

Table Of Contents

Lecture 5

- Spring Introduction
- Build Tools Overview
- Environment Setup
- Forms
- Intro to main project

Lecture 6

- Db setup
- Spring Initializr
- Spring MVC Introduction

Lecture 7

- Spring MVC continued
- Hibernate/ORM in general
- Exception Handling

Table Of Contents

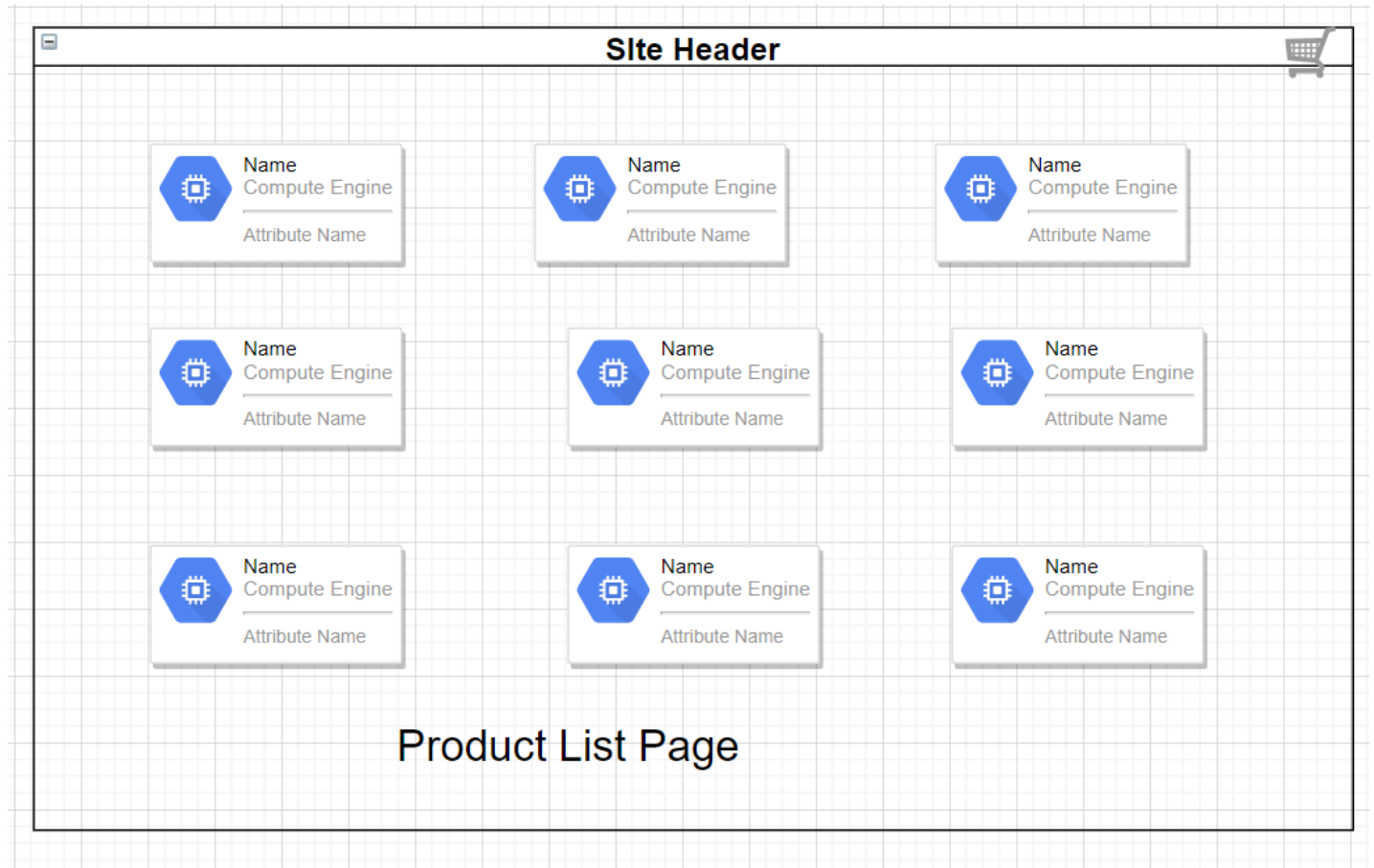
Lecture 8

- Template engines
- Todo Lab

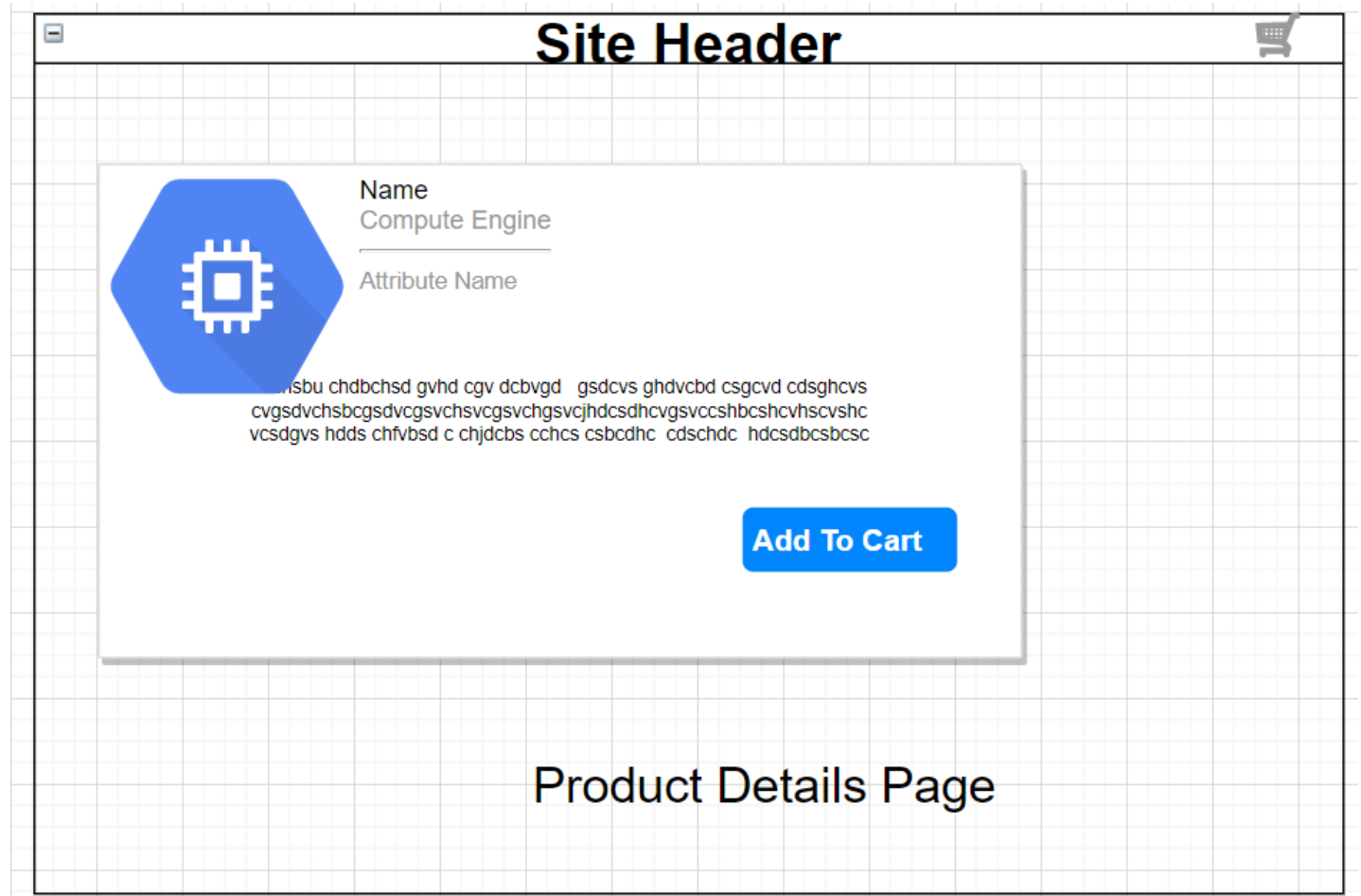
Lecture 9

- Spring Security
- Login Lab

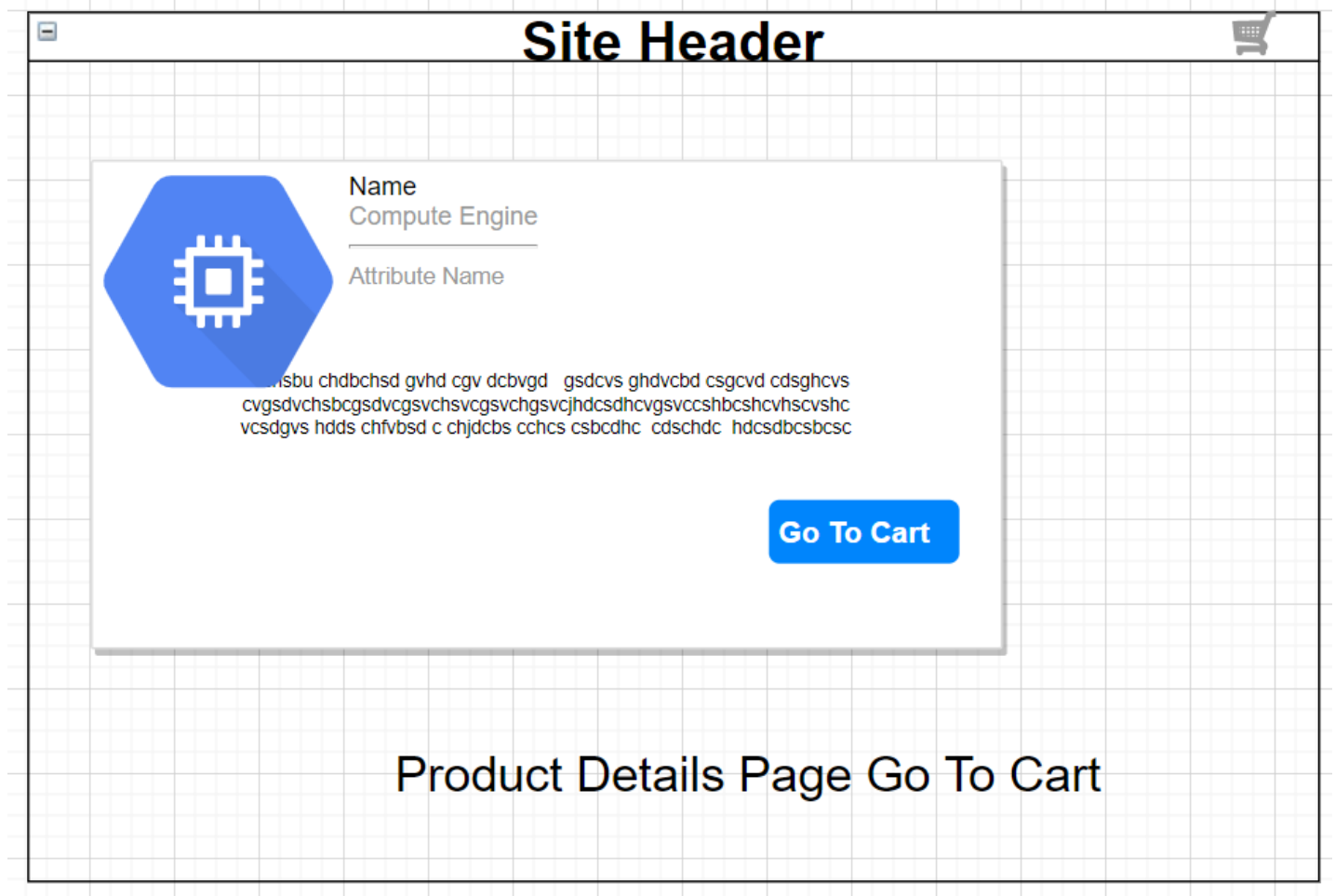
Main project - App Mockups



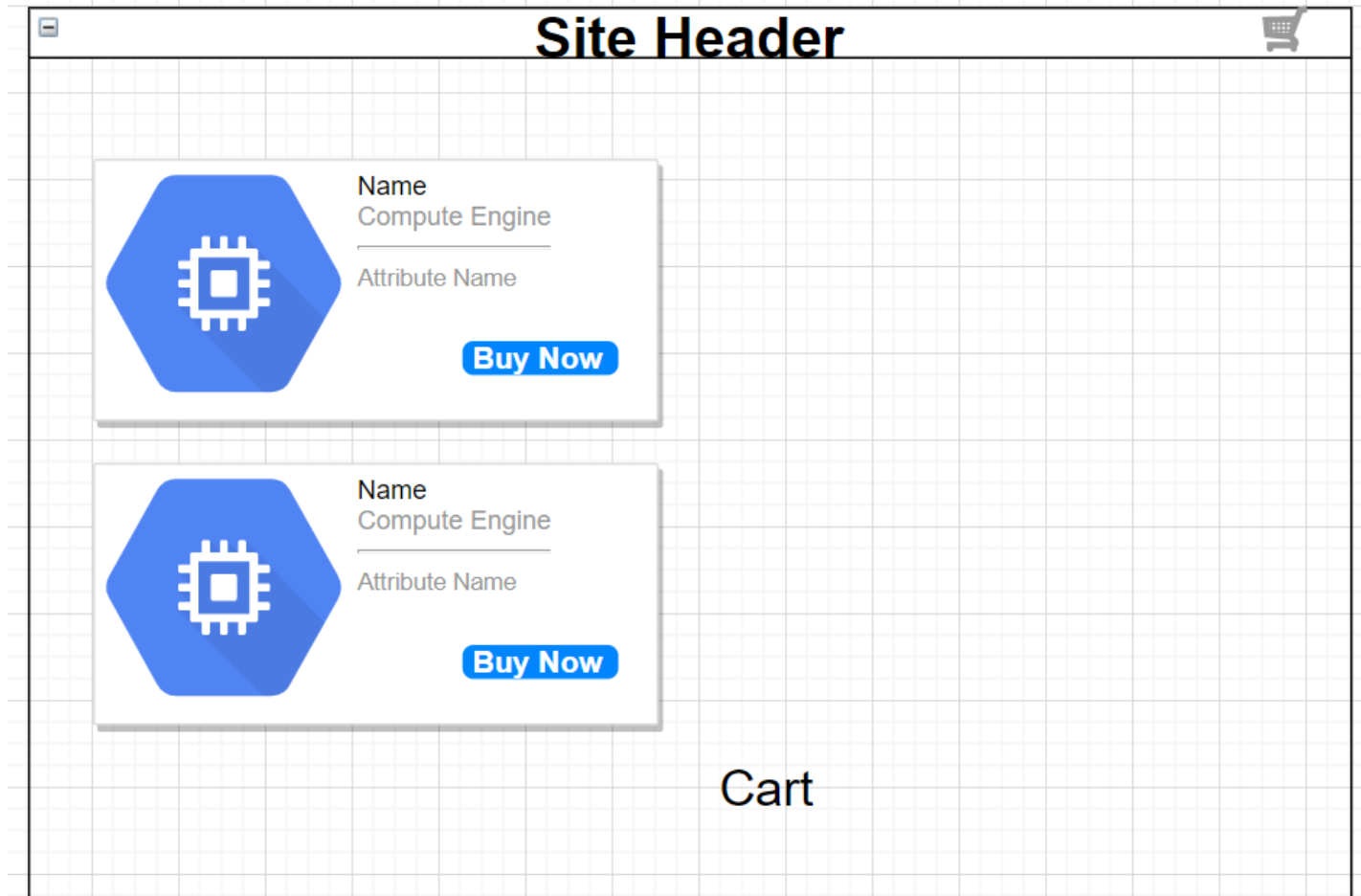
App Mockups



App Mockups




App Mockups



App Mockups

Site Header



Name

Compute Engine

Attribute Name

Name

Mobile Number

Address

1

2

3

4

Make Payment



What is Spring Framework?

Spring is the most popular application development framework for enterprise Java.

Spring supports all major application servers and JEE standards.

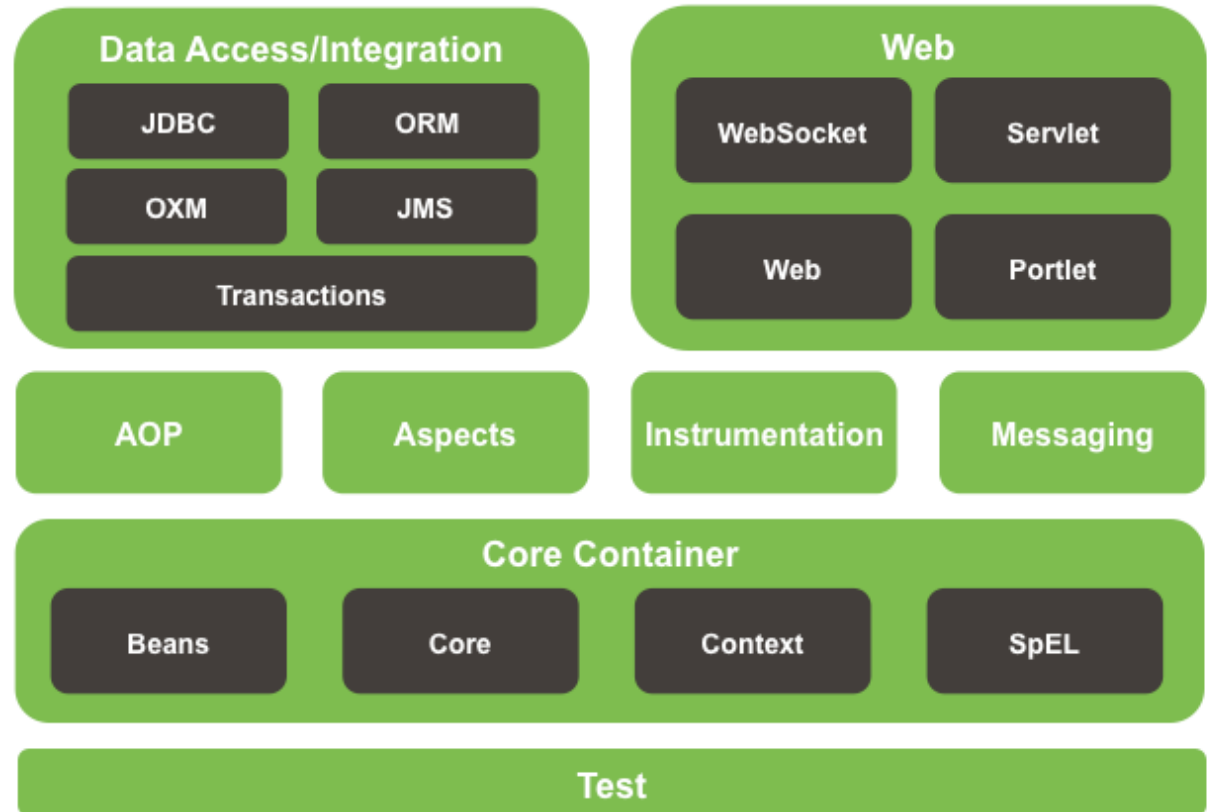
Spring handles the infrastructure so you can focus on your application.

Spring Framework provides a light-weight solution to develop maintainable and reusable enterprise applications

Spring Architecture



Spring Framework Runtime



IoC

Inversion of Control



In software engineering, inversion of control (IoC) describes a design in which custom written portions of a computer program receive the flow of control from a generic, reusable library



The Inversion of Control (IoC) is a general concept, and it can be expressed in many ways and dependency Injection is merely one concrete example of Inversion of Control.

DI

Dependency Injection



The technology that defines Spring (Heart of Spring).



Dependency Injection helps us to keep our classes as independent as possible.



Dependency injection is a pattern where the container passes objects by name to other objects, via either constructors, properties, or factory methods.

IoC Container



The Spring container (IoC Container) is at the core of the Spring Framework.



The container will create the objects, wire them together, configure them, and manage their complete lifecycle from creation till destruction.



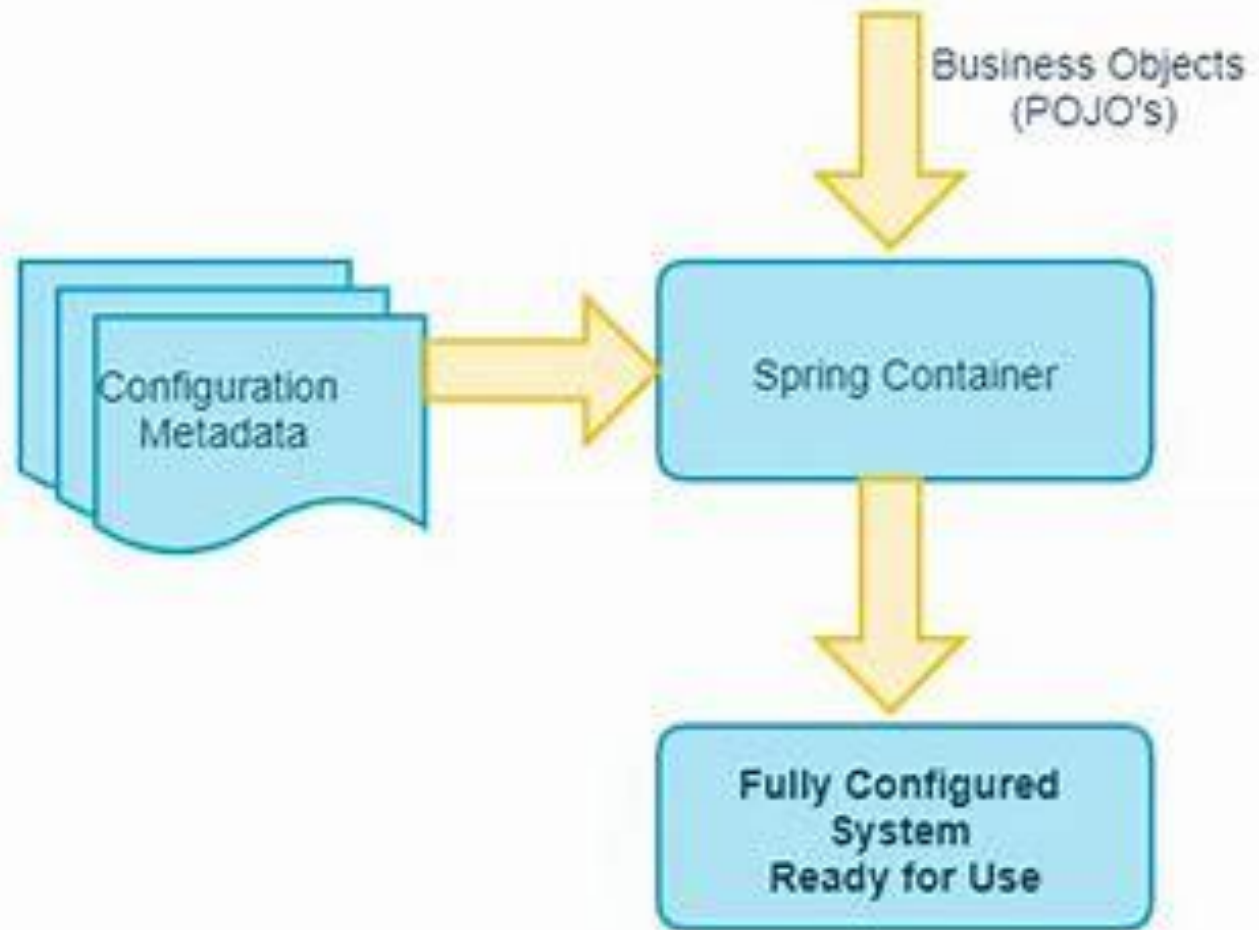
The container gets its instructions on what objects to instantiate, configure, and assemble by reading configuration metadata provided.



The configuration metadata can be represented either by:-

XML
Java annotations
Java code.

Spring IoC Container





Context

- Building a software project typically consists of such tasks as downloading dependencies, putting additional jars on a classpath, compiling source code into binary code, running tests, packaging compiled code into deployable artifacts such as JAR, WAR, and ZIP files, and deploying these artifacts to an application server or repository.
- [Apache Maven](#) automates these tasks, minimizing the risk of humans making errors while building the software manually and separating the work of compiling and packaging our code from that of code construction.

Why use Maven?

- **simple project setup that follows best practices:** Maven tries to avoid as much configuration as possible, by supplying project templates (named *archetypes*)
- **dependency management:** it includes automatic updating, downloading and validating the compatibility, as well as reporting the dependency closures (known also as transitive dependencies)

Why use Maven?

- **central repository system:** project dependencies can be loaded from the local file system or public repositories, such as [Maven Central](#)
- **isolation between project dependencies and plugins:** with Maven, project dependencies are retrieved from the *dependency repositories* while any plugin's dependencies are retrieved from the *plugin repositories*, resulting in fewer conflicts when plugins start to download additional dependencies

Project Object Model - POM

- The configuration of a Maven project is done via a *Project Object Model (POM)*, represented by a *pom.xml* file. The *POM* describes the project, manages dependencies, and configures plugins for building the software.
- The *POM* also defines the relationships among modules of multi-module projects.

Structure of pom.xml

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>org.baeldung</groupId>
  <artifactId>org.baeldung</artifactId>
  <packaging>jar</packaging>
  <version>1.0-SNAPSHOT</version>
  <name>org.baeldung</name>
  <url>http://maven.apache.org</url>
  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>4.12</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
  <build>
    <plugins>
      <plugin>
        //...
      </plugin>
    </plugins>
  </build>
</project>
```

Maven Build Lifecycles Phases

- *validate* – checks the correctness of the project
- *compile* – compiles the provided source code into binary artifacts
- *test* – executes unit tests
- *package* – packages compiled code into an archive file
- *integration-test* – executes additional tests, which require the packaging
- *verify* – checks if the package is valid
- *install* – installs the package file into the local Maven repository
- *deploy* – deploys the package file to a remote server or repository

Plugins and Goals

- A Maven *plugin* is a collection of one or more *goals*. Goals are executed in phases, which helps to determine the order in which the *goals* are executed.
- To gain a better understanding of which *goals* are run in which phases by default, take a look at the [default Maven lifecycle bindings](#).
- To go through any one of the above phases, we just have to call one command:

```
mvn <phase>
```

Maven Installation

- **Prerequisites**

- **Java** - Maven is written in Java. So, to run Maven, we need a system that has Java installed and configured properly.
- Once Java is installed, we need to ensure that the commands from the Java JDK are in our PATH environment variable. Running, for example ``java -version`` must display the right version number.

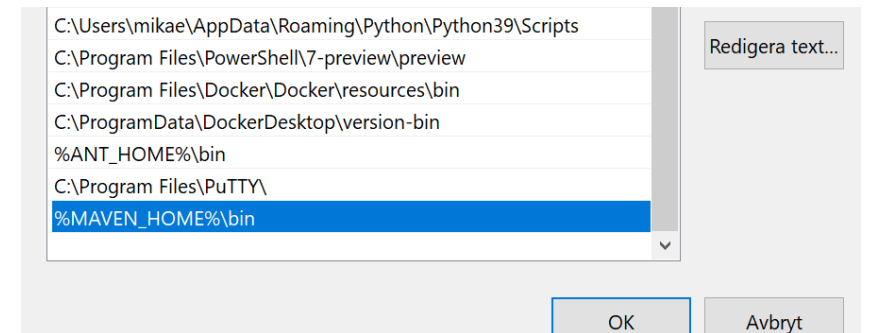
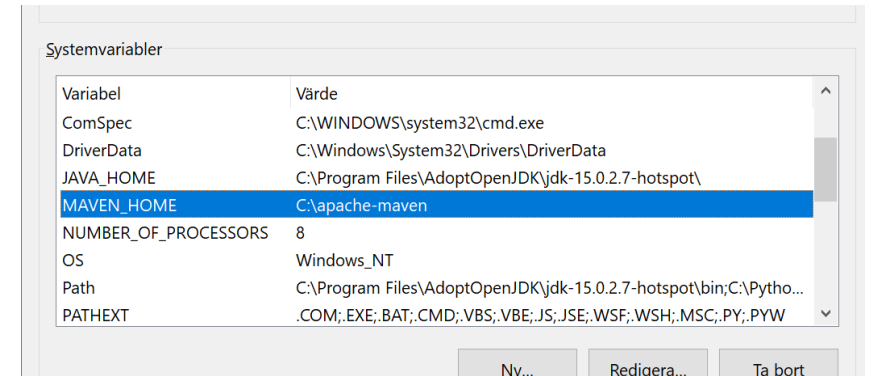
```
java -version
```

Maven Installation

- To install Maven on windows, we head over to the [Apache Maven site](#) to download the latest version and select the Maven zip file, for example, *apache-maven-3.3.9-bin.zip*.
- Then we unzip it to the folder where we want Maven to live.

Adding Maven to the Environment Path

- We add *MAVEN_HOME* variable to the Windows environment using system properties and point it to our Maven folder.
- Then we update the *PATH* variable by appending the Maven bin folder — *%MAVEN_HOME%\bin* — so that we can run the Maven command everywhere.



Verify Installation

- To verify it, we run: “mvn -version” in the command prompt. It should display the Maven version, the java version, and the operating system information. That's it. We've set up Maven on our Windows system.

```
mvn -version
```

Lab1 - Your First Maven Project

- In order to build a simple Java project, let's run the following command:
- `mvn archetype:generate -DgroupId=org.baeldung -DartifactId=org.baeldung.java -DarchetypeArtifactId=maven-archetype-quickstart -DinteractiveMode=false`

```
mvn archetype:generate
  -DgroupId=org.baeldung
  -DartifactId=org.baeldung.java
  -DarchetypeArtifactId=maven-archetype-quickstart
  -DinteractiveMode=false
```

Lab1 - Your First Maven Project

- The ***groupId*** is a parameter indicating the group or individual that created a project, which is often a reversed company domain name.
- The ***artifactId*** is the base package name used in the project, and we use the standard *archetype*.
- Since we didn't specify the version and the packaging type, these will be set to default values — the version will be set to *1.0-SNAPSHOT*, and the packaging will be set to *jar*.
- If you don't know which parameters to provide, you can always specify *interactiveMode=true*, so that Maven asks for all the required parameters.

Lab1 – Include Compiler Plugin

- Add following to pom.xml :

```
<build>
  <sourceDirectory>src</sourceDirectory>
  <plugins>
    <plugin>
      <artifactId>maven-compiler-plugin</artifactId>
      <version>3.6.1</version>
      <configuration>
        <source>1.8</source>
        <target>1.8</target>
      </configuration>
    </plugin>
  </plugins>
</build>
```

Lab1 – pom.xml

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsi:schemaLocation">
  <modelVersion>4.0.0</modelVersion>
  <groupId>org.baeldung</groupId>
  <artifactId>org.baeldung.java</artifactId>
  <packaging>jar</packaging>
  <version>1.0-SNAPSHOT</version>
  <name>org.baeldung.java</name>
  <url>http://maven.apache.org</url>
  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>3.8.1</version>
      <scope>provided</scope>
    </dependency>
  </dependencies>
  <build>
    <sourceDirectory>src</sourceDirectory>
    <plugins>
      <plugin>
        <artifactId>maven-compiler-plugin</artifactId>
        <version>3.6.1</version>
        <configuration>
          <source>1.8</source>
          <target>1.8</target>
        </configuration>
      </plugin>
    </plugins>
  </build>
</project>
```

Lab1 – Compiling Project

- To compile run “mvn compile”. Maven will run through all *lifecycle* phases needed by the *compile* phase to build the project's sources.

```
mvn compile
```

- On success “target” folder will be generated at root of project. This will contain the compiled classes.

Lab1 – Packaging Project

- Now let's invoke the *package* phase, which will produce the compiled archive *jar* file. Run “mvn package”.

```
mvn package
```

- On success a “.jar” file will be created inside “target” folder.

Html Forms



HTML Forms

First name:

Last name:

- An HTML form is used to collect user input. The user input is most often sent to a server for processing.

The <form> Element

- The HTML <form> element is used to create a form for user input.
- The HTML <form> element is used to create an HTML form for user input
- Next we will cover some form elements.

The <input> Element

- An <input> element can be displayed in many ways, depending on the type attribute.

Type	Description
<input type="text">	Displays a single-line text input field
<input type="radio">	Displays a radio button (for selecting one of many choices)
<input type="checkbox">	Displays a checkbox (for selecting zero or more of many choices)
<input type="submit">	Displays a submit button (for submitting the form)
<input type="button">	Displays a clickable button

The <input> Element - Text Fields

- The <input type="text"> defines a single-line input field for text input

```
<form>
  <label for="fname">First name:</label><br>
  <input type="text" id="fname" name="fname">
<br>
  <label for="lname">Last name:</label><br>
  <input type="text" id="lname" name="lname">
</form>
```

The <input> Element – Radio Buttons

- The <input type="radio"> defines a radio button.
- Radio buttons let a user select ONE of a limited number of choices.

```
<form>  
  <input type="radio" id="male" name="gender" value="male">  
  <label for="male">Male</label><br>  
  <input type="radio" id="female" name="gender" value="female">  
  <label for="female">Female</label><br>  
  <input type="radio" id="other" name="gender" value="other">  
  <label for="other">Other</label>  
</form>
```

The <input> Element – Checkboxes

- The <input type="checkbox"> defines a checkbox.
- Checkboxes let a user select ZERO or MORE options of a limited number of choices.

```
<form>
  <input type="checkbox" id="vehicle1" name="vehicle1" value="Bike">
  <label for="vehicle1"> I have a bike</label><br>
  <input type="checkbox" id="vehicle2" name="vehicle2" value="Car">
  <label for="vehicle2"> I have a car</label><br>
  <input type="checkbox" id="vehicle3" name="vehicle3" value="Boat">
  <label for="vehicle3"> I have a boat</label>
</form>
```

The <input> Element – Submit Button

- The <input type="submit"> defines a button for submitting the form data to a form-handler.
- The form-handler is typically a file on the server with a script for processing input data.
- The form-handler is specified in the form's action attribute.

```
<form action="/action_page.jsp">  
  <label for="fname">First name:</label><br>  
  <input type="text" id="fname" name="fname" value="John">  
<br>  
  <label for="lname">Last name:</label><br>  
  <input type="text" id="lname" name="lname" value="Doe">  
<br><br>  
  <input type="submit" value="Submit">  
</form>
```


The <label> Element

- The <label> tag defines a label for many form elements.
- The <label> element is useful for screen-reader users, because the screen-reader will read out loud the label when the user focus on the input element..
- The <label> element also help users who have difficulty clicking on very small regions (such as radio buttons or checkboxes) - because when the user clicks the text within the <label> element, it toggles the radio button/checkbox.
- The for attribute of the <label> tag should be equal to the id attribute of the <input> element to bind them together.

LAB2 – Create Html Form

- Create a new file with “.html” extension.
- Copy below code in the file.
 - `<!DOCTYPE html>`
 - `<html>`
 - `<body>`
 - `//wirte your here`
 - `</body>`
 - `</html>`
- Doubleclick the file to open in browser.
- Inside body tag write code to display a form as shown.

First name:

Last name:

☐ Male

☐ Female

☐ Other

☐ I have a bike

☐ I have a car

☐ I have a boat

LAB2 - Solution

```
<form action="/action_page.jsp">
  <label for="fname">First name:</label><br>
  <input type="text" id="fname" name="fname" value="John"><br>

  <label for="lname">Last name:</label><br>
  <input type="text" id="lname" name="lname" value="Doe"><br><br>

  <input type="radio" id="male" name="gender" value="male">
  <label for="male">Male</label><br>

  <input type="radio" id="female" name="gender" value="female">
  <label for="female">Female</label><br>

  <input type="radio" id="other" name="gender" value="other">
  <label for="other">Other</label><br><br>

  <input type="checkbox" id="vehicle1" name="vehicle1" value="Bike">
  <label for="vehicle1"> I have a bike</label><br>

  <input type="checkbox" id="vehicle2" name="vehicle2" value="Car">
  <label for="vehicle2"> I have a car</label><br>

  <input type="checkbox" id="vehicle3" name="vehicle3" value="Boat">
  <label for="vehicle3"> I have a boat</label><br><br>

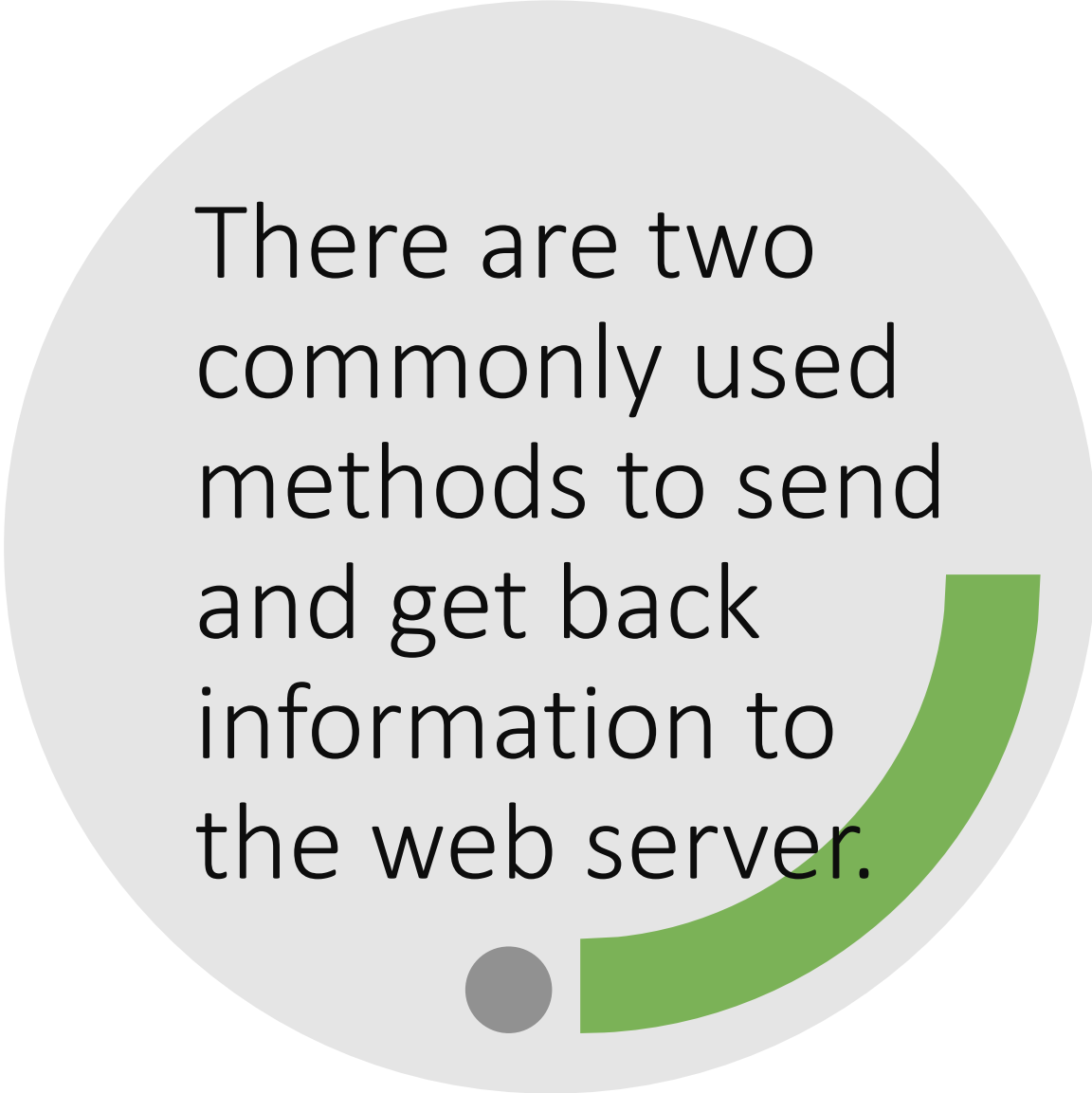
  <input type="submit" value="Submit">
</form>
```

- Forms are the common method in web processing. We need to send information to the web server and process that information.

JSP Form Processing



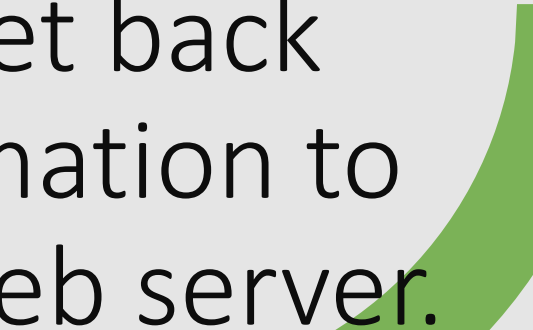
- GET Method:
 - This is the default method to pass information from browser to web server.
 - It sends the encoded information separated by ?character appended to URL page.
 - It also has a size limitation, and we can only send 1024 characters in the request.




There are two commonly used methods to send and get back information to the web server.

- POST Method:
 - Post method sends information as separate message.
 - It sends as text string in the headers in the request.

There are two commonly used methods to send and get back information to the web server.



- `getParameter()` is used to get the value of the form parameter.
- `getParameterValues()` is used to return the multiple values of the parameters.
- `getParameterNames()` is used to get the names of parameters.
- `getInputStream()` is used to read the binary data sent by the client.



JSP handles form data processing by using following methods:

Tomcat Setup

- Go to this link
<https://tomcat.apache.org/download-90.cgi>
- Download the zip.
- Extract the zip in any folder of your choice.
- There was some problems with the latest version, so we'll use an older one.

9.0.45

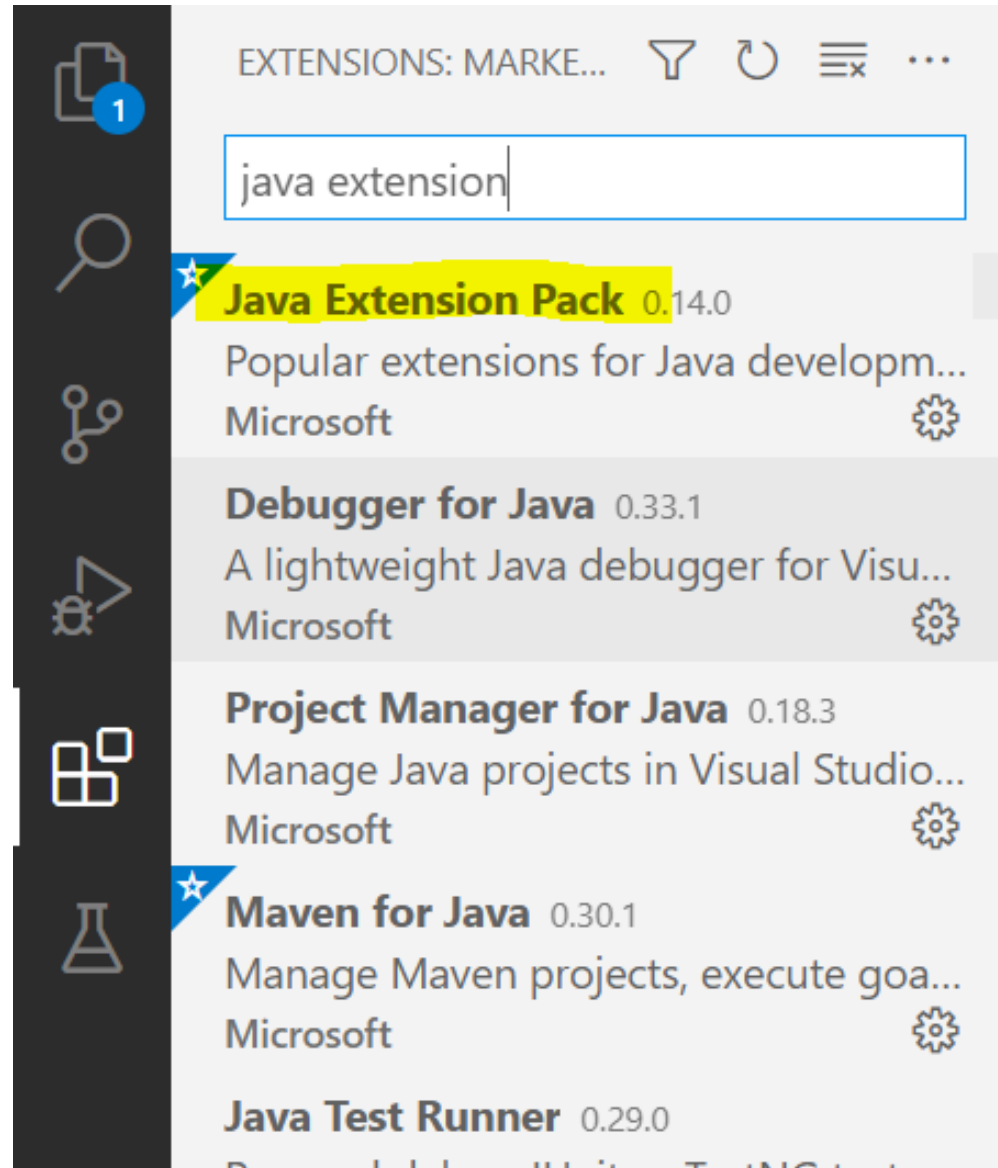
Please see the [README](#) file for packaging information. It explains what

Binary Distributions

- Core:
 - [zip](#) ([pgp](#), [sha512](#))
 - [tar.gz](#) ([pgp](#), [sha512](#))
 - [32-bit Windows zip](#) ([pgp](#), [sha512](#))
 - [64-bit Windows zip](#) ([pgp](#), [sha512](#))
 - [32-bit/64-bit Windows Service Installer](#) ([pgp](#), [sha512](#))
- Full documentation:
 - [tar.gz](#) ([pgp](#), [sha512](#))
- Deployer:
 - [zip](#) ([pgp](#), [sha512](#))

Vscode - Setup

- Open vscode.
- Click on Extensions option on left menu OR press ctrl+shift+x
- Type “java extension” in search box.
- Click on “Java Extension Pack”
- Click Install.



Steps to create web app in VS Code

- Go to command palette under “View” menu OR press “ctrl+shift+p”.
- Type “java” in search bar.
- Select “java: Create Java Project” option.
- Next select “Maven create from archetype”.
- Next select “maven-archetype-webapp”.
- Next select latest version.
- Choose groupid, default is com.example.
- Choose artifactid, default is demo.
- Choose default options in command prompt.

Lab3 – Jsp Form Processing

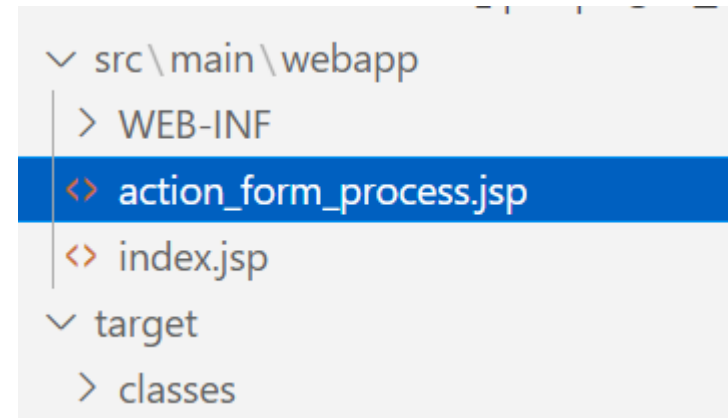
- In Vscode create a web project using maven.
- Create a “form” in index.jsp with “username” and a “Submit” button.

src > main > webapp > <> index.jsp >  html

```
1  <html>
2
3  <body>
4      <form action="action_form_process.jsp" method="GET">
5          UserName: <input type="text" name="username">
6              <br />
7              <input type="submit" value="Submit" />
8      </form>
9  </body>
10
11 </html>
```

Lab3 – Jsp Form Processing

- Create another file under “src/main/webapp”.
- Name it same as the value of “action” attribute of “form” element.
- In our case name is “action_form_processing.jsp”



Lab3 – Jsp Form Processing

- Write code to read the “username” from “request”.
- Show Welcome message on screen.

```
<%@ page language="java"
contentType="text/html; charset=UTF-8"
    pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
<head>
    <title>Form processing</title>
</head>
<body>

<h1>Form Processing</h1>

<p><b>Welcome User:</b>
    <%= request.getParameter("username")%>
</p>

</body>
</html>
```

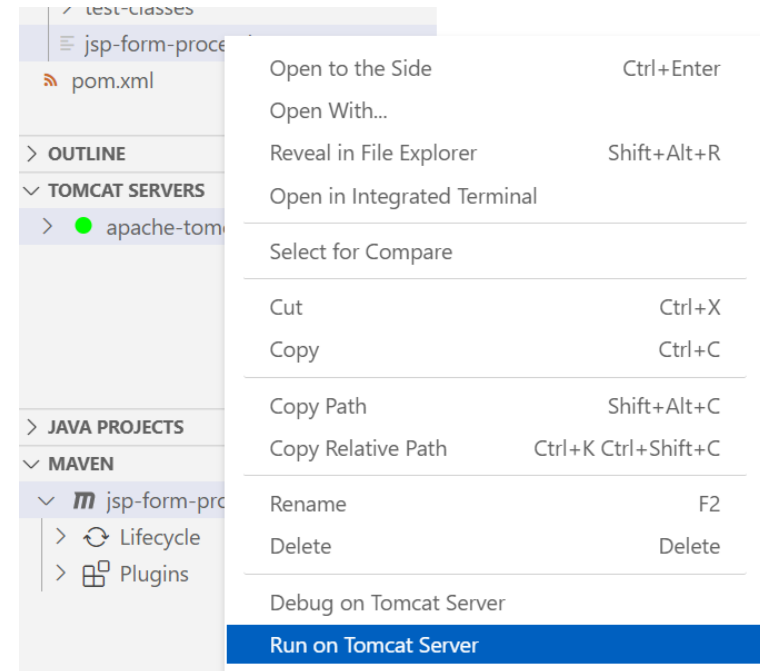
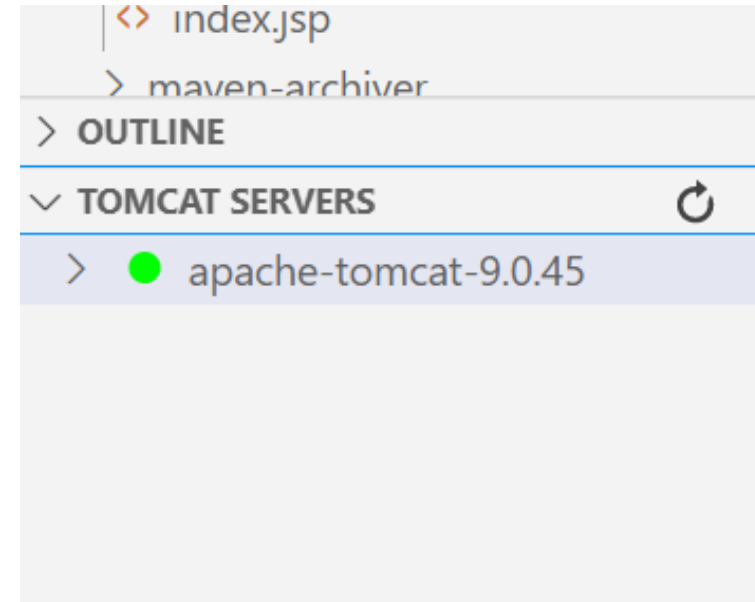
Lab3 – Jsp Form Processing

- Now package the project by running “mvn clean package” from root of your project.
- OR Run “clean” and “package” commands separately from “Maven” tab.
- A “.war” file will be created inside “target” folder.

```
✓ target
  > classes
  ✓ jsp-form-processing
    > META-INF
    > WEB-INF
    <> action_form_process.jsp
    <> index.jsp
    > maven-archiver
    > test-classes
    ≡ jsp-form-processing.war
```

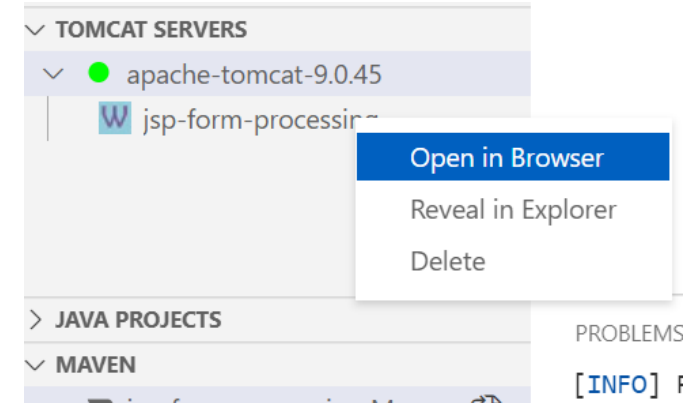
Lab3 – Jsp Form Processing

- Now click plus(+) icon under “Tomcat Servers” tab to add a server.
- Choose the location of your server.
- Now right click the “.war” file and choose “Run on Tomcat Server”.
- Make sure you don’t have any other server running. Stop the server running as service if it is running on port 8080.



Lab3 – Jsp Form Processing

- Open the app in browser by clicking “Open in browser” under tomcat tab.
- Browser window will open. With a form.
- Enter your name in text field.
- Hit Submit.
- It should show the welcome message.

A screenshot of a web browser window. The address bar shows 'localhost:8080/jsp-form-processing/'. The page content includes a label 'UserName:' followed by a text input field. Below the input field is a 'Submit' button. The browser's navigation bar shows back, forward, refresh, and home icons.

Form Processing

Welcome User: John

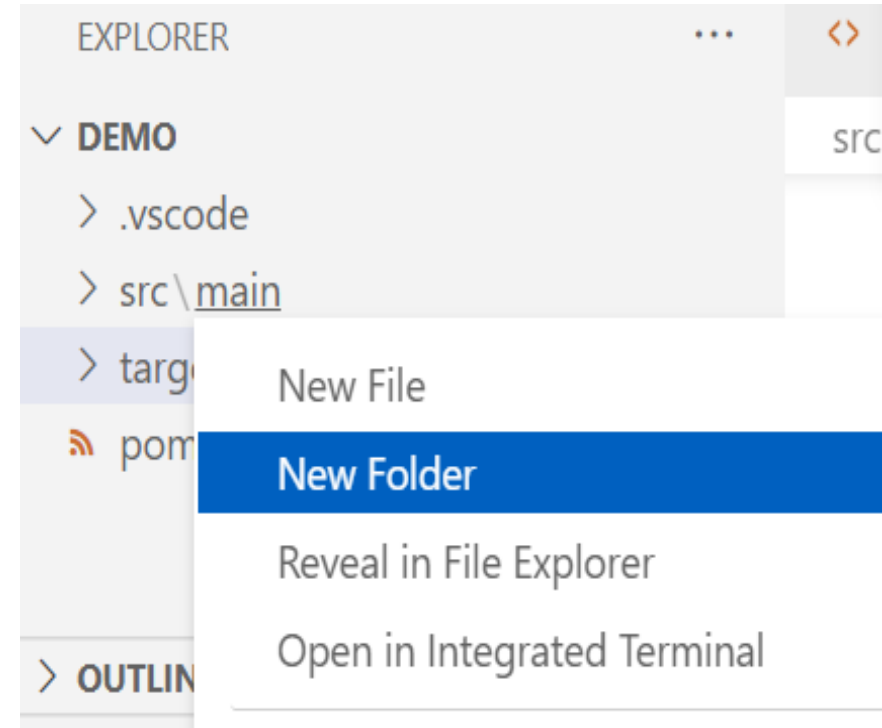
Lab4 – Servlet Form Processing.

In the last lab, we processed form inside jsp.

In this lab we will process form inside Servlet and show a welcome message.

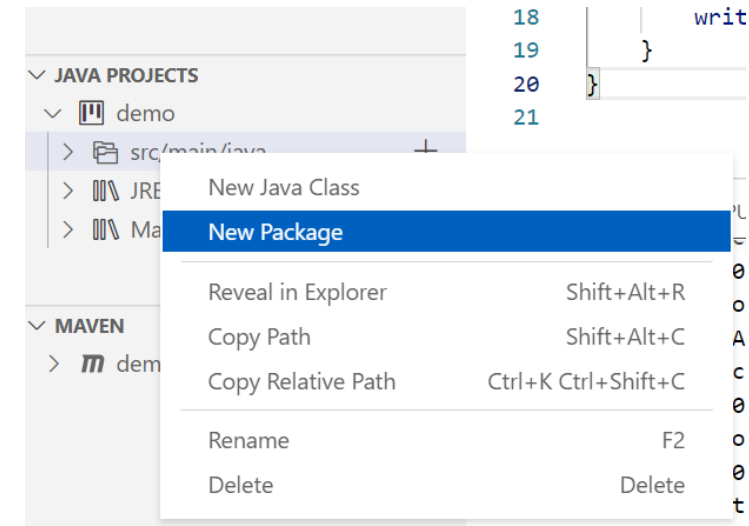
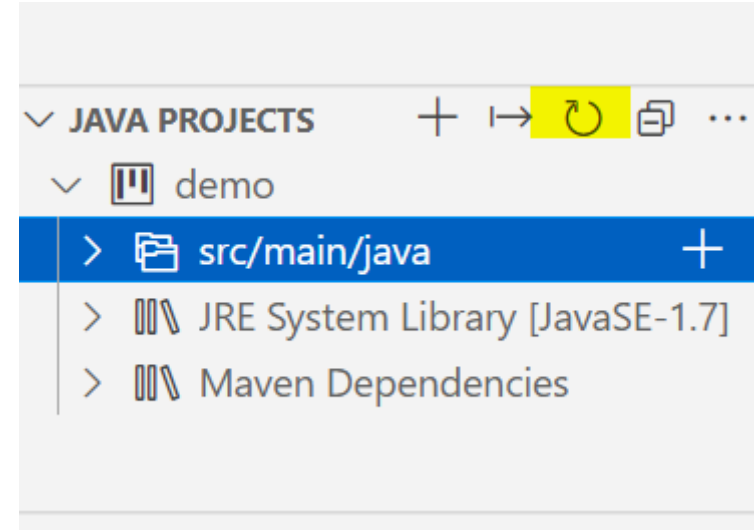
Lab4 – Servlet Form Processing.

- In VS Code create a web project using maven.
- On the left-hand corner, Go to Explorer -> Right click “src/main” -> choose “New Folder”.
- Name the folder as “java”.



Lab4 – Servlet Form Processing.

- Now under “Java Projects” tab hit refresh icon.
- You should see “src/main/java” there.
- Right click there and choose “New Package”.
- Type “group1d” as name of package. (com.example)



Lab4 – Servlet Form Processing.

- In pom.xml add “servlet” and “jsp” dependency under “dependencies” tag.
- Maven will download these dependencies for you.

```
<dependency>
  <groupId>javax.servlet</groupId>
  <artifactId>javax.servlet-api</artifactId>
  <version>3.1.0</version>
  <scope>provided</scope>
</dependency>
<dependency>
  <groupId>javax.servlet.jsp</groupId>
  <artifactId>javax.servlet.jsp-api</artifactId>
  <version>2.3.1</version>
  <scope>provided</scope>
</dependency>
```

Lab4 – Servlet Form Processing.

- In index.jsp write the following code.

```
<html>
<body>
<h2>Hello World!</h2>
<form action="/api" method="post">
    Enter your name: <input type="text" name="yourName" size="20">
    <input type="submit" value="Call Servlet" />
</form>
</body>
</html>
```

Lab4 – Servlet Form Processing.

- Create a java class under com.example package.
- This would be your servlet.
- Write following code in this file.
- Package the project and run on tomcat server.

```
@WebServlet(name = "MyServlet",urlPatterns = "/api")
public class MyServlet extends HttpServlet{
    @Override
    protected void doPost(HttpServletRequest req, HttpServletResponse resp) throws ServletException, IOException {
        String yourName = req.getParameter("yourName");
        PrintWriter writer = resp.getWriter();
        writer.println("<h1>Hello " + yourName + "</h1>");
        writer.close();
    }
}
```

Afternoon Exercise

- Make a new project with a form. In the form the user should be able to enter two numbers in a number input, select or similar. Let's call the numbers x and y .
- In the receiving page, draw a table with x columns and y rows.
- In each cell, print the x and y values.
- Extra:
 - Use CSS to make the table look like a chessboard.
 - Add headings to give the columns letters (A, B, C etc) and the rows numbers.
 - Add a text input where the user can enter a sequence like "C3". In the receiving page, the cell "C3" should then get some kind of marking, maybe through a separate class.
- Work in your groups and let's get back here at 16:00 to show what we've done.

Retrospective

- Take a few minutes in your groups to talk about what's been good today and what we could do differently.

Thanks!!

