Spring SOAP Web-Services Life cycle

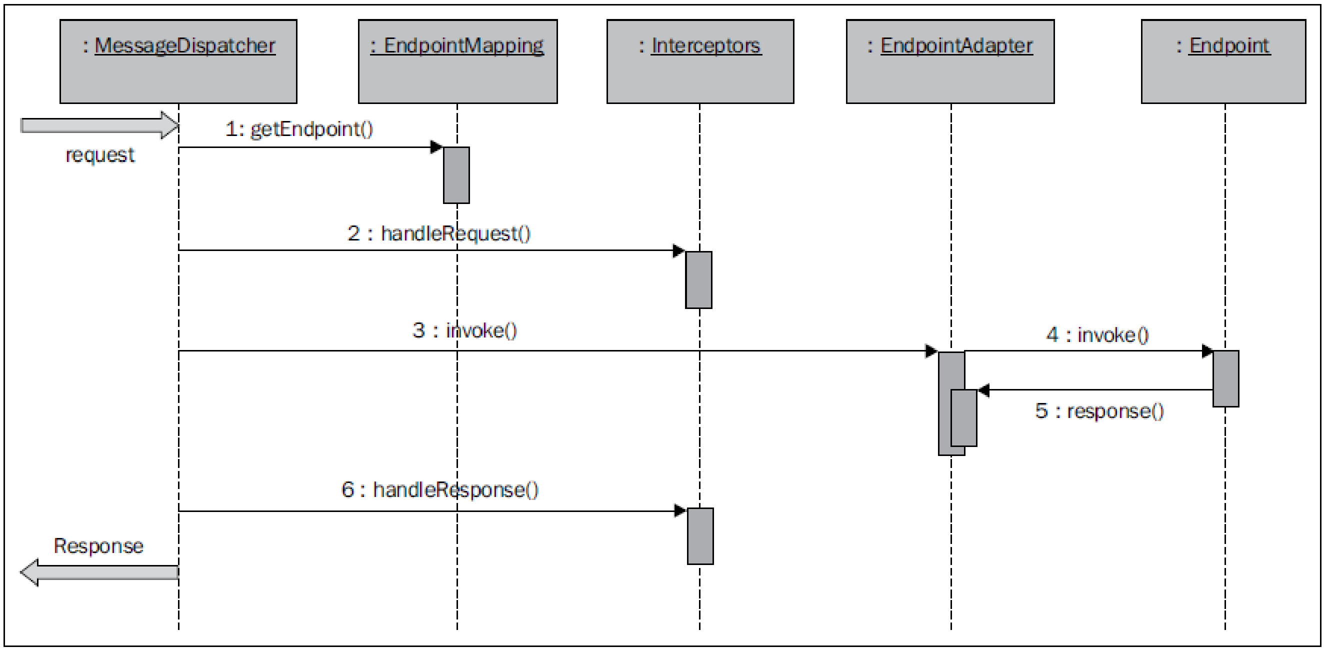
MessageDispatcher is the central point for a Spring Web-Service and dispatches Web- Service messages to the registered endpoint. In Spring-WS, request/response messages are wrapped inside the MessageContext object and the MessageContext will be passed to the MessageDispatcher (response will be set into MessageContext after invoking the endpoint). When a message arrives, MessageDispatcher uses the request object to get the endpoint. (Mapping a request to an endpoint is called endpoint mapping and it can be done by using data from beans registration within application context, scanning, and autodetection of annotations). Then the MessageDispatcher by using the endpoint, gets endpopint's interceptors (which range from zero to many) and calls handleRequest method on them.

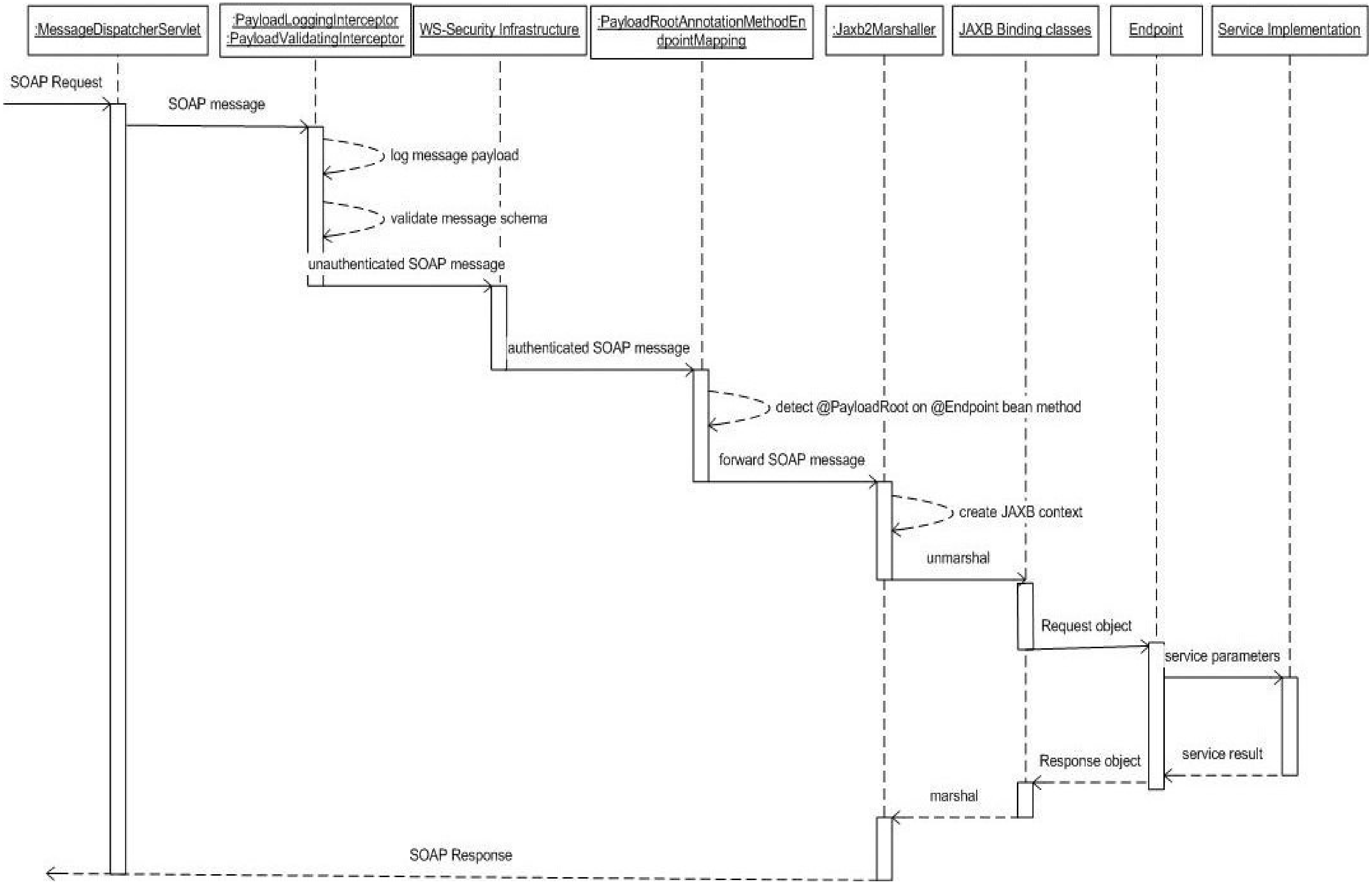
An interceptor (EndpointInterceptor here), as the name suggests, intercepts the request/response to perform some operations prior to (for request)/after (for response) invoking the endpoint. This EndpointInterceptor gets called before/after calling the appropriate endpoint to perform several processing aspects such as logging, validating, security, and so on. Next, MessageDispatcher gets appropriate endpoint adapter for the endpoint method to be called. This adapter offers compatibility with various types of endpoint methods. Each adapter is specialized to call a method with specific method parameter and return type. And Finally, EndpointAdapter invokes the endpoint's method and transforms the response to the desired form and set it into the MessageContext object. Now the initial message context that was passed to MessageDispatcher, contains the response object, that will be forwarded to the client (by the caller of MessageDispatcher). Spring-WS only supports the contract-first development style in which creating the contract (XSD or WSDL) is the first step. The required steps to build a contract-first Web-Service using

Spring-WS are as follows:

1. Contract definition (either XSD or WSDL)
2. Creating endpoint: the class that receives and processes an incoming message.
3. Configuration of Spring beans and the endpoint.

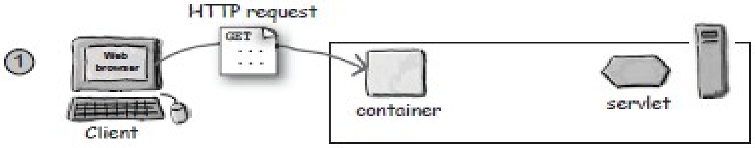
There are two types of endpoints, namely, payload endpoints and message endpoints. While message endpoints can access the entire XML SOAP envelop, the payload endpoint will only access the payload part of a SOAP envelop, that is, the body of a SOAP envelop.



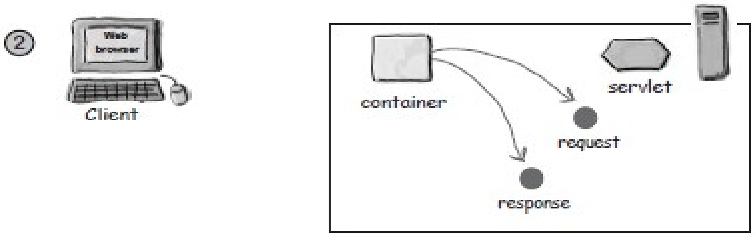

:MessageDispatcherServlet
:PayloadLoggingInterceptor
:PayloadValidatingInterceptor
WS-Security Infrastructure
:PayloadRootAnnotationMethodEndpointMapping
:Jaxb2Marshaller
JAXB Binding classes
Endpoint
Service Implementation
SOAP Request
SOAP message
log message payload
validate message schema
unauthenticated SOAP message
authenticated SOAP message
detect @PayloadRoot on @Endpoint bean method
forward SOAP message
create JAXB context
unmarshal
Request object
service parameters
service result
Response object
marshal
SOAP Response

Servlet Life cycle

How the Container handles a request



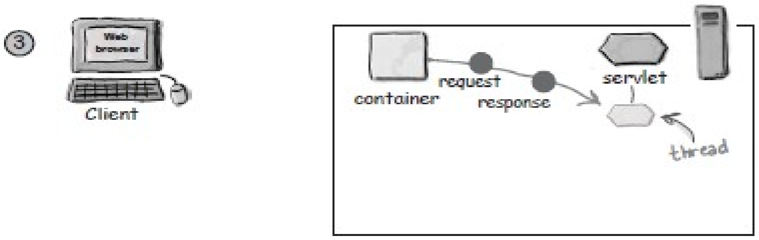
User clicks a link that has a URL to a servlet instead of a static page.



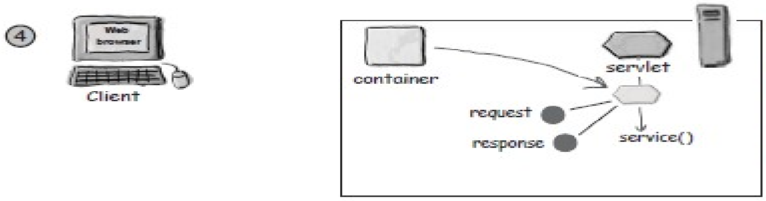
The container "sees" that the request is for a servlet, so the container creates two objects:

1) HttpServletResponse

2) HttpServletRequest

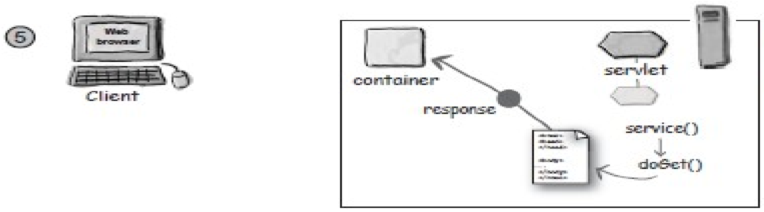


The container finds the correct servlet based on the URL in the request, creates or allocates a thread for that request, and passes the request and response objects to the servlet thread.

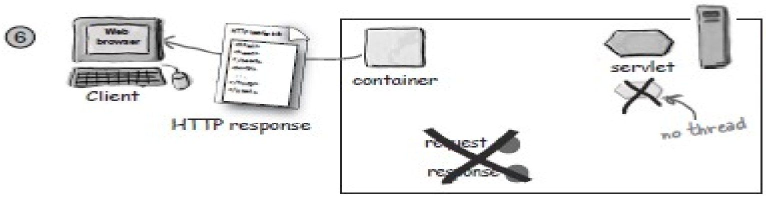


The container calls the servlet's service() method. Depending on the type of request, the service() method calls either the doGet() or doPost() method.

For this example, we'll assume the request was an HTTP GET.



The doGet() method generates the dynamic page and stuffs the page into the response object. Remember, the container still has a reference to the response object!

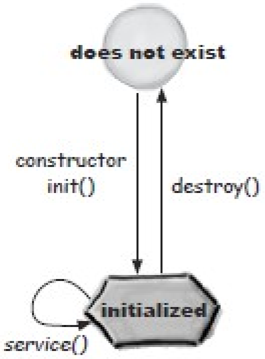


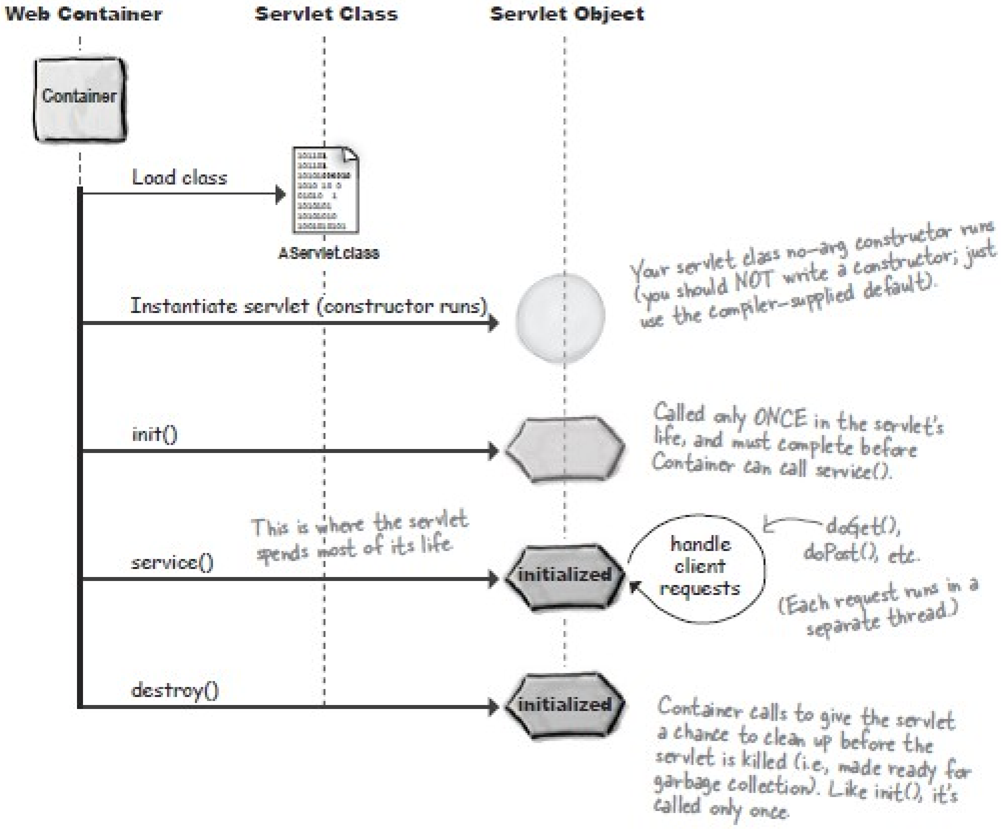
The thread completes, the container converts the response object into an HTTP response, sends it back to the client, then deletes the request and response objects.

But there's more to a servlet's life

We stepped into the middle of the servlet's life, but that still leaves questions: when was the servlet class loaded? When did the servlet's constructor run? How long does the servlet object live? When should your servlet initialize resources? When should it clean up its resources?

The servlet lifecycle is simple; there's only one main state—***initialized***. If the servlet isn't initialized, then it's either *being initialized* (running its constructor or init()method), *being destroyed* (running its destroy() method), or it simply ***does not exist.***





Your servlet class no-arg constructor runs (you should NOT write a constructor; just use the compiler-supplied default).

Called only ONCE in the servlet's life, and must complete before Container can call service().

This is where the servlet spends most of its life.

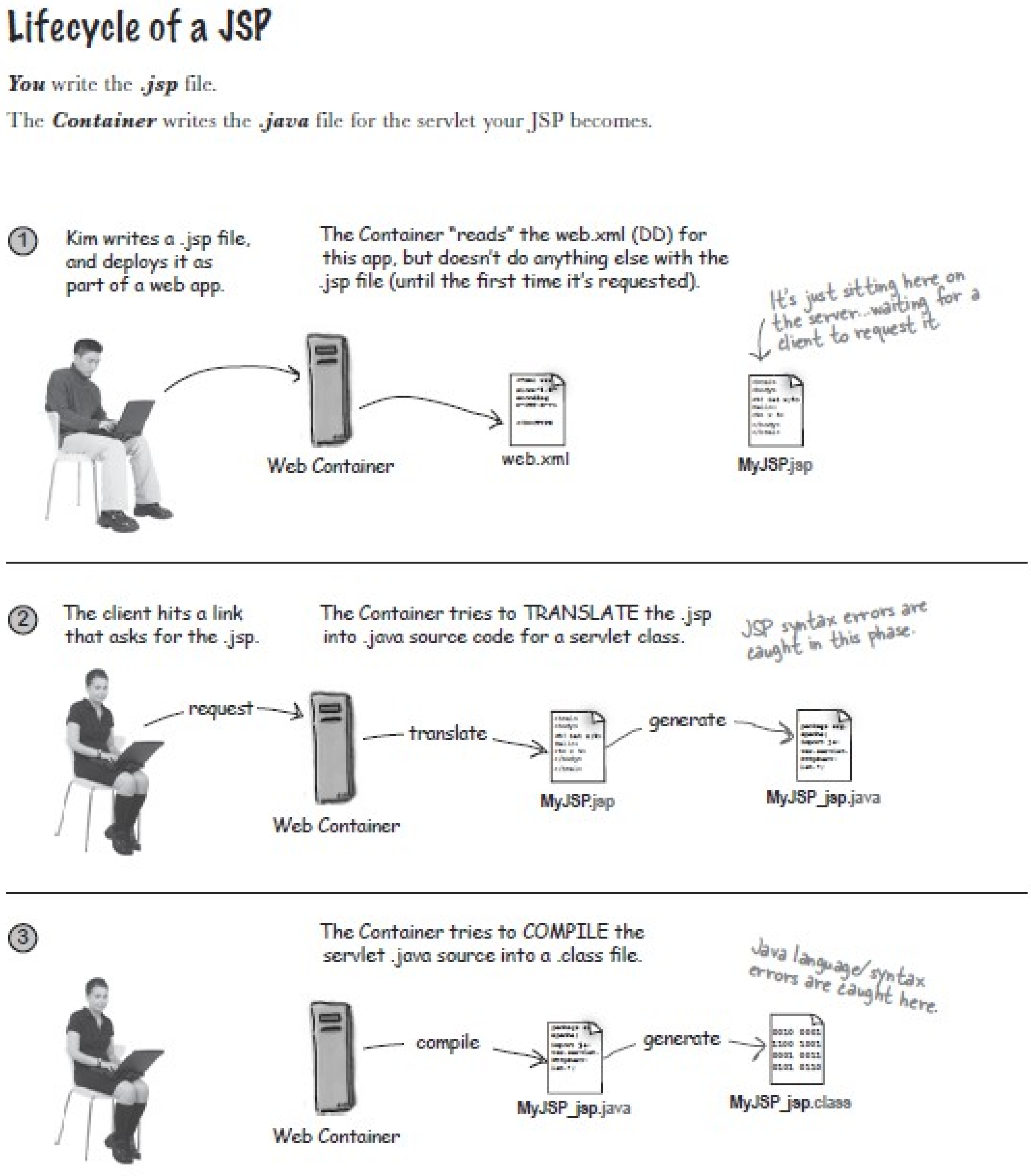
(Each request runs in a separate thread.)

Container calls to give the servlet a chance to clean up before the servlet is killed (i.e., made ready for garbage collection). Like init(), it's called only once.

Bullet Points

- The Container initializes a servlet by loading the class, invoking the servlet's no-arg constructor, and calling the servlet's init() method.
- The init() method (which the developer can override) is called only once in a servlet's life, and always before the servlet can service any client requests.
- The init() method gives the servlet access to the ServletConfig and ServletContext objects, which the servlet needs to get information about the servlet configuration and the web app.
- The Container ends a servlet's life by calling its destroy() method.
- Most of a servlet's life is spent running a service() method for a client request.
- Every request to a servlet runs in a separate thread! There is only one instance of any particular servlet class.
- Your servlet will almost always extend javax.servlet.http. HttpServlet, from which it inherits an implementation of the service() method that takes an HttpServletRequest and an HttpServletResponse.

- HttpServlet extends javax.servlet.GenericServlet—an abstract class that implements most of the basic servlet methods.
- GenericServlet implements the Servlet interface.
- Servlet classes (except those related to JSPs) are in one of two packages: javax.servlet or javax.servlet.http.
- You can override the init() method, and you must override at least one service method (doGet(), doPost(), etc.).



Lifecycle of a JSP

You write the ***.jsp*** file.

The ***Container*** writes the ***.java*** file for the servlet your JSP becomes.

① Kim writes a .jsp file, and deploys it as part of a web app.

The Container "reads" the web.xml (DD) for this app, but doesn't do anything else with the .jsp file (until the first time it's requested).

It's just sitting here on the server...waiting for a client to request it

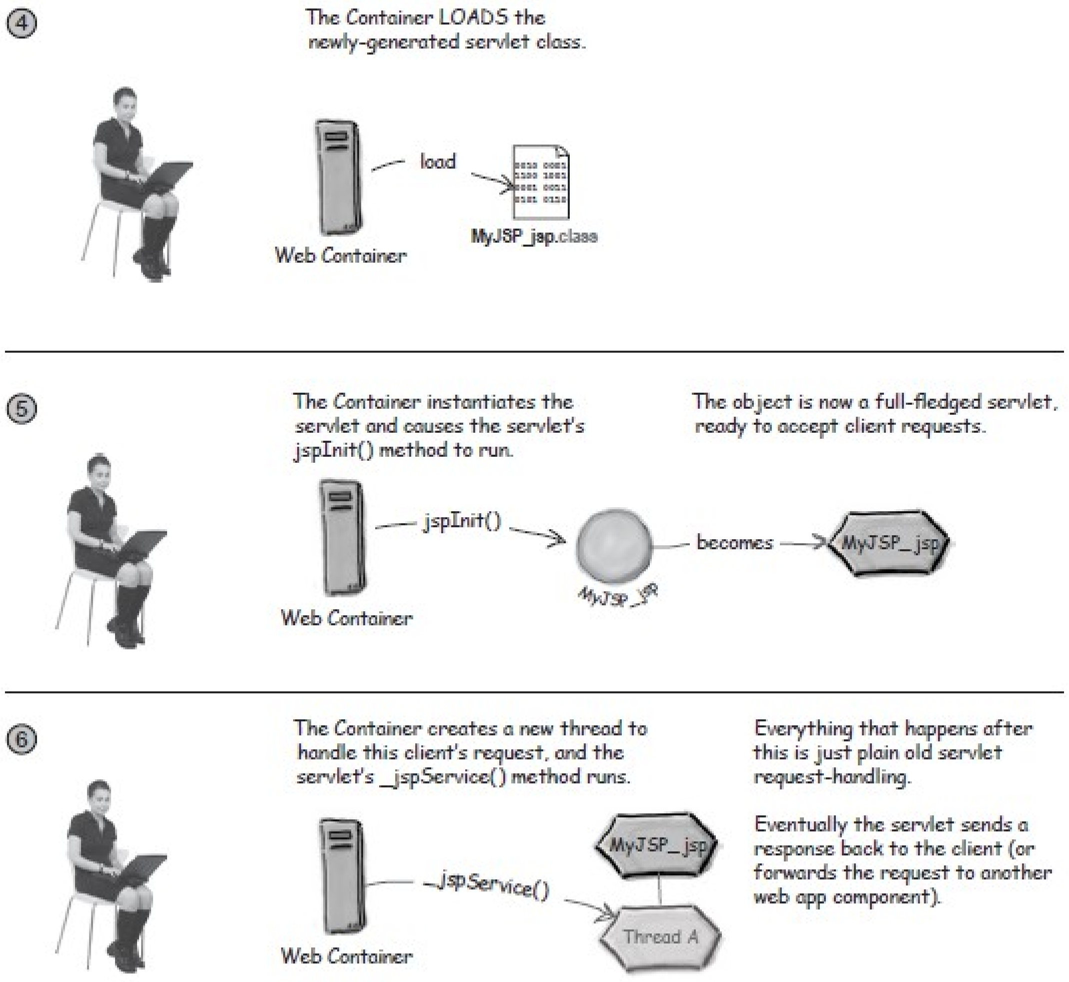
② The client hits a link that asks for the .jsp.

The Container tries to TRANSLATE the .jsp into .java source code for a servlet class.

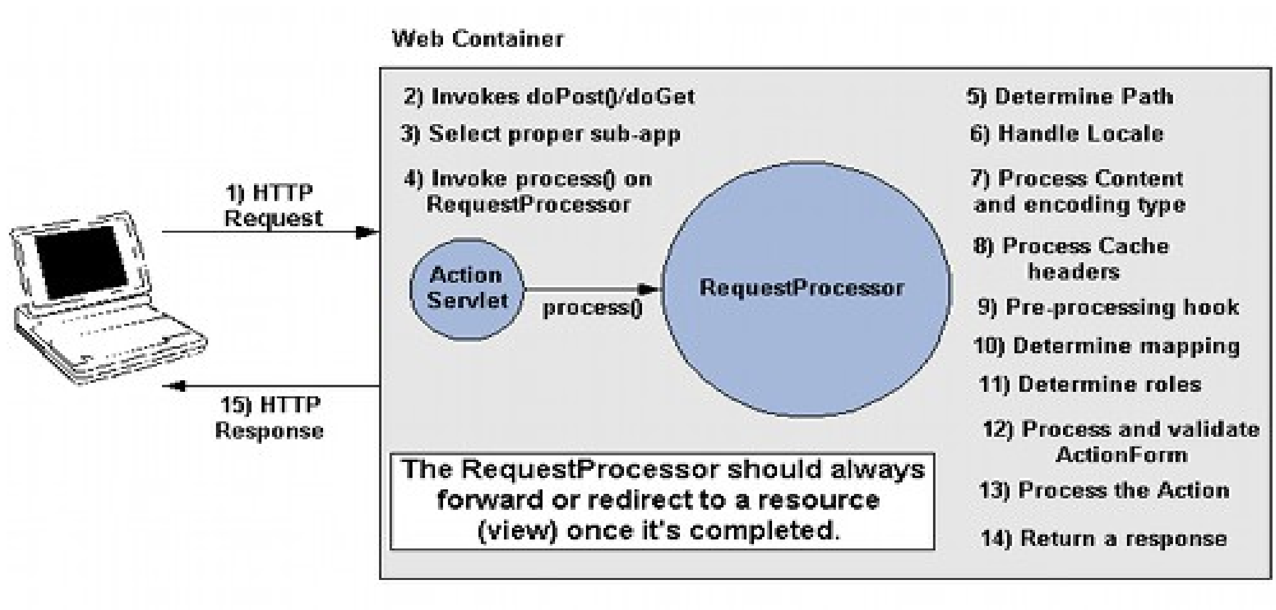
JSP syntax errors are caught in this phase.

③ The Container tries to COMPILE the servlet .java source into a .class file.

Java language/syntax errors are caught here.


4
The Container LOADS the newly-generated servlet class.
load
Web Container
MyJSP_jsp.class
5
The Container instantiates the servlet and causes the servlet's jspInit() method to run.
The object is now a full-fledged servlet, ready to accept client requests.
jspInit()
becomes
MyJSP_jsp
Web Container
MyJSP_jsp
6
The Container creates a new thread to handle this client's request, and the servlet's _jspService() method runs.
Everything that happens after this is just plain old servlet request-handling.
Eventually the servlet sends a response back to the client (or forwards the request to another web app component).
MyJSP_jsp
_jspService()
Thread A
Web Container

Life cycle of struts 1.x



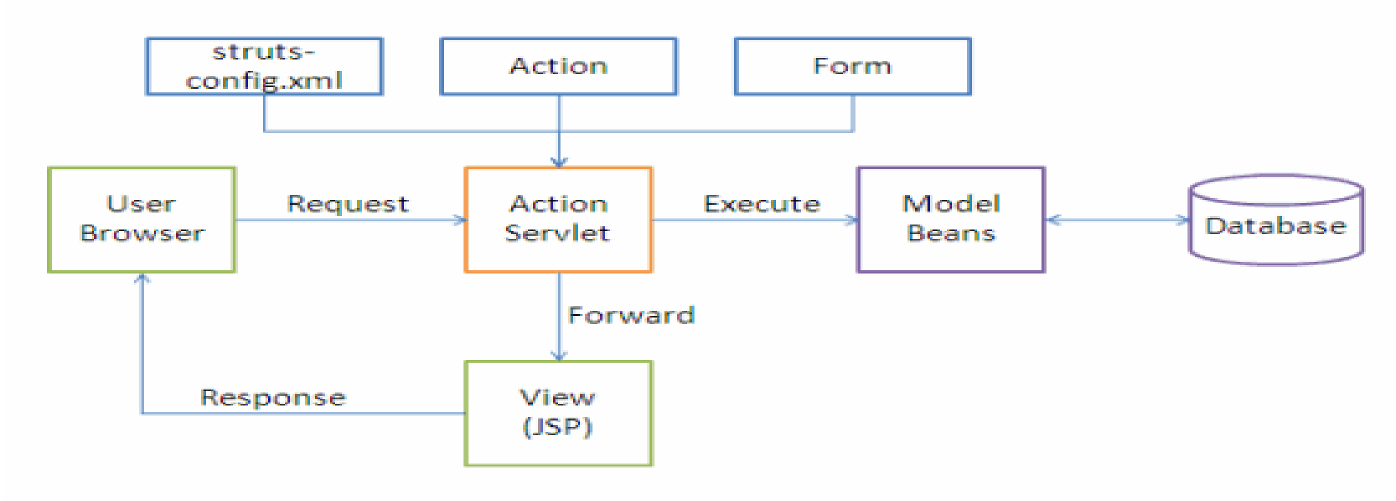
The following events happen when the Client browser issues an HTTP request.

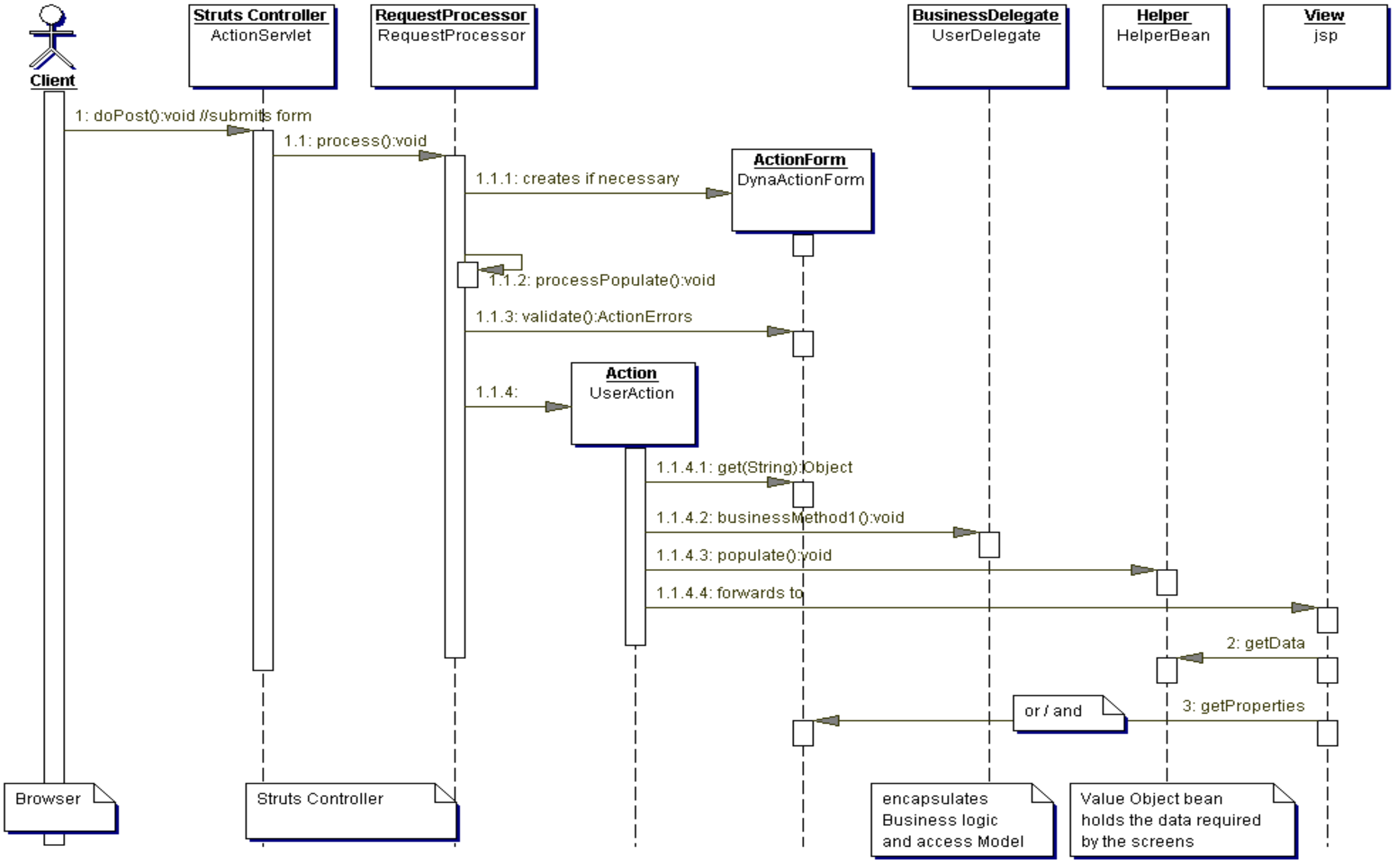
- The *ActionServlet* receives the request.

- The *struts-config.xml* file contains the details regarding the *Actions*, *ActionForms*, *ActionMappings* and *ActionForwards*.
- During the startup the *ActionServelet* reads the *struts-config.xml* file and creates a database of configuration objects. Later while processing the request the *ActionServlet* makes decision by refering to this object.

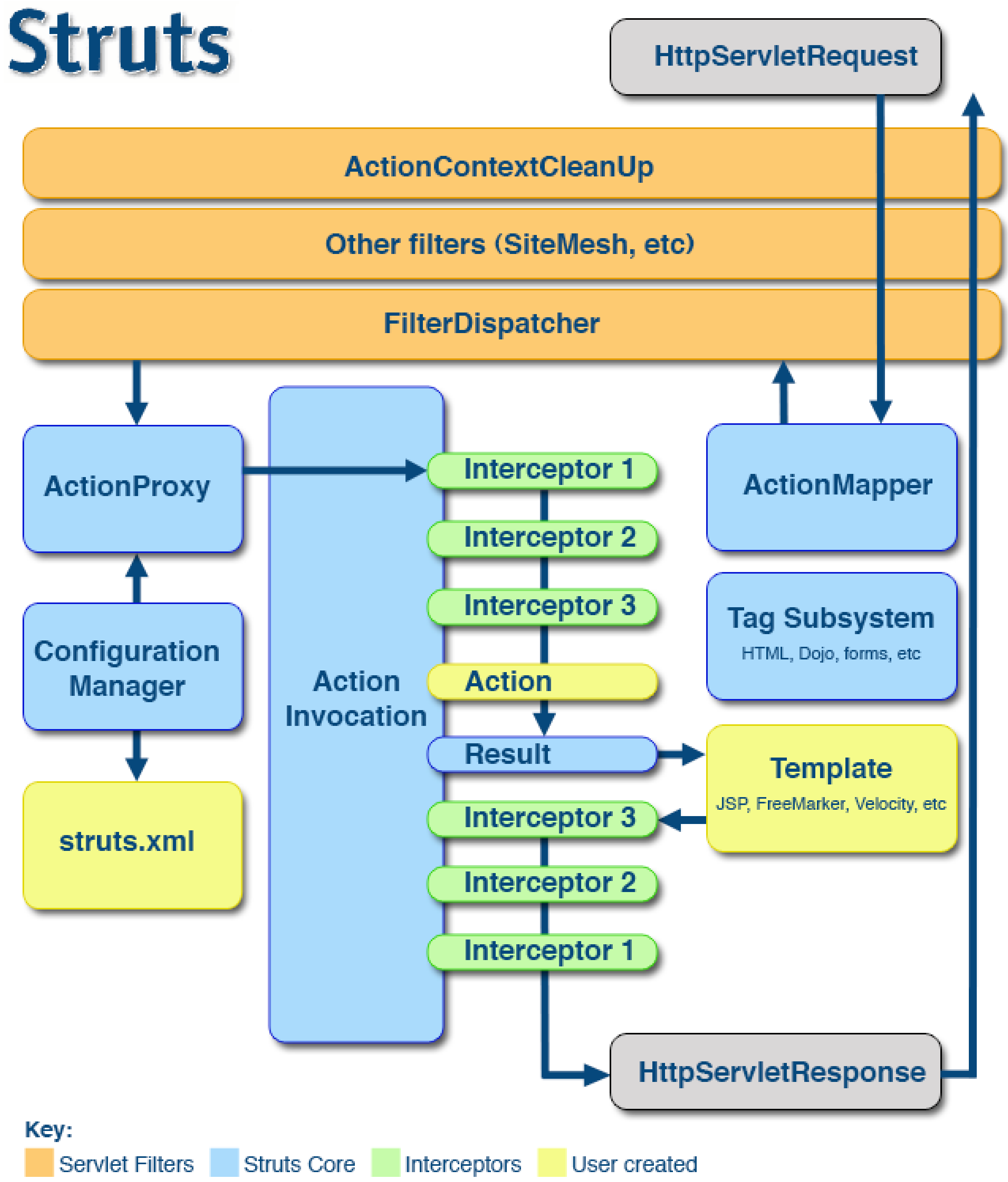
When the *ActionServlet* receives the request it does the following tasks.

- Bundles all the request values into a JavaBean class which extends Struts *ActionForm* class.
- Decides which action class to invoke to process the request.
- Validate the data entered by the user.
- The action class process the request with the help of the model component. The model interacts with the database and process the request.
- After completing the request processing the *Action* class returns an *ActionForward* to the controller.
- Based on the *ActionForward* the controller will invoke the appropriate view.





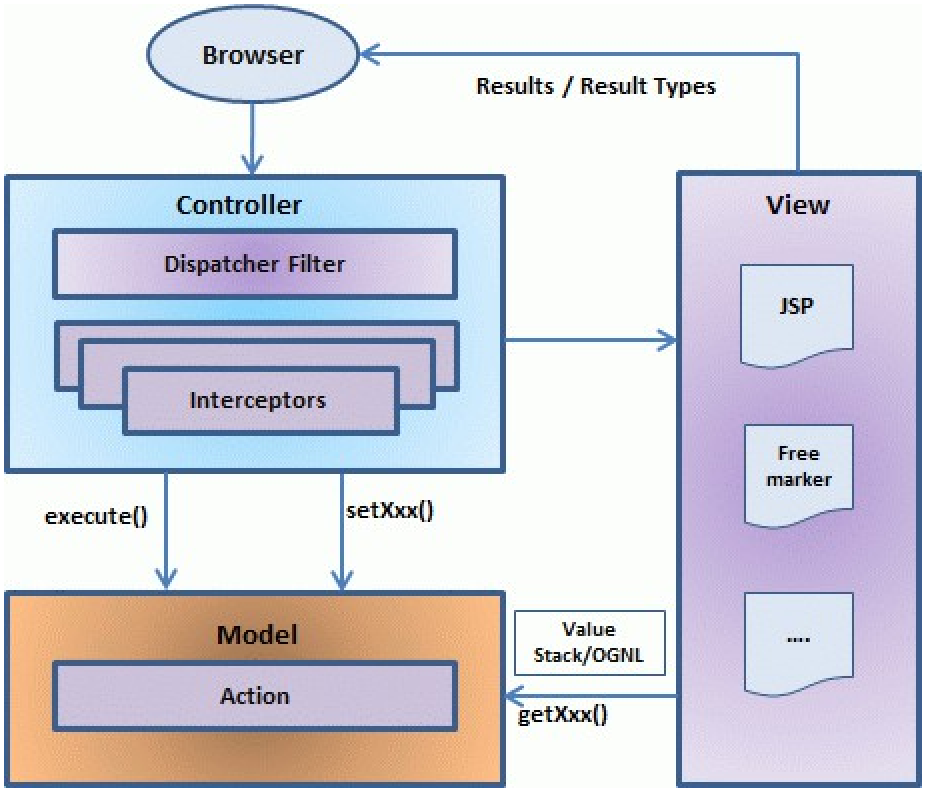
Struts2 Life Cycle

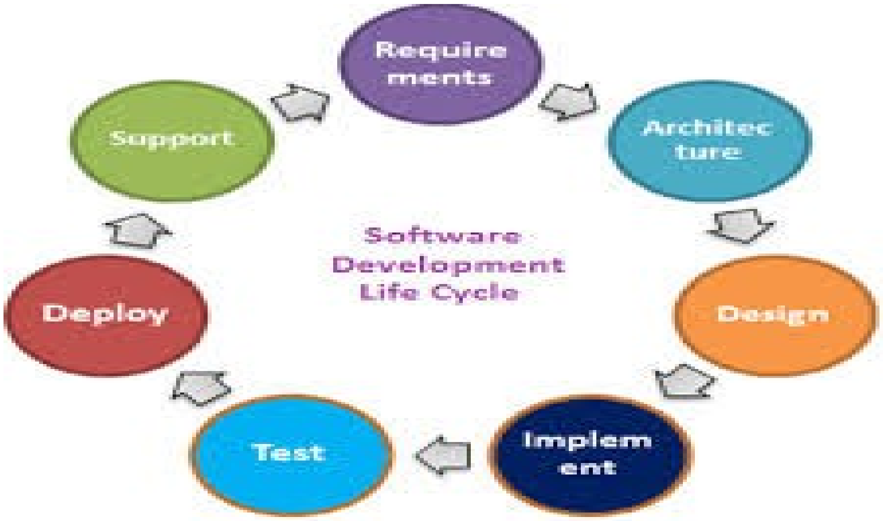


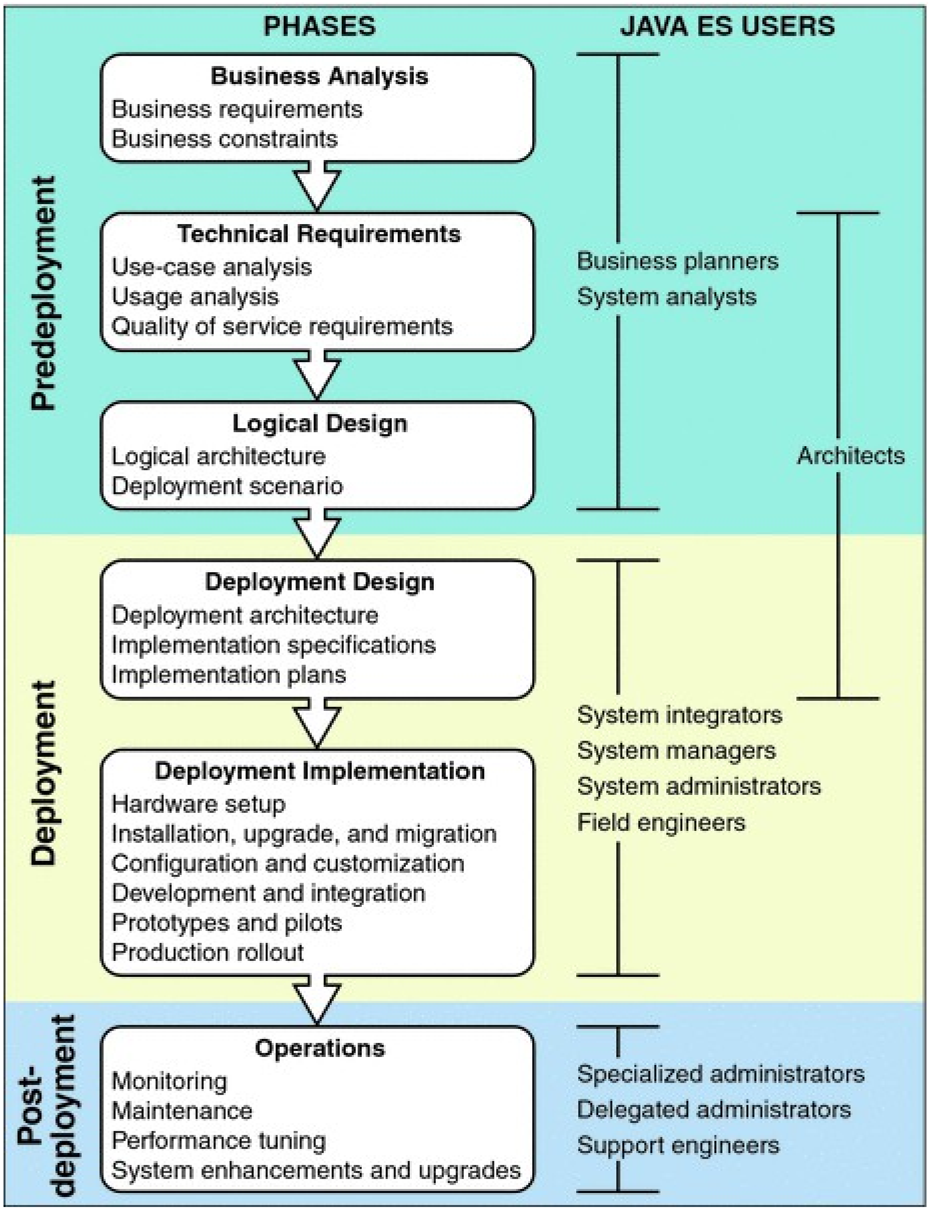
Request Life Cycle in Struts2 Framework:

- Lifecycle begins with the request sent from User. Once this request reaches the ServetContainer container passes the request to Filter according to configuration defined in web.xml. If request need to forwarded to Struts2 Framework request will go to Filter(StrutsPrepareAndExecuteFilter / StrutsPrepareFilter / StrutsExecuteFilter).
- Filter then called which consults the ActionMapper to determine whether an Action should be invoked or not.
- If ActionMapper finds an Action to be invoked, the FilterDispatcher delegates control to ActionProxy.
- ActionProxy reads the configuration file such as struts.xml. ActionProxy creates an instance of ActionInvocation class and delegates the control.
- ActionInvocation is responsible for command pattern implementation. It invokes the Interceptors one by one (if required) and

then invoke the Action.
•Once the Action returns, the ActionInvocation is responsible for looking up the proper result associated with the Action result code mapped in struts.xml.
•The result is then executed, which often (but not always, as is the case for Action Chaining) involves a template written in JSP or FreeMarker to be rendered. While rendering, templates can use the Struts Tags provided by the framework. Some of those components will work with the ActionMapper to render proper URLs for additional requests.
•The Interceptors are executed again in reverse order and the response is returned to the Filter. And the result is then sent to the servlet container which in turns sends it back to client.
•If the ActionContextCleanUp filter is present, the FilterDispatcher will not clean up the ThreadLocal ActionContext. If the ActionContextCleanUp filter is not present, the FilterDispatcher will cleanup all ThreadLocals.







Spring MVC life cycle.

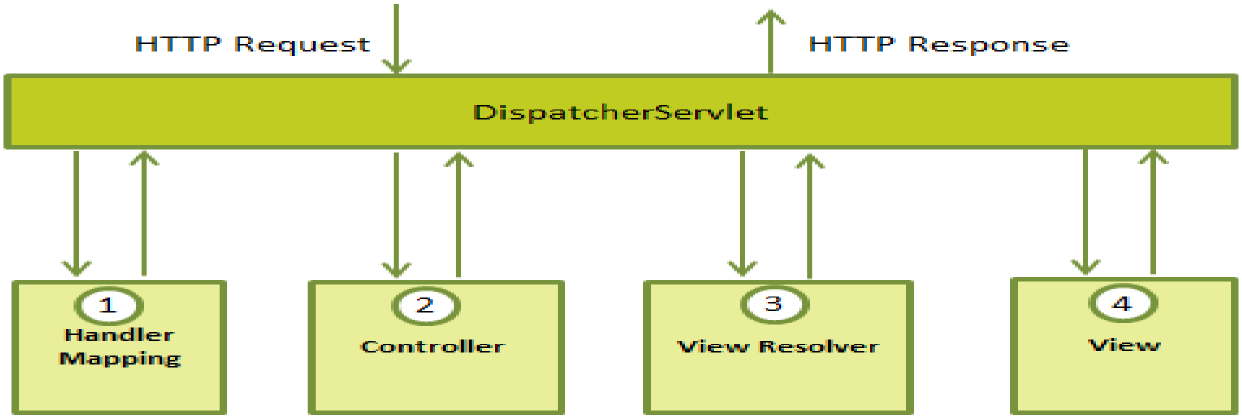
Following is the sequence of events corresponding to an incoming HTTP request to DispatcherServlet:

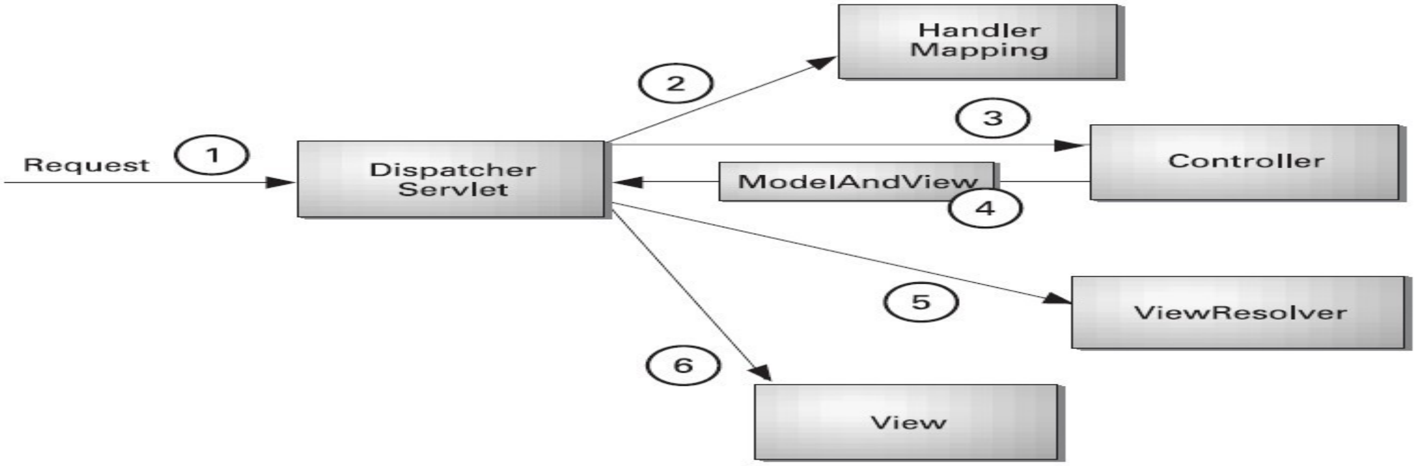
1.After receiving an HTTP request, DispatcherServlet consults the HandlerMapping to call the appropriate Controller.

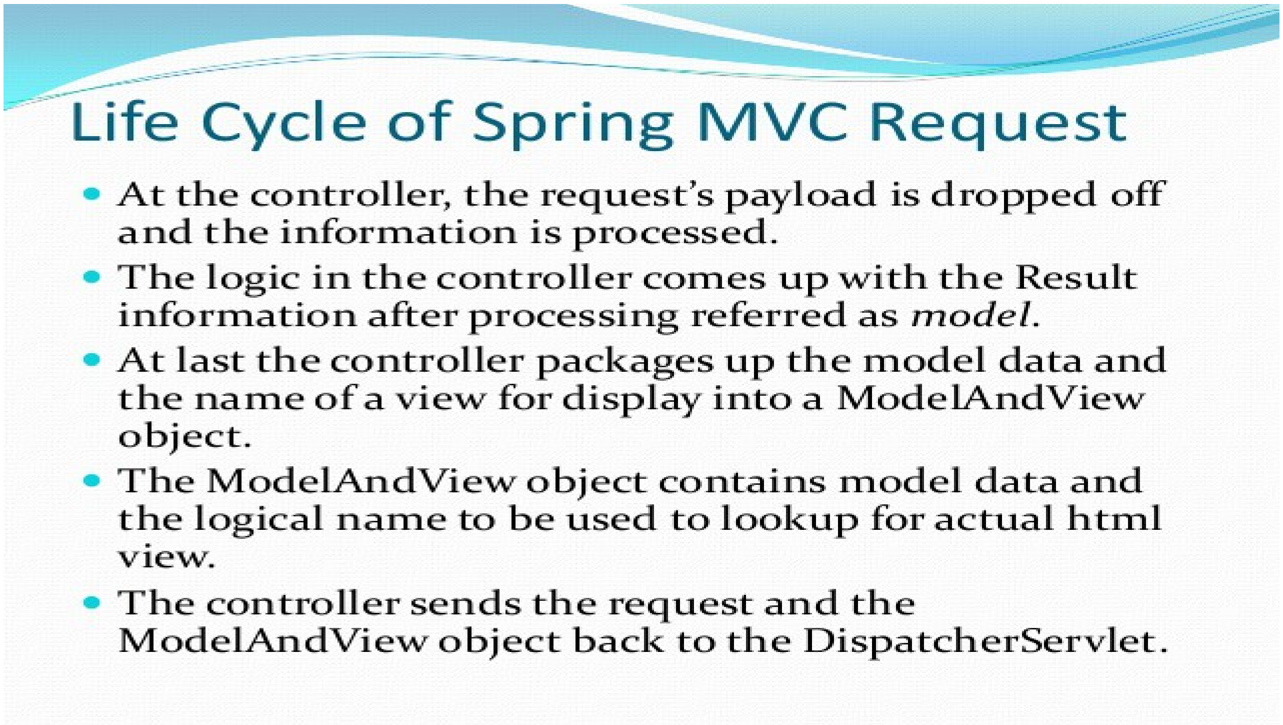
2.The Controller takes the request and calls the appropriate service methods based on used GET or POST method. The service method will set model data based on defined business logic and returns view name to the DispatcherServlet.

3.The DispatcherServlet will take help from ViewResolver to pickup the defined view for the request.

4.Once view is finalized, The DispatcherServlet passes the model data to the view which is finally rendered on the browser.

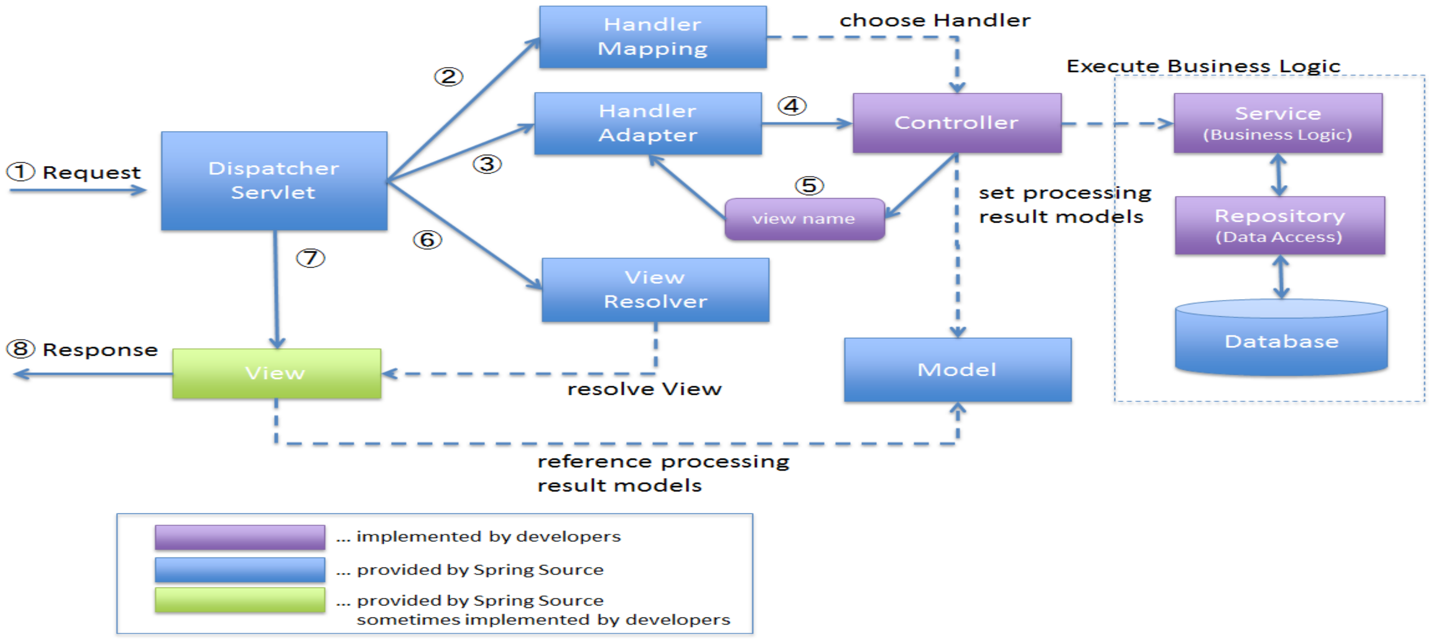






Life Cycle of Spring MVC Request

- At the controller, the request's payload is dropped off and the information is processed.
- The logic in the controller comes up with the Result information after processing referred as *model*.
- At last the controller packages up the model data and the name of a view for display into a ModelAndView object.
- The ModelAndView object contains model data and the logical name to be used to lookup for actual html view.
- The controller sends the request and the ModelAndView object back to the DispatcherServlet.



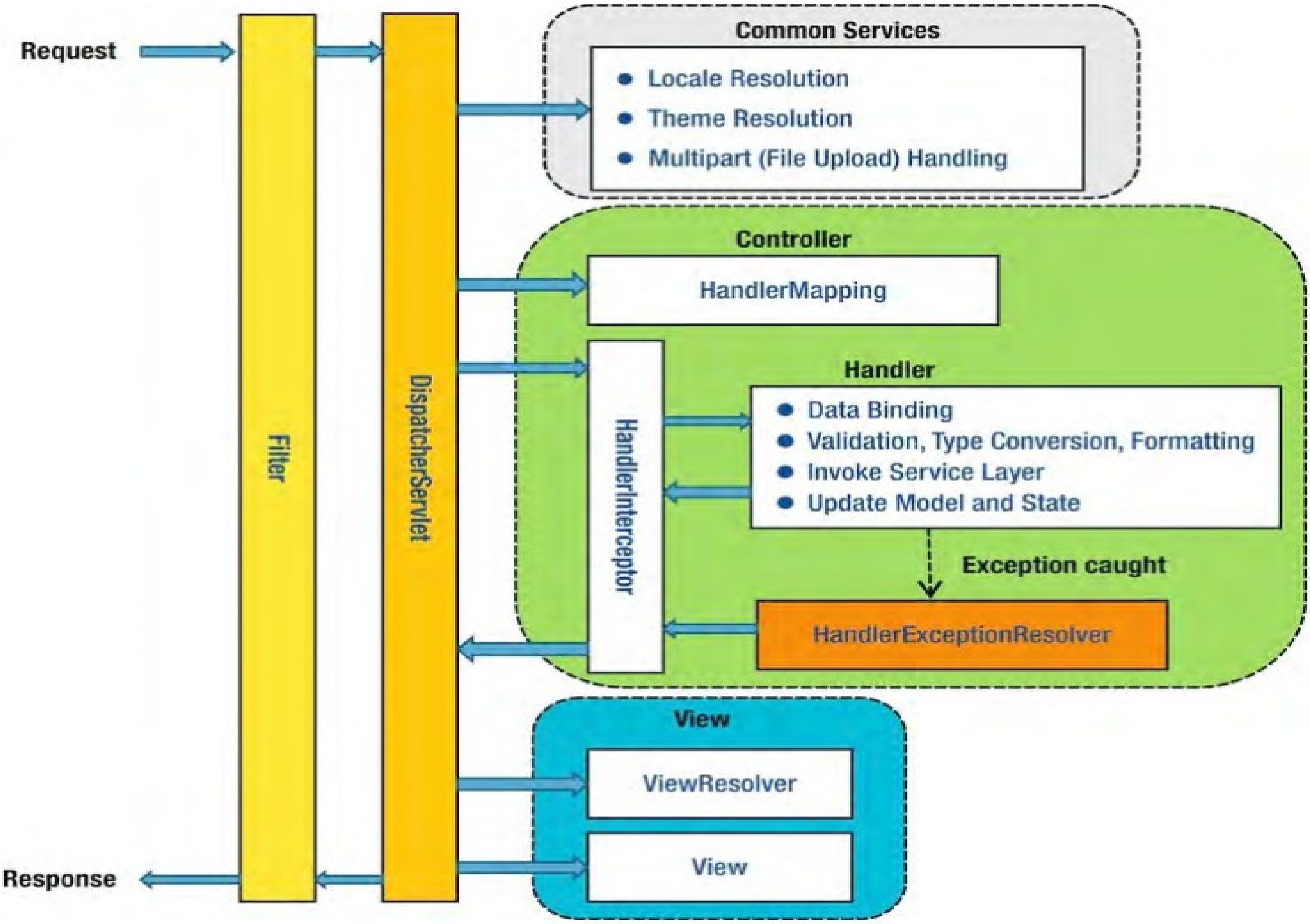


Figure 17-3. Spring MVC request life cycle