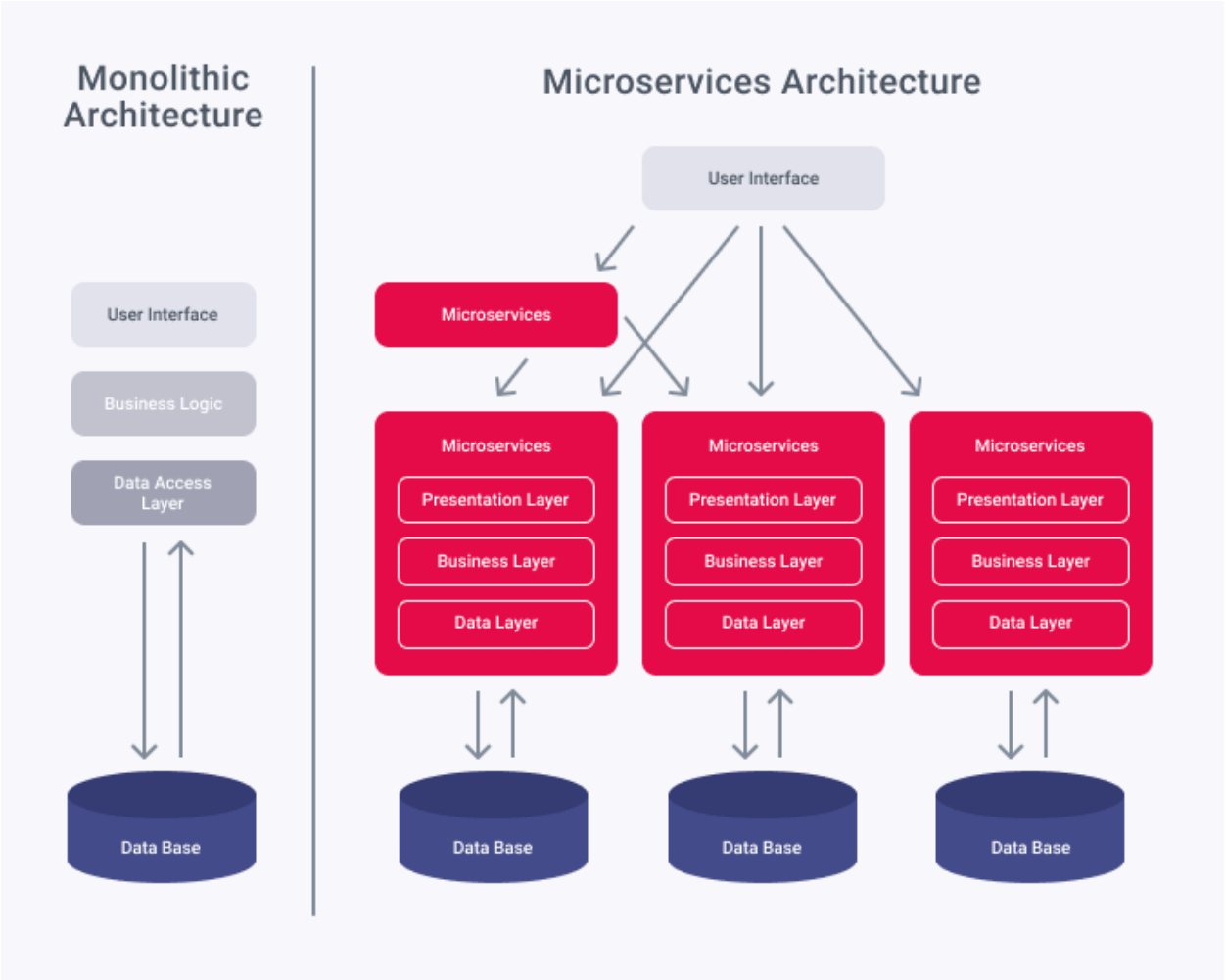
Application's architecture describes the behavior of applications that focuses on how they interact with each other and with users.

In a web application architecture, there are major independent software architecture components called layers. Here they are:

1. Presentation Layer (Presentation Services and Presentation Logic).
2. Business Logic (Business or Application and Data Logic).
3. Data Layer (Data Services and File Services).

Let's look at the components of the application's layers and their functions a bit closer:

- Presentation Services are means of presenting info on the screen.
- The logic of the presentation describes the rules and scenarios of user interaction with an application.
- Business or Application Logic describes rules for making decisions, computational procedures, etc.
- Data logic means operations with data stored.
- Data Services perform internal DB operations – DBMS actions called in response to queries.
- File Services conduct standard actions with files and a file system.



Microservices vs Monolithic Applications

In a monolithic architecture, the app has a single code base with multiple modules that are dependent upon each other and a common database.

A microservice architecture, in its turn, consists of autonomous API – interconnected services. Each microservice is self-contained and implements a specific business capability. Each microservice possesses in itself a presentation layer, business logic layer, and data layer.

Monolith vs Microservices Pros and Cons

Development and Deployment

The monolithic architecture is easier to develop as it doesn't require a diverse technology stack and is frequently limited to several commonly used frameworks or programming languages. Development is performed with one directory and deployed fast on the server after testing.

With microservice architecture, you'll need a team familiar with a larger amount of technologies. Microservices require more time to develop, test and deploy. But at the same time, it's easier to roll back one of them or make changes to the project comparing to a monolith app.

Microservice vs Monolithic Performance

Monolithic apps with a properly built architecture show good performance, especially in the early stages of the project. Over time, as the product develops and enters new markets, the number of users and the load increases. As bottlenecks in performance appear, most of the big and successful applications are transferred to microservices. In microservices, a load balancer distributes the load over the servers. There are also services (e.g. Elastic Beanstalk) that can vary the number of servers to stabilize the work of application for all users.

Accessibility and Reliability

These two things are most important for today's users. We transfer personal info, fulfill payments, demand high-quality service here, now and at full volume. That's why we are not ready to tolerate the unavailability of the app, its decline, or running out of work. It happens that a customer cannot correctly assess the estimated load on the application. One lays a certain percentage of the margin, but in reality, the number of requests to the system can grow many times. Sometimes startups may get into such situations, facing explosive growth. Still, even experienced companies that brought the product for a specific short-term goal (e.g. population census) may encounter problems with availability. For such situations, microservices is a good reply. In the case of server overload or crash, an additional one will be connected.

Scalability

Monolithic architecture cannot grow infinitely. Continuous growth can lead to two unfortunate consequences: Firstly, controllability falls, and secondly, performance problems begin.

With Microservices when the need for growth arises, it is not necessary to scale the entire system, disassembling it to the base. It is enough to make the required changes only at a particular microservice.

What Architecture is Better For What Projects

Many successful apps started as a monolith and transferred to microservice with time. It is possible when modules are loosely coupled. When the monolith grows, and complexity cannot be controlled, when making changes is getting harder, then going to microservices is a good idea. Sometimes when the app wasn't developed following the Dependency Inversion Principle, then it needs to be rewritten for the microservices.

Monolith architecture is ideal if you:

– have a small team

– have an MVP version of a new product

– need a fast time to market

– don't expect explosive growth

– don't plan to conquer new markets.

Microservice architecture is beneficial if:

– you don't have a deadline for launching a product

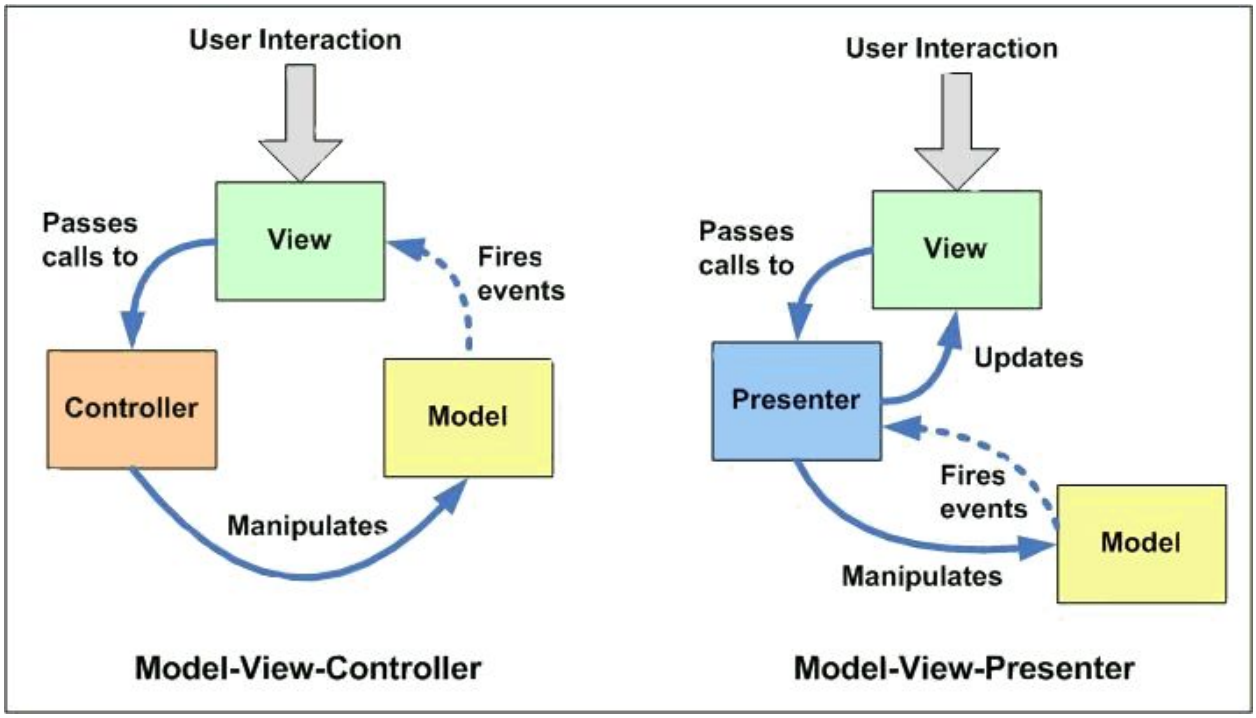
– scalability and reliability go first for you

– you need to make constant complex changes to the product.

Conclusions

Monolithic architecture is better for MVPs, or simple apps and shows good performance. Microservices are suitable for complex projects with lots of users and give high accessibility, reliability, and scalability.

Model–view–presenter (**MVP**) is a derivation of the model–view–controller (MVC) **architectural pattern**, and is used mostly for building user interfaces. In **MVP**, the presenter assumes the functionality of the "middle-man". In **MVP**, all presentation logic is pushed to the presenter.


User Interaction
Passes calls to
View
Fires events
Controller
Model
Manipulates
Model-View-Controller
User Interaction
Passes calls to
View
Updates
Presenter
Fires events
Model
Manipulates
Model-View-Presenter