SOFTWARE ENGINEERING

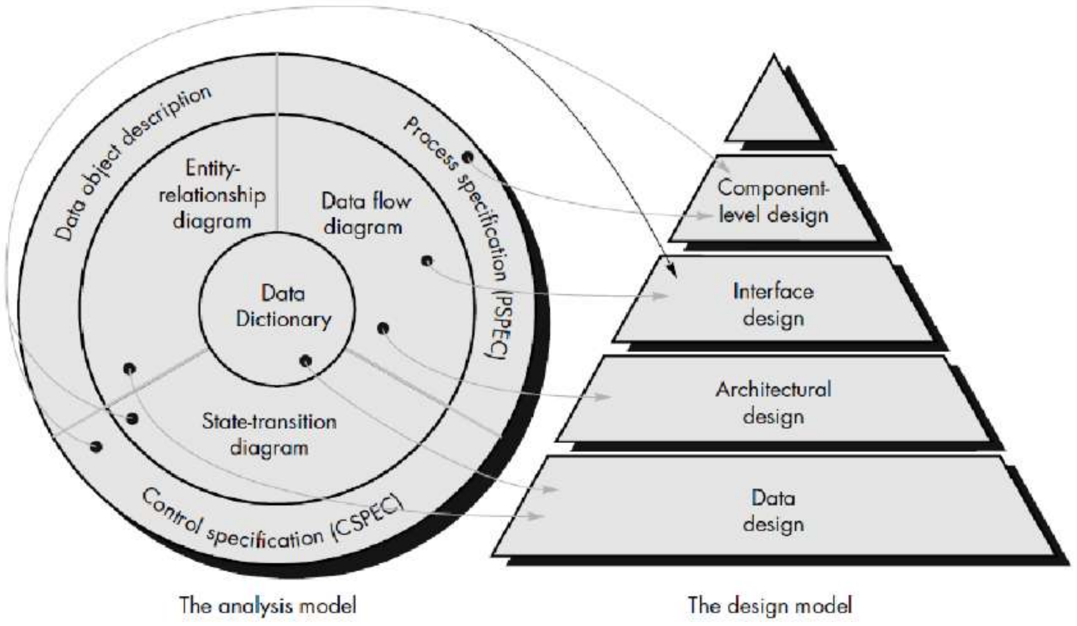
UNIT – 3

CHAPTER – 1

DESIGN ENGINEERING

I. INTRODUCTION TO DESIGN:

Design is a place where stakeholder requirements, business needs, and technical considerations all come together in the formulation of a product or system. Unlike a requirements model, design creates a representation or model of the software, which provides details about software architecture, data structures, interfaces, and components that are necessary to implement the system. Design engineering ensures the quality of the software.



Translating the analysis model into a software design

The **Data/class design** transforms class models into design class realizations and requisite data structures which are required to implement the software. Part of class design may occur in conjunction with the design of software architecture.

The **Architectural design** defines the relationship between major structural elements of the software, the architectural styles and design patterns that can be used to achieve the requirements defined for the system, and the constraints that affect the way in which architecture can be implemented.

The **Interface design** describes how the software communicates with systems that interoperate with it, and with humans who use it. An interface implies a flow of information (e.g., data and/or control) and a specific type of behavior. Therefore, usage scenarios and behavioral models provide much of the information required for interface design.

The **Component-level design** transforms structural elements of the software architecture into a procedural description of software components. Information obtained from the class-based models, flow models, and behavioral models serve as the basis for component design.

Design provides you with representations of software that can be assessed for quality. Design is the only way that you can accurately translate stakeholder's requirements into a finished software product or system. Software design serves as the foundation for all the software engineering and software support activities that follow.

II. THE DESIGN PROCESS AND QUALITY:

Software design is an iterative process through which requirements are translated into a "blueprint" for constructing the software. That is, the design is represented at a high level of abstraction—a level that can be directly traced to the specific system objective and more detailed data, functional, and behavioral requirements. As design iterations occur, subsequent refinement leads to design representations at much lower levels of abstraction.

Software Design and Quality Guidelines: Throughout the design process, the quality of the evolving design is assessed with a series of technical reviews. The three characteristics that serve as a guide for the evaluation of a good design are:

1. The design must implement all of the explicit requirements contained in the requirements model, and it must accommodate all of the implicit requirements desired by stakeholders.

2. The design must be a readable and understandable to guide those, who generate code, test and subsequently support the software.
3. The design should provide a complete picture of the software, addressing the data, functional, and behavioral domains from an implementation perspective.

Quality Guidelines. In order to evaluate the quality of a design representation, the software team must establish technical criteria for good design. Guide lines are as follows:

1. A design should exhibit an architecture that:

    (a) Has been created using recognizable architectural styles or patterns,

    (b) Is composed of components that exhibit good design characteristics (these are discussed later in this chapter), and

    (c) Can be implemented in an evolutionary fashion, thereby facilitating implementation and testing.

2. A design should be modular; that is, the software should be logically partitioned into elements or subsystems.

3. A design should contain distinct representations of data, architecture, interfaces, and components.

4. A design should lead to data structures that are appropriate for the classes to be implemented and are drawn from recognizable data patterns.

5. A design should lead to components that exhibit independent functional characteristics.

6. A design should lead to interfaces that reduce the complexity of connections between components and with the external environment.

7. A design should be derived using a repeatable method that is driven by information obtained during software requirements analysis.

8. A design should be represented using a notation that effectively communicates its meaning.

III. THE DESIGN CONCEPTS:

Design concepts has evolved over the history of software engineering. Each concept provides the software designer with a foundation from which more sophisticated design methods can be applied. A brief overview of important software design concepts that span both traditional and object-oriented software development is given below.

I. Abstraction: When you consider a modular solution to any problem, many levels of abstraction can be posed. At the *highest level of abstraction*, a solution is stated in *broad terms* using the language of the problem environment. At *lower levels* of *abstraction*, a *more detailed description* of the solution is provided. Finally, at the lowest level of abstraction, the solution is stated in a manner that can be directly implemented.

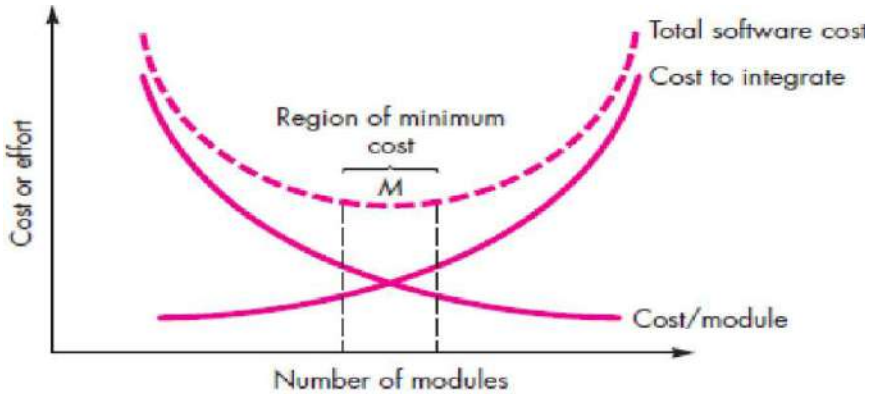
1. *A procedural abstraction* refers to a sequence of instructions that have a specific and limited function. The name of a procedural abstraction implies these functions, but specific details are suppressed.
2. *A data abstraction* is a named collection of data that describes a data object.
3. *Control abstraction* is the third form of abstraction used in software design. Like procedural and data abstraction, control abstraction implies a program control mechanism without specifying internal details

II. Refinement: Stepwise refinement is a *top-down design strategy.* A program is developed by successively refining levels of procedural detail. A hierarchy is developed by decomposing a *macroscopic statement* of function (a procedural abstraction) in a stepwise fashion until programming language statements are reached.

1. Refinement is actually a process of elaboration begins with a statement of function (or description of information) that is defined at a high level of abstraction. You then elaborate on the original statement, providing more and more detail as each successive refinement (elaboration) occurs.
2. Refinement helps you to reveal low-level details as design progresses.

III. Modularity: Modularity is the most common manifestation of separation of concerns. Software is divided into separately named and addressable components, sometimes called modules that are integrated to satisfy problem requirements.

1. It has been stated that "modularity is the single attribute of software that allows a program to be intellectually manageable". The number of control paths, span of reference, number of variables, and overall complexity would make understanding close to impossible.
2. In almost all instances, you should break the design into many modules, hoping to make understanding easier and, as a consequence, reduce the cost required to build the software.
3. If you subdivide software indefinitely the effort required to develop it will become negligibly small! Unfortunately, other forces come into play, causing this conclusion to be (sadly) invalid. The effort (cost) to develop an individual software module does decrease as the total number of modules increases.



Given the same set of requirements, more modules mean smaller individual size. However, as the number of modules grows, the effort (cost) associated with integrating the modules also grows. These characteristics lead to a total cost or effort curve shown in the figure. There is a number, M, of modules that would result in minimum development cost, but we do not have the necessary sophistication to predict M with assurance.

The curves shown in Figure 8.2 do provide useful qualitative guidance when modularity is considered. You should modularize, but care should be taken to stay in the vicinity of M. *Under modularity* or *over modularity* should be avoided.

You modularize a design (and the resulting program) so that development can be more easily planned; software increments can be defined and delivered; changes can be more easily accommodated; testing and debugging can be conducted more efficiently, and long-term maintenance can be conducted without serious side effects.

We have five criteria's that enable us to evaluate a design method with respect to its ability to define an effective modular system:

1. **Modular decomposability.** If a design method provides a systematic mechanism for decomposing the problem into sub problems, it will reduce the complexity of the overall problem, thereby achieving an effective modular solution.
2. **Modular composability.** If a design method enables existing (reusable) design components to be assembled into a new system, it will yield a modular solution that does not reinvent the wheel.
3. **Modular understandability.** If a module can be understood as a standalone unit (without reference to other modules), it will be easier to build and easier to change.
4. **Modular continuity.** If small changes to the system requirements result in changes to individual modules, rather than system wide changes, the impact of change-induced side effects will be minimized.
5. **Modular protection**. If an aberrant condition occurs within a module and its effects are constrained within that module, the impact of error-induced side effects will be minimized.

IV. Architecture: Software architecture alludes to "*the overall structure of the software and the ways in which that structure provides conceptual integrity for a system*". In its simplest form, *architecture* is the structure or organization of program components (modules), the manner in which these components interact, and the structure of data that are used by the components. One goal of software design is to derive an architectural rendering of a system. A set of architectural patterns enables a software engineer to solve common design problems.

The set of properties as part of an architectural design are as follows:

1. **Structural properties**. This aspect of the architectural design representation defines the components of a system (e.g., modules, objects, filters) and the manner in which those components are packaged and interact with one another. For example, objects are packaged to encapsulate both data and the processing that manipulates the data and interact via the invocation of methods.

2. **Extra-functional properties.** The architectural design description should address how the design architecture achieves requirements for performance, capacity, reliability, security, adaptability, and other system characteristics.
3. **Families of related systems.** The architectural design should draw upon repeatable patterns that are commonly encountered in the design of families of similar systems. In essence, the design should have the ability to reuse architectural building blocks.

Given the specification of these properties, the architectural design can be represented using one or more of a number of different models.

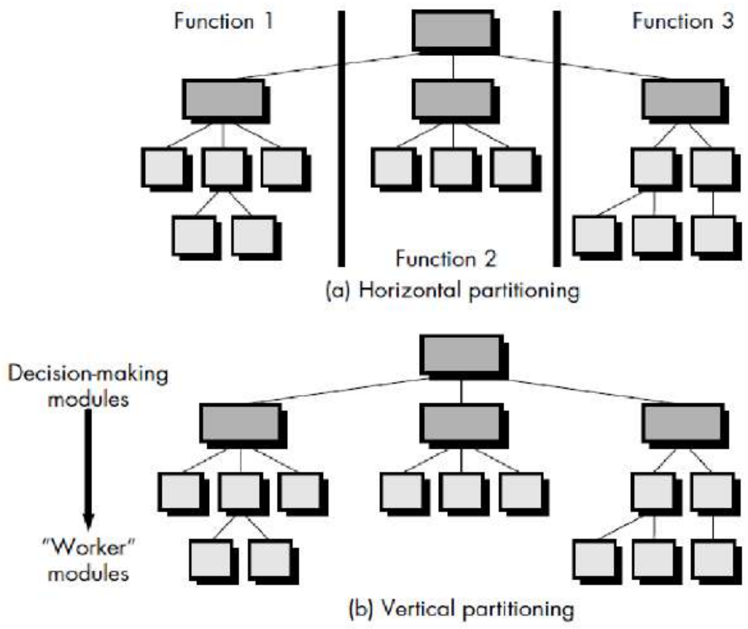
1. ***Structural models*** represent architecture as an organized collection of program components.
2. ***Framework models*** increase the level of design abstraction by attempting to identify repeatable architectural design frameworks (patterns) that are encountered in similar types of applications.
3. ***Dynamic models*** address the behavioral aspects of the program architecture, indicating how the structure or system configuration may change as a function of external events.
4. ***Process models*** focus on the design of the business or technical process that the system must accommodate.
5. ***Functional models*** can be used to represent the functional hierarchy of a system.

V. Information Hiding: The principle of information hiding suggests that modules be "characterized by design decisions that (each) hides from all others." In other words, modules should be specified and designed so that information (algorithms and data) contained within a module is inaccessible to other modules that have no need for such information. Hiding implies that effective modularity can be achieved by defining a set of independent modules that communicate with one another only that information necessary to achieve software function. Abstraction helps to define the procedural (or informational) entities that make up the software. Hiding defines and enforces access constraints to both procedural detail within a module and any local data structure used by the module. The use of information hiding as a design criterion for modular systems provides the greatest benefits when modifications are required during testing and later during software maintenance. Because most data and procedural detail are hidden from other parts of the software, inadvertent errors introduced during modification are less likely to propagate to other locations within the software.

VI. Control Hierarchy: Control hierarchy, also called program structure, represents the organization of program components (modules) and implies a hierarchy of control. The control relationship among modules is expressed in the following way:

1. A module that controls another module is said to be **superordinate** to it, and conversely, a module controlled by another is said to be **subordinate** to the controller. The control hierarchy also represents two subtly different characteristics of the software architecture: visibility and connectivity.
2. **Visibility** indicates the set of program components that may be invoked or used as data by a given component, even when this is accomplished indirectly. **Connectivity** indicates the set of components that are directly invoked or used as data by a given component.

VII. Structural Partitioning: If the architectural style of a system is hierarchical, the program structure can be partitioned both horizontally and vertically. *Horizontal partitioning* defines separate branches of the modular hierarchy for each major program function. *Control modules,* represented in a darker shade are used to coordinate communication between and execution of the functions.



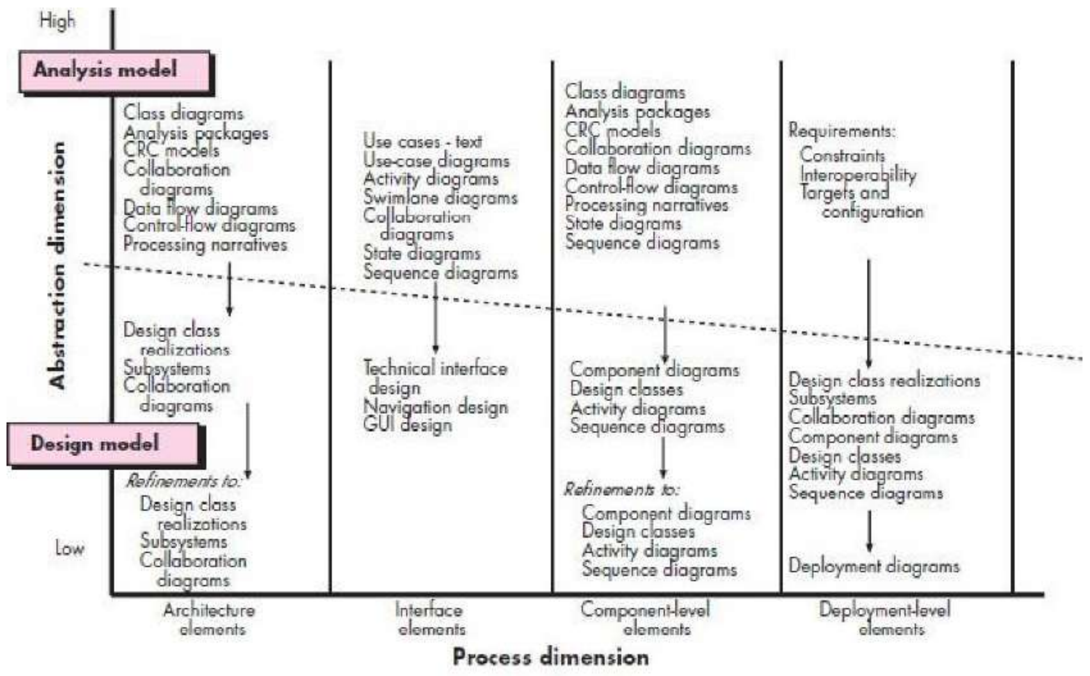
The simplest approach to **horizontal partitioning** defines three partitions—input, data transformation (often called processing) and output. Partitioning the architecture horizontally provides a number of distinct benefits:

- Software that is easier to test
- Software that is easier to maintain
- Propagation of fewer side effects
- Software that is easier to extend

Vertical partitioning, often called factoring, suggests that control (decision making) and work should be distributed top-down in the program structure. The nature of change in program structures justifies the need for vertical partitioning with more maintainability and less side effects which is a key quality factor.

IV. THE DESIGN MODEL:

The design model can be viewed in two different dimensions. The process dimension indicates the evolution of the design model as design tasks are executed as part of the software process. The abstraction dimension represents the level of detail as each element of the analysis model is transformed into a design equivalent and then refined iteratively. Referring to Figure 8.4, the dashed line indicates the boundary between the analysis and design models. The analysis model slowly blends into the design and a clear distinction is less obvious. The elements of the design model use UML diagrams that were used in the analysis model. The difference is that these diagrams are refined and elaborated as part of design; more implementation-specific detail is provided, and architectural structure and style, components that reside within the architecture, and interfaces between the components and with the outside world are all emphasized. You should note, however, that model elements indicated along the horizontal axis are not always developed in a sequential fashion. The deployment model is usually delayed until the design has been fully developed.



Data Design Elements: Like other software engineering activities, data design (sometimes referred to as data architecting) creates a model of data and/or information that is represented at a high level of abstraction. This data model is then refined into progressively more implementation-specific representations that can be processed by the computer-based system.

The structure of data has always been an important part of software design. At the program component level, the design of data structures and the associated algorithms required to manipulate them is essential to the creation of high-quality applications. At the *application level*, the translation of a data model into a database is pivotal to achieving the business objectives of a system. At the *business level*, the collection of information stored in disparate databases and reorganized into a "data warehouse" enables data mining or knowledge discovery that can have an impact on the success of the business itself.

Architectural Design Elements: The architectural design for software is the equivalent to the floor plan of a house. The floor plan gives us an overall view of the house. Architectural design elements give us an overall view of the software. The architectural model is derived from three sources:

(1) Information about the application domain for the software to be built;

(2) specific requirements model elements such as data flow diagrams or analysis classes, their relationships and collaborations for the problem at hand; and

(3) The availability of architectural *styles* and *patterns*.

The architectural design element is usually depicted as a set of interconnected subsystems,
Each subsystem may have its own architecture.

Interface Design Elements: The interface design for software is analogous to a set of detailed drawings for the doors, windows, and external utilities of a house. The interface design elements for software depict information flows into and out of the system and how it is communicated among the components defined as part of the architecture.

There are three important elements of interface design:

(1) The user interface (UI);

(2) External interfaces to other systems, devices, networks, or other producers or consumers of information; and

(3) Internal interfaces between various design components.

These interface design elements allow the software to communicate externally and enable internal communication and collaboration among the components that populate the software architecture.

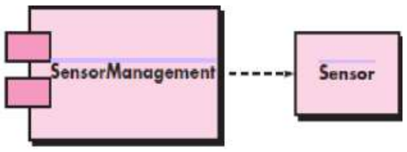
UI design (increasingly called usability design) is a major software engineering action. Usability design incorporates aesthetic elements (e.g., layout, color, graphics, interaction mechanisms), ergonomic elements (e.g., information layout and placement, metaphors, UI navigation), and technical elements (e.g., UI patterns, reusable components).

The design of *external interfaces* requires definitive information about the entity to which information is sent or received. The design of *internal interfaces* is closely aligned with component-level design.

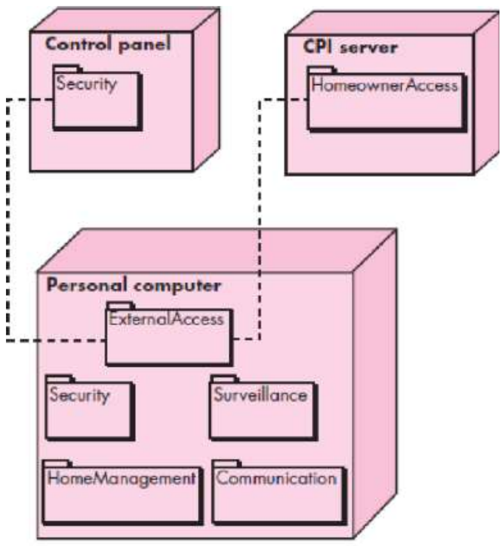
Component-Level Design Elements: The component-level design for software is the equivalent to a set of detailed drawings (and specifications) for each room in a house. The component-level design for software fully describes the internal detail of each software component.

To accomplish this, the component-level design defines data structures for all local data objects and algorithmic detail for all processing that occurs within a component and an interface that allows access to all component operations (behaviors). Within the context of object-oriented software engineering, a component is represented in UML diagrammatic form

.



Deployment-Level Design Elements: Deployment-level design elements indicate how software functionality and subsystems will be allocated within the physical computing environment that will support the software.



During design, a UML deployment diagram is developed and then refined. The diagram shown is in descriptor form. This means that the deployment diagram shows the computing environment but does not explicitly indicate configuration details.

SOFTWARE ENGINEERING

UNIT – 3

CHAPTER – 2

ARCHITECTURAL DESIGN

I. Creating an architectural design

Creating an architectural design in software engineering involves defining the overall structure and organization of a software system. The architectural design phase is critical for ensuring that the software meets its functional and non-functional requirements while being scalable, maintainable, and adaptable to future changes.

The key steps involved in creating an architectural design in software engineering:

1) **Understand Requirements:**
   a) Clearly understand the functional and non-functional requirements of the software system.
   b) Identify any constraints, such as hardware limitations, performance requirements, and regulatory compliance.
2) **System Decomposition:**
   a) Decompose the system into smaller, manageable subsystems or modules.
   b) Define the relationships and interactions between these components.
3) **Choose Architectural Style:**
   a) Select an architectural style that aligns with the requirements and characteristics of the system.
   b) Common architectural styles include layered architecture, microservices, client-server, and event-driven architecture.
4) **Define Components:**
   a) Identify and define the major components or modules of the system.
   b) Specify the responsibilities and interfaces of each component.
5) **Data Design:**
   a) Design the data model, including databases, data structures, and storage mechanisms.
   b) Consider data access patterns, caching strategies, and data synchronization.

6) **User Interface Design:**
   a) Design the user interface components, considering usability, accessibility, and user experience.
   b) Specify how users will interact with the system.
7) **Concurrency and Communication:**
   a) Address concurrency issues by defining how the system will handle simultaneous operations.
   b) Specify communication protocols and mechanisms between components and subsystems.
8) **Scalability and Performance:**
   a) Consider scalability requirements and design the system to handle increased load.
   b) Address performance concerns by optimizing algorithms, data structures, and resource usage.
9) **Security Design:**
   a) Identify potential security risks and design security mechanisms to mitigate them.
   b) Implement authentication, authorization, and encryption strategies.
10) **Error Handling and Recovery:**
   a) Design mechanisms for error detection, reporting, and recovery.
   b) Define how the system will handle exceptional situations and ensure graceful degradation.
11) **Testing and Quality Assurance:**
   a) Plan for testing at the architectural level, including unit testing, integration testing, and system testing.
   b) Define quality metrics and ensure that the architecture supports testing requirements.
12) **Documentation:**
   a) Document the architectural design, including diagrams, interface specifications, and design decisions.
   b) Provide documentation that is useful for both developers and other stakeholders.
13) **Review and Validation:**
   a) Conduct architectural reviews to ensure that the design aligns with the requirements and meets quality standards.
   b) Validate the design against use cases and scenarios.
14) **Evolution and Adaptability:**

a) Design the system with future changes in mind.
b) Consider the extensibility and adaptability of the architecture to accommodate new features and technology changes.

Throughout the architectural design process, collaboration among team members and communication with stakeholders are crucial. It's essential to revisit and refine the architectural design as the project progresses and new insights are gained. Additionally, incorporating feedback and lessons learned from previous projects can contribute to continuous improvement in architectural design practices.

II. Architectural Design:

In software engineering, architectural design refers to the process of defining the high-level structure and organization of a software system. It involves making decisions about the system's components or modules, their relationships, and how they will interact to achieve the overall goals of the software. The architectural design phase is a crucial step in software development as it sets the foundation for the entire system and significantly influences its quality, maintainability, and adaptability.

- It represents the structure of data and program components that are required to build a computer-based system, it considers
    i. The architectural style that the system will take the structure and properties of the components that constitute the system
    ii. The interrelationship that occurs among all architectural components of a system

- **Software Architecture:**

Software architecture refers to the high-level structure or organization of a software system. It involves making key design decisions to define the system's components, their relationships, and the principles that guide their arrangement. Software architecture provides a blueprint for the development team, guiding the construction of the software and ensuring that it meets its functional and non-functional requirements.

1) The software architecture of a program or computing system is the structure (Or) structures of the system which comprises
    i) Software components

ii) Externally visible properties of those components

iii) the relationships among them

2) The architecture is not the operational software, but it is a representation that enables a software engineer to

(i) analyze the effectiveness of the design in meeting its stated Requirements

(ii) Consider architectural alternatives at a stage when making design changes relatively easy

(iii)Reduce the risks associated with the construction of the software

- **Software Architecture – Importance:**

Software architecture is a crucial element in software engineering that influences the success of a software project. It provides a strategic foundation for the development process, addressing key concerns and guiding the team toward building a system that meets both current and future needs.

i. Representations of software architecture are an enabler for communication, between all parties [customer] interested in the development of a computer-based system.

ii. The architecture highlights early design decision that will have a profound impact on all software engineering works that follows on the ultimate success of the system as an operational entity

iii. Architecture constitutes a relatively small, intellectually graspable model of

a. How the system is structured and

b. How its components work together

IV. Architectural styles and patterns

Architectural styles and patterns in software engineering are design solutions or templates that address common design problems. They provide reusable, abstract solutions to recurring design challenges, helping software architects and developers create systems that are well-organized, maintainable, and scalable.

Architectural style describes a system category that encompasses:

(1) A set of **components** (e.g., a database, computational modules) that perform a function required by a system;

(2) A set of **connectors** that enable "communication, coordination and cooperation" among components;

(3) **Constraints** that define how components can be integrated to form the system; and

(4) **Semantic models** that enable a designer to understand the overall properties of a system by analyzing the known properties of its constituent parts.

An architectural style is a transformation that is imposed on the design of an entire system. The intent is to establish a structure for all components of the system.

An architectural pattern, like an architectural style, imposes a transformation on the design of an architecture.

Difference between Architectural Styles and Architectural Patterns:

The terms "architectural style" and "architectural pattern" are related concepts, but they refer to slightly different aspects of software design.

An architectural style is a set of principles that shape the organization and interaction of system components. It defines a family of systems with a similar structure and behavior, providing a high-level abstraction to guide the design process.

An architectural pattern is a reusable solution to a recurring problem in a specific context. It represents a template for solving a particular design problem and provides guidance on how to structure and organize system components.

Patterns can be used in conjunction with an architectural style to shape the overall structure of a system.

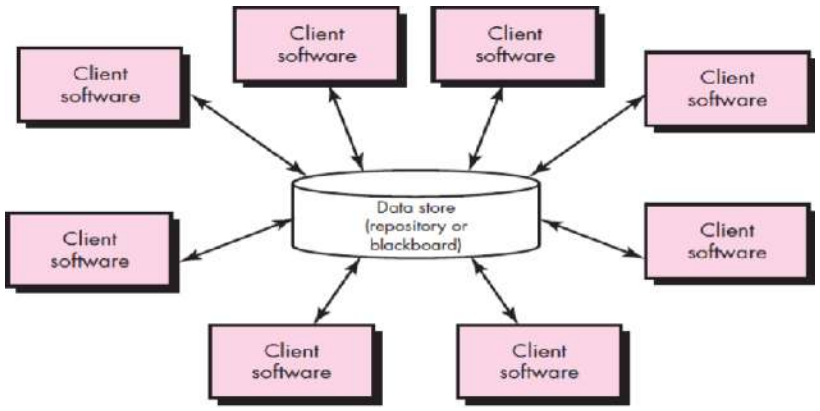
A Brief Taxonomy of Architectural Styles:

There are several architectural styles in software engineering, each with its own set of principles and guidelines for organizing and structuring software systems. These architectural styles provide different approaches to designing software systems, and the choice of a particular style depends on the specific requirements, constraints, and goals of the project. It's common to combine elements from multiple styles, and the selection often depends on factors like scalability, maintainability, and the nature of the application.

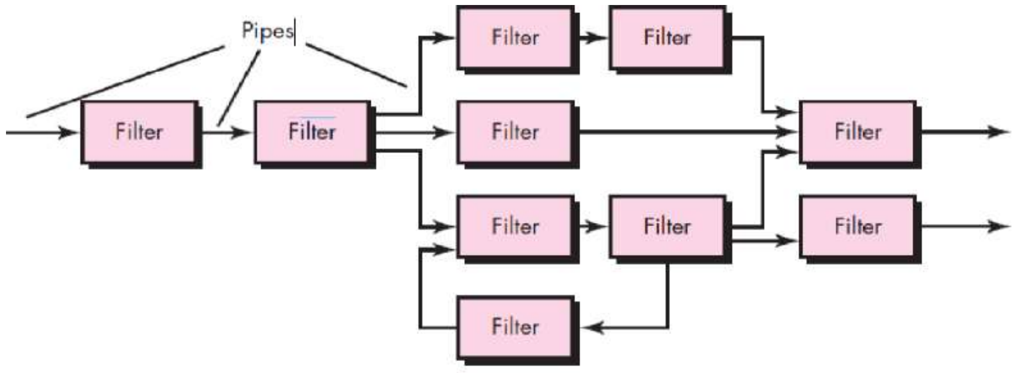
Here are some common architectural styles:

1. **Data-centered architectures:** Data-centric architecture in software engineering is an architectural approach that focuses on the organization, storage, and retrieval of data as a central aspect of the system's design. In a data-centric architecture, the structure and management of data take precedence, and the system's components are designed to facilitate efficient data processing, storage, and access. This architectural style is often used in systems where data is a critical aspect

of the application, such as database-centric applications and information systems. Data store (e.g., a file or database) resides at the center of this architecture and is accessed frequently by other components that update, add, delete, or otherwise modify data within the store.

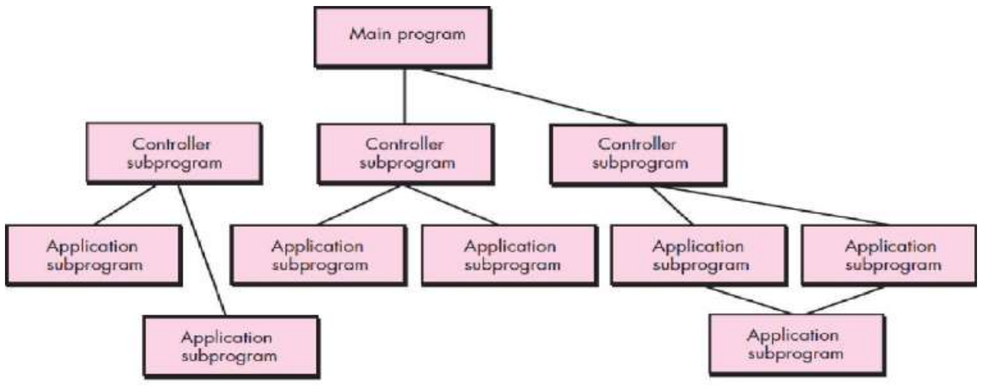


2. **Data-flow architectures:** Data Flow Architecture, also known as Data-Flow Oriented Architecture (DFOA), is an architectural style in software engineering that focuses on the flow of data within a system. This architecture emphasizes the movement and transformation of data as it moves through different processing stages and components. Data Flow Architecture is particularly suitable for systems where data processing and manipulation are central aspects of the application.

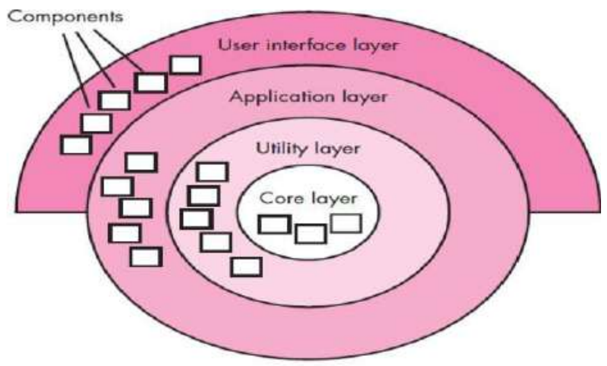


Pipes and filters

3. **Call and return architectures:** In software engineering, "call and return" is a style of software architecture that describes how functions or procedures communicate with each other. This style is often associated with procedural programming and is a fundamental concept in the design of many traditional software systems. It refers to the mechanism by which one part of a program invokes a procedure or function (call) and then returns control to the point immediately after the call (return). This architectural style enables you to achieve a program structure that is relatively easy to modify and scale.



4. **Object-oriented architectures:** Object-oriented architecture (OOA) is an architectural style in software engineering that organizes the software design based on the concept of objects. Object-oriented programming (OOP) is a key part of object-oriented architecture, and the principles of OOP guide the overall design and structure of the software system. Object-oriented architecture is known for encapsulating data and behavior within objects, promoting code reuse, and facilitating the modeling of real-world entities. Communication and coordination between components are accomplished via message passing.



5. **Layered architectures:** Layered architecture, also known as the Layered System or Layered Architecture Style, is a software architectural style that organizes the structure of a software system into distinct and vertically stacked layers, each serving a specific set of functionalities. In a layered architecture, each layer has a well-defined responsibility, and communication between layers follows a strict hierarchy. This architectural style is commonly used to achieve modularity, separation of concerns, and maintainability in software systems.

V. Architectural patterns:

Architectural patterns in software engineering are high-level strategies or templates that provide solutions to commonly occurring design problems. These patterns offer proven, reusable solutions to address various concerns in software architecture, such as scalability, maintainability, and flexibility. These patterns serve as guidelines and best practices, and their application depends on the specific requirements and constraints of a given software project. It's common to combine multiple patterns to create a robust and scalable architecture for complex systems.

some types of architectural patterns:

1) **Layered Architecture:**
   a) **Description:** Organizes the system into layers, each responsible for a specific set of functionalities. Communication typically occurs only between adjacent layers.
   b) **Benefits:** Separation of concerns, modularity, and ease of maintenance.
2) **Model-View-Controller (MVC):**
   a) **Description:** Divides an application into three interconnected components: Model (business logic and data), View (user interface), and Controller (handles user input and updates the model and view).
   b) **Benefits:** Separation of concerns, modularity, and maintainability.
3) **Microservices Architecture:**
   a) **Description:** Decomposes a system into a collection of small, independent services that communicate through well-defined APIs. Each service is developed, deployed, and scaled independently.
   b) **Benefits:** Scalability, flexibility, and ease of development and deployment.
4) **Service-Oriented Architecture (SOA):**
   a) **Description:** Organizes a system as a set of services, where each service represents a self-contained business functionality. Services communicate through well-defined interfaces.
   b) **Benefits:** Reusability, flexibility, and interoperability.
5) **Event-Driven Architecture (EDA):**
   a) **Description:** Emphasizes the production, detection, consumption, and reaction to events. Components communicate through events, allowing for loose coupling and flexibility.
   b) **Benefits:** Decoupling of components, scalability, and responsiveness.
6) **Monolithic Architecture:**
   a) **Description:** All components of the software are interconnected and interdependent. The entire application is deployed as a single unit.
   b) **Benefits:** Simplicity, easier development and deployment in smaller projects.

The architectural patterns for software define a specific approach for handling some behavioral characteristics of the system.

Concurrency: Applications must handle multiple tasks in a manner that simulates parallelism.

Persistence: Data persists if it survives past the execution of the process that created it. Two patterns are common, a) A database management System, b) An application-level persistence.

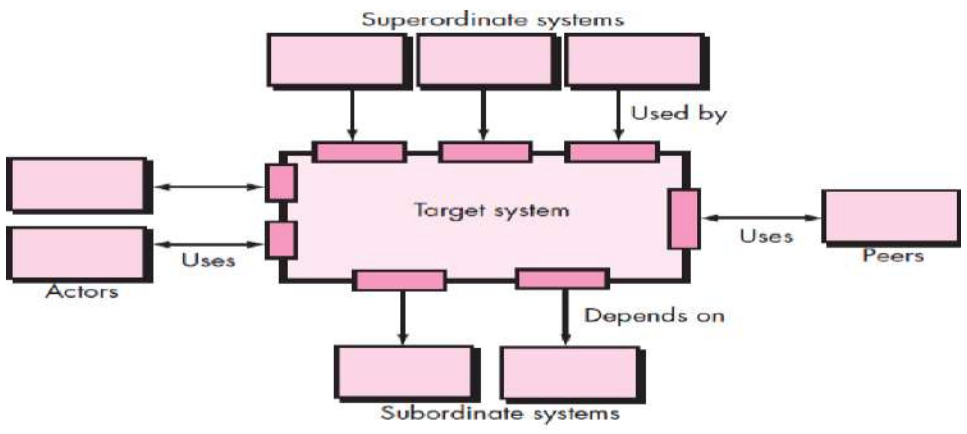
Distribution: The manner in which systems or components within systems communicate with one another in a distributed environment.

CONTEXT MODELS:

A context model is a representation of the external entities (or "actors") that interact with a system and the relationships between them. Context modeling is a crucial step in the early stages of system design, helping to identify the boundaries of the system and understand how it interacts with its environment. There are different ways to represent context models, and one common approach is to use context diagrams and scenarios.

Representing the system in context:

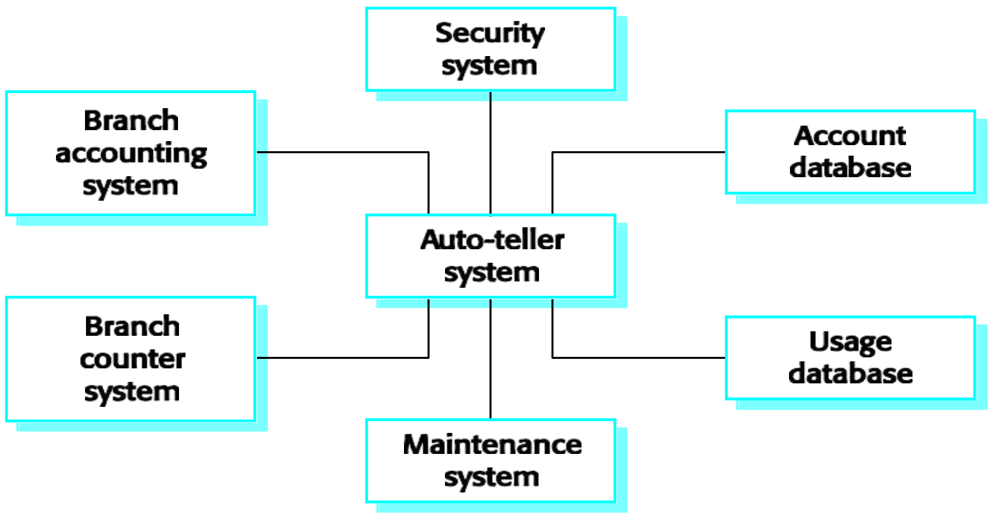
An architectural context diagram in software engineering is a high-level visualization that illustrates the relationships between a system and its external entities or contexts. It is a type of context model that helps in understanding the broader environment in which a software system operates. The diagram provides a simplified view of the system's architecture, highlighting key external entities and their interactions with the system. In an architectural context model, various entities play different roles and have specific relationships with each other. Understanding the terms superordinate, subordinate, peer-level, and actors can provide clarity on the hierarchy and interactions within the system's context.



i. Superordinate systems—those systems that use the target system as part of some higher-level processing scheme.
ii. Subordinate systems—those systems that are used by the target system and provide data or processing that are necessary to complete target system functionality.
iii. Peer-level systems—those systems that interact on a peer-to-peer basis (i.e., information is either produced or consumed by the peers and the target system.
iv. Actors—entities (people, devices) that interact with the target system by producing or consuming information that is necessary for requisite processing. Each of these external entities communicates with the target system through an interface (the small shaded rectangles).

In essence, the architectural context diagram provides a big-picture view of how a software system interacts with its external environment, setting the stage for more detailed design and development activities.

Example: The context diagram of an ATM system: A context diagram for an ATM (Automated Teller Machine) system illustrates the system's interactions with external entities.



SOFTWARE ENGINEERING

UNIT – 3

CHAPTER – 3

UML DESIGN

I. A conceptual model of the UML

A conceptual model in Unified Modeling Language (UML) refers to the high-level representation of a system's structure and behavior. It provides a simplified and abstract view of the system, helping stakeholders understand its key components and interactions.

Unified Modeling Language (UML) is a standardized modeling language used in software engineering for visualizing, specifying, constructing, and documenting the artifacts of a system. The basic building blocks of UML are the elements and diagrams that help model different aspects of a system.

Vocabulary of the UML encompasses three kinds of building blocks:

1. Things
2. Relationships
3. Diagrams

Things are the abstractions that are first-class citizens in a model; relationships tie these things together; diagrams group interesting collections of things.

In Unified Modeling Language (UML), "things" refer to the various elements and constructs used to model the structure and behavior of a system. There are four kinds of things in the UML:

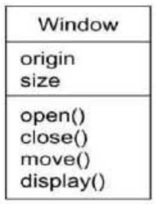
1. Structural things
2. Behavioral things
3. Grouping things
4. Annotational things

These things are the basic object-oriented building blocks of the UML.

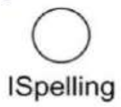
1. Structural Things:

Structural elements in UML represent the static aspects of a system, describing the composition, relationships, and organization of its components. In all, there are seven kinds of structural things.

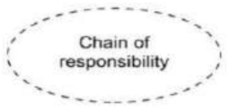
First, a **class** is a description of a set of objects that share the same attributes, operations, relationships, and semantics. A class implements one or more interfaces. Graphically, a class is rendered as a rectangle, usually including its name, attributes, and operations.



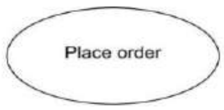
Second, an **interface** is a collection of operations that specify a service of a class or component. An interface therefore describes the externally visible behavior of that element. An interface might represent the complete behavior of a class or component or only a part of that behavior. Graphically, an interface is rendered as a circle together with its name. An interface rarely stands alone. Rather, it is typically attached to the class or component that realizes the interface.



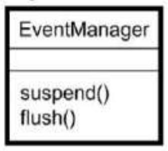
Third, a **collaboration** defines an interaction and is a society of roles and other elements that work together to provide some cooperative behavior. Graphically, a collaboration is rendered as an ellipse with dashed lines, usually including only its name.



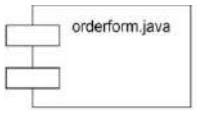
Fourth, a **use case** is a representation of a specific interaction between a system (or its components) and an external actor, which can be a user, another system, or any external entity. Use cases help to describe the functional requirements of a system from an external user's perspective. Graphically, a use case is rendered as an ellipse with solid lines, usually including only its name.



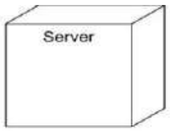
Fifth, an **active class** is a class that plays a special role in modeling concurrent or parallel behavior within a system. An active class is used to represent an object that can execute operations concurrently with other objects. The concept of active classes is particularly relevant in the context of concurrent or parallel programming, where multiple tasks or processes can be executed simultaneously. Graphically, an active class is rendered just like a class, but with heavy lines, usually including its name, attributes, and operations.



Sixth, a **component** is a physical and replaceable part of a system that conforms to and provides the realization of a set of interfaces. A component is a modular and deployable unit of software that encapsulates its contents and provides a well-defined interface. Components are used to represent and model the physical or logical building blocks of a system. They can be libraries, executables, documents, or any other artifacts that are part of the system. Graphically, a component is rendered as a rectangle with tabs, usually including only its name, as



Seventh, a **node** is a physical element that exists at run time and represents a computational resource, generally having at least some memory and, often, processing capability. A set of components may reside on a node and may also migrate from node to node. Graphically, a node is rendered as a cube, usually including only its name.

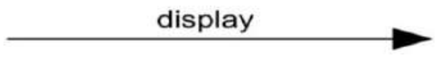


2. Behavioral Things:

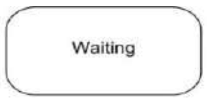
Behavioral things are the dynamic parts of UML models. These are the verbs of a model, representing behavior over time and space. In all, there are two primary kinds of behavioral things.

First, an interaction is a behavior that comprises a set of messages exchanged among a set of objects within a particular context to accomplish a specific purpose. An interaction involves a number of other elements, including messages, action sequences (the behavior invoked by a message), and links (the connection between objects).

Graphically, a message is rendered as a directed line, almost always including the name of its operation.



Second, a state machine is a behavior that specifies the sequences of states an object or an interaction goes through during its lifetime in response to events, together with its responses to those events. The behavior of an individual class or a collaboration of classes may be specified with a state machine. A state machine involves a number of other elements, including states, transitions (the flow from state to state), events (things that trigger a transition), and activities (the response to a transition). Graphically, a state is rendered as a rounded rectangle, usually including its name and its sub states.

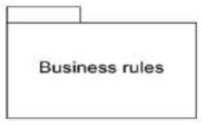


These two elements• interactions and state machines• are the basic behavioral things that you may include in a UML model. Semantically, these elements are usually connected to various structural elements, primarily classes, collaborations, and objects.

3. Grouping Things

Grouping things are the organizational parts of UML models. These are the boxes into which a model can be decomposed. In all, there is one primary kind of grouping thing, namely, packages.

A package is a general-purpose mechanism for organizing elements into groups. Structural things, behavioral things, and even other grouping things may be placed in a package. Unlike components (which exist at run time), a package is purely conceptual (meaning that it exists only at development time). Graphically, a package is rendered as a tabbed folder, usually including only its name and, sometimes, its contents.

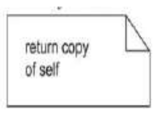


Packages are the basic grouping things with which you may organize a UML model. There are also variations, such as frameworks, models, and subsystems (kinds of packages).

4. **Annotational Things**

Annotations provide a way to add supplementary information or comments to a model without affecting its semantics. Annotations are not formal UML elements but are commonly used to enhance the clarity and understanding of a model. There is one primary kind of annotational thing, called a note.

A note is simply a symbol for rendering constraints and comments attached to an element or a collection of elements. Graphically, a note is rendered as a rectangle with a dog-eared corner, together with a textual or graphical comment.

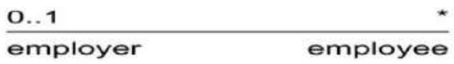


RELATIONSHIPS IN THE UML

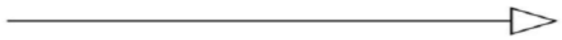
Unified Modeling Language (UML) provides several types of relationships to model the associations and connections between different elements in a system. These relationships help to define the structure, behavior, and interactions within a system. There are four kinds of relationships in the UML:

First, a **dependency** Represents a relationship where a change in one element may affect another. Indicates that a client element depends on a supplier element. Denoted by a dashed arrow pointing from the dependent to the independent element.

Second, an **association** Represents a bi-directional relationship between two classes. Indicates a structural connection between instances of the associated classes. Can be annotated with multiplicity to specify the number of instances involved.



Third, a ***generalization*** Represents an "is-a" relationship between a general (superclass) and a specific (subclass) class. Allows the subclass to inherit attributes and behaviors from the superclass. Denoted by a solid line with an open arrowhead pointing to the superclass.



Fourth, a ***realization,*** represents the implementation of an interface by a class or a component. Indicates that the realizing class provides the operations defined by the interface. Denoted by a dashed line with an open arrowhead pointing to the interface.

DIAGRAMS IN THE UML

Unified Modeling Language (UML) diagrams are widely used in software engineering and other fields for several reasons:

a) Visualization: UML diagrams provide a visual representation of complex systems, making it easier for stakeholders to understand and comprehend the system's structure, behavior, and interactions.

b) Communication: UML diagrams serve as a common language for communication among stakeholders, including developers, analysts, designers, and clients. They offer a visual way to convey ideas and concepts.

c) Specification: UML allows for the specification of system requirements, design, and architecture in a standardized and unambiguous manner. This helps ensure that all parties involved have a shared understanding of the system.

d) Design and Analysis: UML supports various types of diagrams, each focusing on specific aspects of a system. This makes it a versatile tool for both high-level system design and detailed analysis of specific components or behaviors.

e) Documentation: UML diagrams serve as documentation for software projects, providing a visual record of decisions made during the design and development process. This documentation is valuable for future maintenance and updates.

f) Analysis and Modeling: UML supports the modeling of system requirements, use cases, classes, objects, interactions, and more. This modeling capability helps analysts and designers explore and refine system specifications.

g) Code Generation: Some UML tools support code generation, allowing developers to automatically generate code from UML diagrams. This helps maintain consistency between the design and implementation phases.

h) System Understanding: UML diagrams aid in gaining a holistic understanding of a system by capturing its various aspects, including static structure, dynamic behavior, and deployment.

i) Iterative Development: UML supports an iterative and incremental development process. As a project evolves, UML diagrams can be updated and refined to reflect changes in requirements, design decisions, or implementation details.

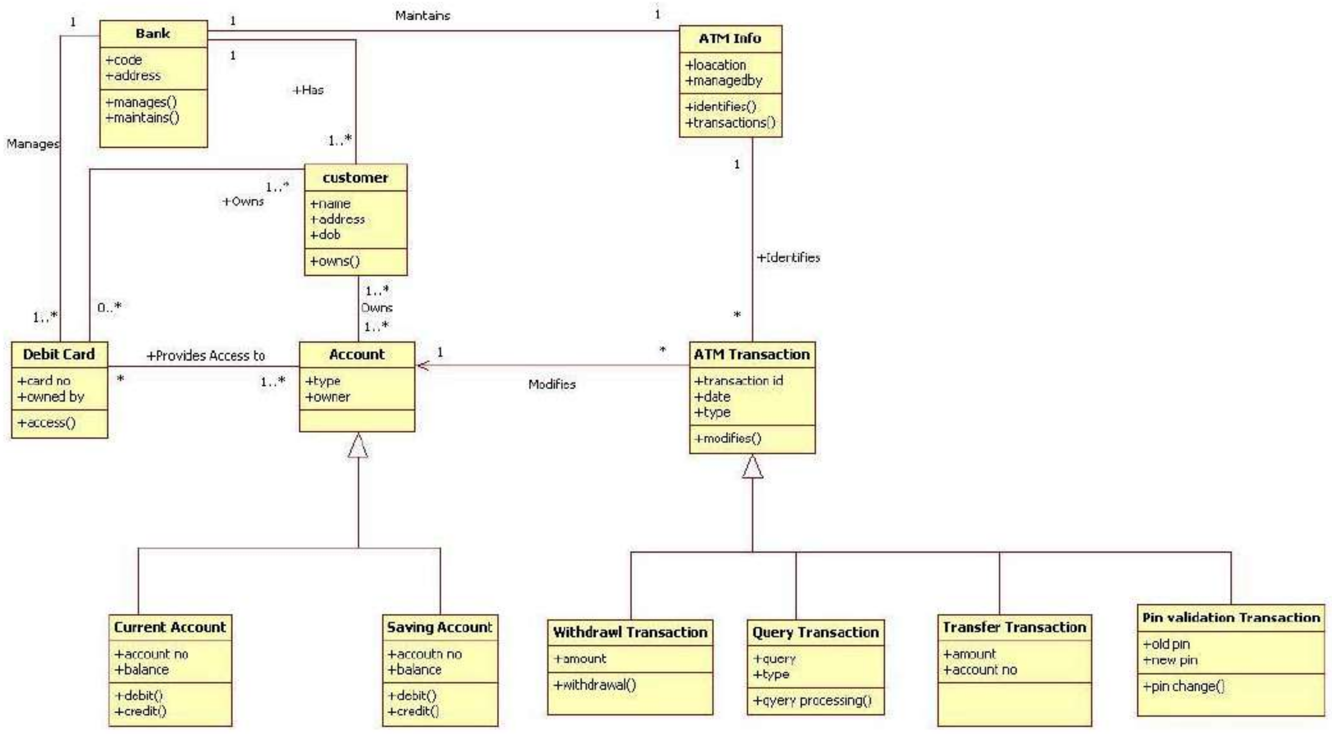
j) Collaboration: UML diagrams facilitate collaboration among team members, enabling them to discuss, review, and refine the system design collaboratively. This is particularly useful in distributed or interdisciplinary development environments.

k) Problem Solving: UML diagrams help in problem-solving by providing a clear and visual representation of the system. They allow teams to identify potential issues, refine designs, and make informed decisions. For this reason, the UML includes nine such diagrams:

1. Class diagram
2. Object diagram
3. Use case diagram
4. Sequence diagram
5. Collaboration diagram
6. State chart diagram
7. Activity diagram
8. Component diagram
9. Deployment diagram

Class diagrams:

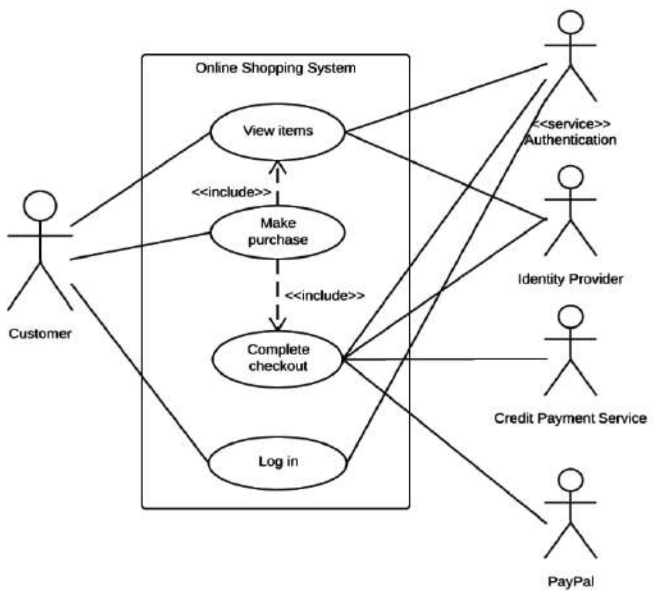
Illustrates the static structure of a system, showing classes, their attributes, methods, and relationships. Used for modeling the object-oriented aspects of a system.



Class diagram for ATM system

Use case diagrams:

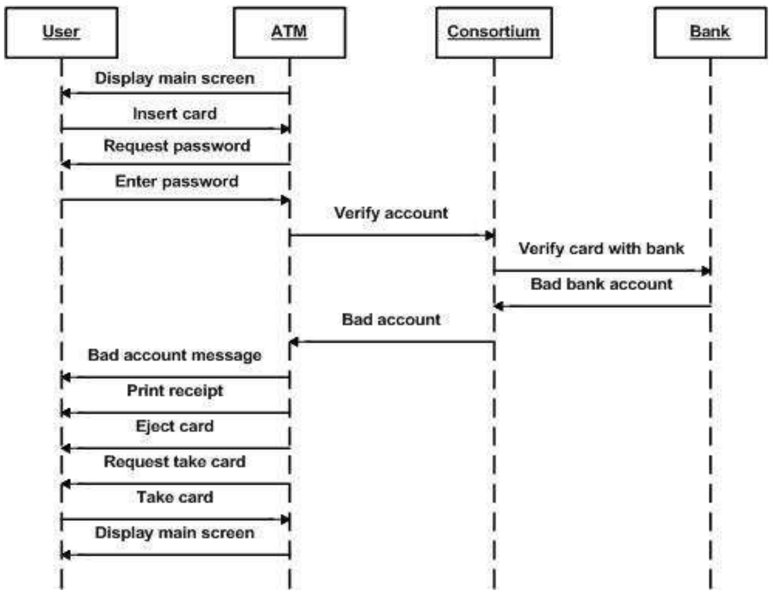
Describes the functional requirements of a system from an external user's perspective. Represents use cases, actors, and their relationships to model system behavior.



Use case diagram for Online Shopping system

Interaction diagrams – Sequence Diagram:

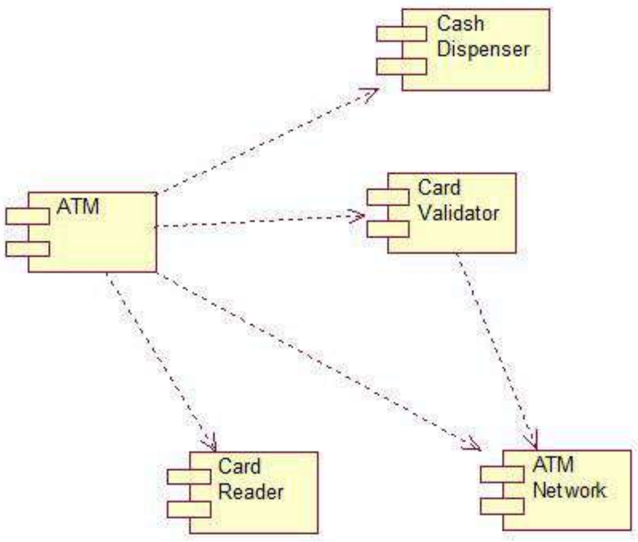
Depicts the interactions between objects or components over time. Shows the order of messages exchanged between objects in response to various stimuli.



Sequence diagram for ATM system.

Component diagrams:

Depicts the physical components of a system and their dependencies. Illustrates the organization and dependencies among components in the implementation of a system.



Component diagram for ATM system

SOFTWARE ENGINEERING

UNIT – 3

CHAPTER – 4

INTRODUCTION TO MAVEN, INSTALLATION AND CONFIGURATION

I. Definition of Maven

Apache Maven is a widely used build automation and project management tool. It is part of the Apache Software Foundation's project and is used for managing the build lifecycle of a software project. Maven can handle tasks such as compiling source code, running tests, packaging artifacts, and managing project dependencies. To use Maven, you typically create a Maven project by defining a POM file and then use Maven commands to perform various tasks like building, testing, and packaging your project. Maven simplifies the build process and promotes best practices in project management and development.

1. Maven is build tool that helps in project management. This tool helps in building and documenting the project.
2. Maven is written in java and c#.
3. It is based on the project object model (POM) and focuses on simplification and standardization of the building process.
4. This tool is used to build and manage any java-based project. It simplifies the day-to-day work of java developers and helps them in their projects.

II. Maven – A build tool

A build tool is a software tool that automates the process of compiling source code, managing project dependencies, and producing executable programs or libraries from the source code. The primary goal of a build tool is to streamline and automate the build process, making it more efficient, repeatable, and less error-prone. Build tools are commonly used in software development to transform source code into a deliverable product.

The build tool is needed for the following processes:

1. Generating source code
2. Generating documentation from the source code
3. Compiling source code

4. Packaging the compiled codes into JAR files
5. Installing the packaged code in the local repository, server, or central repository

III. Use of Maven

Maven is widely used in the software development industry for various purposes. Advantages of using Maven:

i. Project Management: Maven serves as a comprehensive project management tool. It helps in defining project structure, dependencies, and configurations through the Project Object Model (POM) in the pom.xml file. This standardized project structure makes it easier for team members to understand and collaborate on projects.

ii. Dependency Management: One of Maven's significant strengths is its ability to manage project dependencies. It automatically downloads and includes libraries or modules from remote repositories, such as Maven Central, based on the dependencies specified in the POM file. This simplifies the handling of external libraries and ensures that the project has the correct and compatible versions of dependencies.

iii. Build Automation: Maven automates the build process, allowing developers to perform common tasks, such as compiling source code, running tests, and packaging artifacts, with simple commands. The build lifecycle, defined in Maven, includes phases like compile, test, package, install, and deploy. This consistency in build processes helps in maintaining a uniform and predictable development environment.

iv. Plugin Ecosystem: Maven has a rich ecosystem of plugins that extend its functionality. These plugins cover a wide range of tasks, including code analysis, documentation generation, reporting, and integration with other tools. Developers can customize the build process by configuring or adding plugins to meet project-specific requirements.

v. Consistent Project Structure: Maven enforces a standard project structure, making it easier for developers to navigate and understand the layout of a project. This convention over configuration approach promotes consistency across projects, making it simpler for team members to switch between different codebases.

vi. Integration with Continuous Integration (CI) Systems: Maven integrates seamlessly with popular Continuous Integration tools such as Jenkins, Travis CI, and others. This allows for automated builds, tests, and deployments as part of the CI/CD (Continuous Integration/Continuous Deployment) pipeline.

vii. Ease of Project Reporting: Maven generates comprehensive project reports, including test results, code coverage, and other metrics. These reports can be valuable for project managers, team leads, and developers to assess the health and quality of a project.

viii. Multi-Module Projects: Maven supports the creation of multi-module projects, where a large project is divided into smaller, more manageable modules. This modular approach simplifies project maintenance, improves code organization, and facilitates better collaboration among team members.

IV. Maven Repository

A Maven repository is a centralized location where Maven stores and manages project artifacts and their corresponding metadata. These artifacts typically include JAR files, source code, documentation, and other binary or text files that are produced during the build process. Maven repositories play a crucial role in facilitating dependency management and providing a standardized way for developers to share and consume libraries. There are 3 types of maven repository:

1. Local Repository
2. Central Repository
3. Remote Repository

Maven searches for the dependencies in the following order, first in **Local repository** then **Central repository** then **Remote repository**.

a. **Local Repository:** Maven **local repository** is located in your local system. It is created by the maven when you run any maven command. By default, maven local repository is %USER_HOME%/.m2 directory.

We can change the location of maven local repository by changing the **settings.xml** file. It is located in **MAVEN_HOME/conf/settings.xml**, for example: **E:\apache-maven-3.1.1\conf\settings.xml**.

b. **Maven Central Repository:** Maven **central repository** is located on the web. It has been created by the Apache maven community itself.

c. Maven Remote Repository: Maven remote repository is located on the web. Most of libraries can be missing from the central repository such as JBoss library etc, so we need to define remote repository in pom.xml file.

V. Basic concepts of Maven

1. POM

In Maven, a POM (Project Object Model) is an XML file that contains configuration information about the project and its build process. The POM file, named pom.xml, is a fundamental part of a Maven project, and it serves as the project's descriptor. It defines how the project should be built, including its dependencies, plugins, goals, and other configuration details. A Project Object Model or POM is the fundamental unit of work in Maven. It contains default values for most projects. Examples for this, is, the build directory, which is target; the source directory, which is src/main/java; the test source directory, which is src/test/java; and so on. The XML file is in the project home directory. When executing a task or goal, Maven looks for the POM in the current directory. It reads the POM, gets the needed configuration information then executes the goal.

Some of the configuration that can be specified in the POM are the project dependencies, the plugins or goals that can be executed, the build profiles, and so on. Other information such as the project version, description, developers, mailing lists and such can also be specified.

2. Super POM

The Super POM is Maven's default POM. All POMs extend the Super POM unless explicitly set, meaning the configuration specified in the Super POM is inherited by the POMs you created for your projects.

3. Dependencies and repositories

Dependencies are java libraries that are needed for the project. Repositories refer to the directories of packaged JAR files. If dependencies are not present in local repository then Maven downloads them from central repository and store them in local repository.

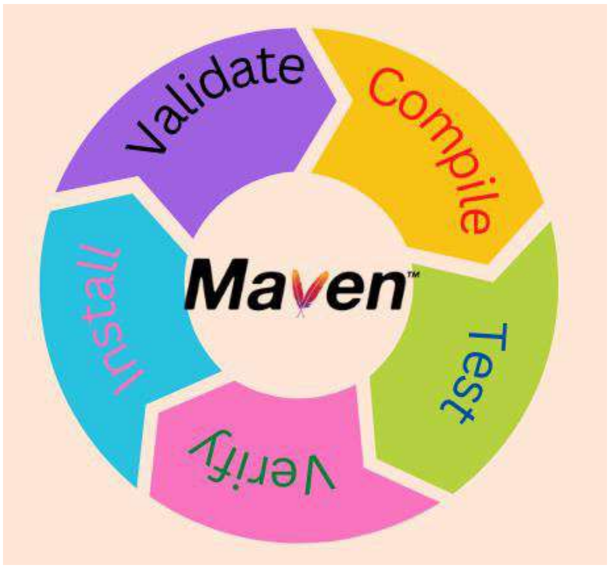
4. **Build life cycles** consist of a sequence of build phases and each build phase consists of a sequence of goals
5. **Goal:** Each goal is responsible for a particular task. When a phase is in run all the goals related to that phase and its plugins are also compiled.

6. **Build Profiles** refer to the set of configuration values that are required to build a project using different configurations. Different build profiles are added to the POM files when enabling different builds. A build profile helps in customizing build for different environments

7. **Build Plugins**: Maven plugin refers to the group of goals. These goals may or may not be of the same phase. The plugins are used to perform specific goal.

VI. Maven build life cycle

The Maven build lifecycle consists of a series of phases that define the order in which goals are executed. Each phase represents a different stage in the build process, and Maven provides a default set of phases that cover common build activities. The build lifecycle is defined in the Project Object Model (POM) file, specifically in the <build> section.



The standard Maven build lifecycle includes three main phases:

I. Clean Lifecycle:

a. clean: The clean phase is responsible for cleaning the project by removing files generated during the build process. This includes the target directory, where compiled classes, JAR files, and other artifacts are usually stored.

II. Default Lifecycle:

a. validate: Validates the project's structure and dependencies.
b. compile: Compiles the source code into bytecode.
c. test: Runs unit tests using a suitable testing framework.
d. package: Takes the compiled code and packages it into a distributable format, such as a JAR or WAR file.
e. verify: Runs integration tests and checks if the package is valid and meets quality criteria.
f. install: Installs the packaged artifact into the local Maven repository, making it available for other projects on the same machine.
g. deploy: Copies the final package to a remote repository, making it available for other developers or projects.

III. Site Lifecycle:

a. site: Generates project documentation, reports, and other project-related information.
b. deploy-site: Deploys the generated site documentation to a remote server.

Developers can execute these phases and associated goals using Maven commands. For example:

To clean the project: mvn clean

To compile the code: mvn compile

To package the code: mvn package

To run tests: mvn test

To install the artifact locally: mvn install

To deploy the artifact to a remote repository: mvn deploy

Maven also allows developers to bind their own goals to these phases or define additional build phases according to the specific needs of a project.

Understanding the Maven build lifecycle and its phases is crucial for effectively managing and customizing the build process in Maven projects. It provides a standardized and predictable way to build, test, and deploy software, promoting consistency across different projects and development environments.

VII. Installation and configuration

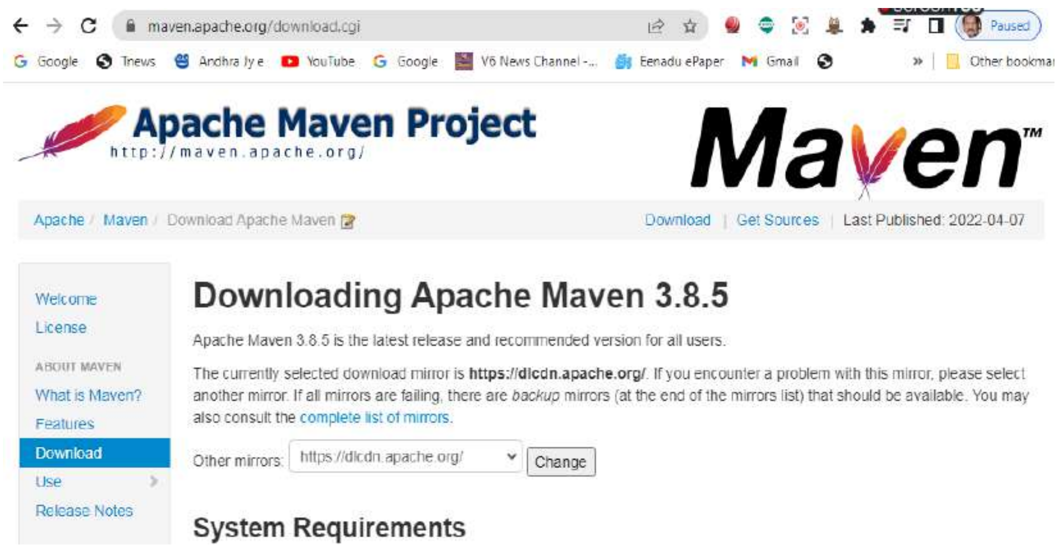
You can download and install Maven on Windows, Linux, and MAC OS platforms. To install Maven on windows, you need to perform the following steps:

1. Download Maven and extract it
2. Add JAVA_HOME and MAVEN_HOME in the environment variable
3. Add maven path in the environment variable
4. Verify Maven

To install maven on windows, you need to perform following steps:

1. Download maven and extract it (For example: a**pache-maven-3.8.5-bin.zip**)

   Note You can get latest version of maven.



Downloading Apache Maven 3.8.5

Apache Maven 3.8.5 is the latest release and recommended version for all users.

The currently selected download mirror is **https://dlcdn.apache.org/**. If you encounter a problem with this mirror, please select another mirror. If all mirrors are failing, there are *backup* mirrors (at the end of the mirrors list) that should be available. You may also consult the complete list of mirrors.

Other mirrors: https://dlcdn.apache.org/ Change

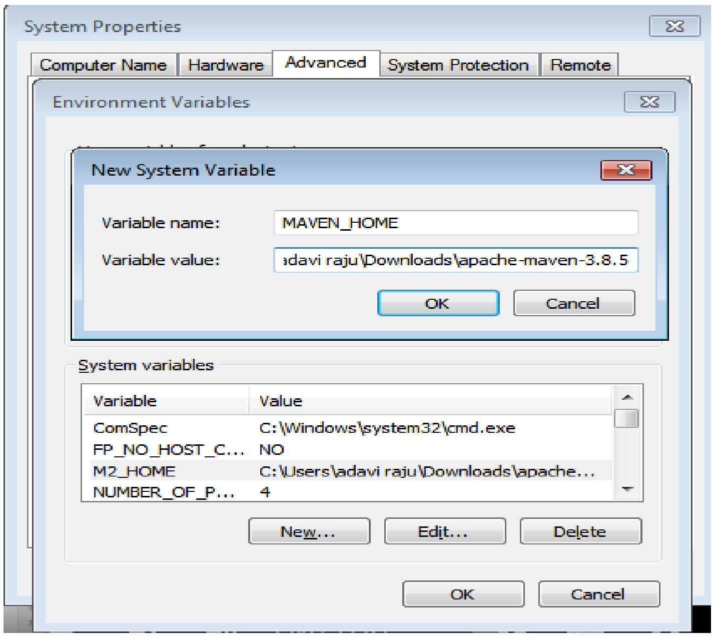
System Requirements

Click on apache-maven-3.859-bin zip. Extract it.

2. Add JAVA_HOME and MAVEN_HOME in environment variable

   Right click on **My Computer** -> **properties** -> **Advanced System Settings** -> **Environment variables** -> **click new button**

   Now **add MAVEN_HOME** in variable name and path of maven in variable value. It must be the home directory of maven i.e., outer directory of bin. For example: **C:\Users\advi raju\Downloads\apache-maven-3.8.5**. It is displayed below:

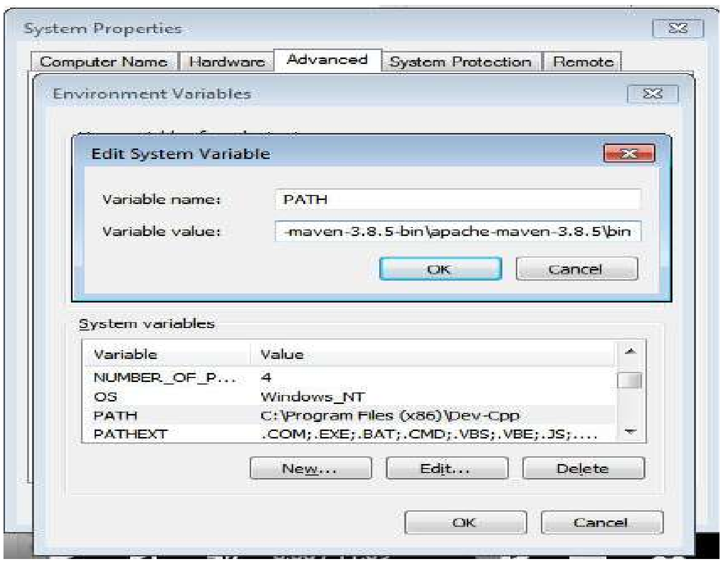


Variable name: MAVEN_HOME
Variable value: ıdavi raju\Downloads\apache-maven-3.8.5

Variable	Value
ComSpec	C:\Windows\system32\cmd.exe
FP_NO_HOST_C...	NO
M2_HOME	C:\Users\adavi raju\Downloads\apache...
NUMBER_OF_P...	4

3. Add maven path in environment variable

The path of maven should be **%maven home%/bin**. For example, **E:\apache-maven-3.8.5\bin**



Variable name: PATH
Variable value: -maven-3.8.5-bin\apache-maven-3.8.5\bin

Variable	Value
NUMBER_OF_P...	4
OS	Windows_NT
PATH	C:\Program Files (x86)\Dev-Cpp
PATHEXT	.COM;.EXE;.BAT;.CMD;.VBS;.VBE;.JS;....

4. Verify Maven

To verify whether maven is installed or not, open the command prompt and write:

```
mvn –version
```

SOFTWARE ENGINEERING

UNIT – 3

CHAPTER – 5

JENKINS ARCHITECTURE

I. Definition of Jenkins

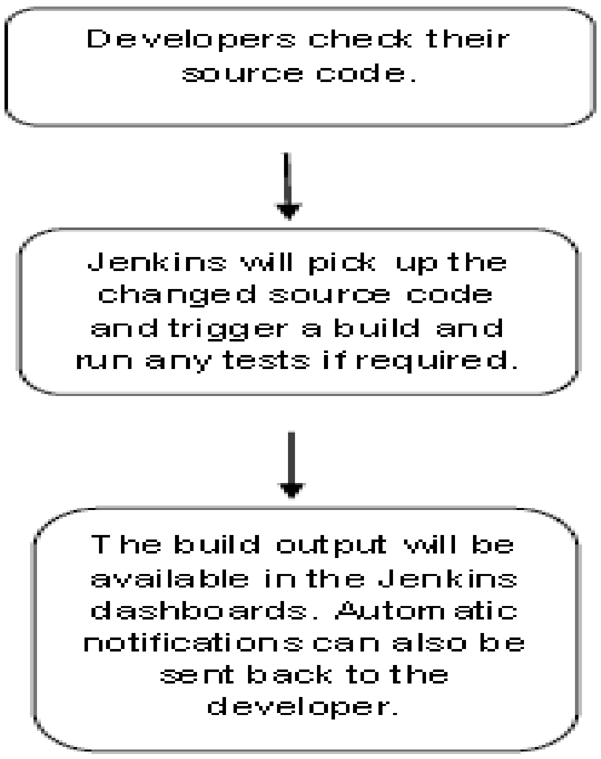
Jenkins is an open-source automation server used for building, testing, and deploying software. It facilitates continuous integration and continuous delivery (CI/CD) by automating the repetitive tasks involved in the software development process. Jenkins allows developers to automate the building, testing, and deployment of their code, making it easier to integrate changes and deliver software more efficiently.

Key features of Jenkins include:

1. **Automation:** Jenkins automates the building and testing of code, making it easier to catch errors early in the development process.
2. **Integration:** It integrates with version control systems (such as Git, SVN), build tools, and testing frameworks, providing a seamless workflow for developers.
3. **Plugins:** Jenkins supports a wide range of plugins that extend its functionality. These plugins can be used to integrate Jenkins with various tools and technologies.
4. **Distributed Builds:** Jenkins allows the distribution of build and test tasks across multiple machines, which can significantly speed up the build process.
5. **Extensibility:** Developers can extend Jenkins' functionality through its open architecture, creating custom plugins to meet specific needs.
6. **Monitoring and Reporting:** Jenkins provides monitoring and reporting features, helping developers and teams track the progress of builds, identify issues, and generate reports.
7. **Pipeline Support:** Jenkins supports the creation of pipelines, which define the entire software delivery process. This allows for the automation of complex workflows involving multiple stages, such as building, testing, and deploying.

By automating these processes, Jenkins helps development teams reduce manual errors, improve collaboration, and achieve a more efficient and reliable software development lifecycle. Many organizations use Jenkins as a core tool in their CI/CD pipelines to streamline the delivery of high-quality software. Jenkins achieves CI (Continuous Integration) with the help of plugins. Plugins is used to allow the integration of various DevOps stages. If you want to integrate a particular tool, you have to install the plugins for that tool. For example: Maven 2 Project, Git, HTML Publisher, Amazon EC2, etc.

Work Flow:


Developers check their source code.
Jenkins will pick up the changed source code and trigger a build and run any tests if required.
The build output will be available in the Jenkins dashboards. Automatic notifications can also be sent back to the developer.

A workflow in Jenkins typically involves the following basic steps in a Continuous Integration (CI) pipeline:

1. **Code Repository:** Developers write code and push it to a version control system (e.g., Git).
2. **Jenkins Trigger:** Jenkins is configured to monitor the code repository for changes (either by polling or using webhooks).
3. **Source Code Checkout:** When changes are detected, Jenkins checks out the latest version of the code from the version control system.
4. **Build:** Jenkins triggers a build process that involves compiling the code, resolving dependencies, and generating artifacts (e.g., executable files, libraries).

5. **Unit Testing:** Automated unit tests are executed to verify that individual components of the code work as expected.
6. **Artifact Archiving:** Jenkins archives the build artifacts for future reference or deployment.
7. **Notification:** Notifications (success, failure, or custom messages) are sent to relevant stakeholders.

II. Continuous Integration

Continuous Integration (CI) is a software development practice that involves automatically integrating code changes from multiple contributors into a shared repository multiple times a day. The primary goal of CI is to detect and address integration issues early in the development process, allowing teams to deliver high-quality software more efficiently. The general practice is that whenever a code commit occurs, a build should be triggered.

Continuous Integration is a fundamental practice in modern software development, especially in agile and DevOps environments. It lays the groundwork for more advanced practices like Continuous Delivery (CD) and helps teams deliver reliable, high-quality software with greater speed and confidence.

Continuous Integration with Jenkins

Continuous Integration (CI) using Jenkins involves setting up automated processes to build, test, and potentially deploy your code every time changes are made to a version control repository. Here's a step-by-step guide to implementing a basic CI pipeline using Jenkins:

Prerequisites:

1) Jenkins Installation: Install Jenkins on a server or a machine.
2) Version Control System: Your project should be hosted on a version control system, such as Git.

Setting up Continuous Integration in Jenkins:

1) Create a New Jenkins Job:
   i) Open Jenkins in your web browser and create a new job:

(a) Click on "New Item" on the Jenkins dashboard.

(b) Enter a name for your job (e.g., "MyCIJob").

(c) Choose the "Freestyle project" option.

2) Configure Source Code Management:

(a) In the job configuration, go to the "Source Code Management" section.

(b) Choose your version control system (e.g., Git).

(c) Provide the repository URL and credentials if needed.

3) Configure Build Steps:

(a) In the job configuration, go to the "Build" section.

(b) Add build steps based on your project requirements. For example:

(c) Build a Maven Project:

(d) Select "Invoke top-level Maven targets."

(e) Enter the goals (e.g., clean install).

(f) Build a Gradle Project:

(g) Select "Invoke Gradle script."

(h) Enter the tasks (e.g., clean build).

4) Configure Post-Build Actions:

(a) In the job configuration, go to the "Post-build Actions" section.

(b) Depending on your project, you may want to:

(c) Archive the artifacts.

(d) Trigger downstream jobs.

(e) Send email notifications.

5) Save and Run the Job:

(a) Save your job configuration and manually trigger a build to test if everything is set up correctly. Jenkins will clone your repository, perform the build steps, and report the build status.

6) Set Up Webhooks (Optional):

(a) For automatic triggering of builds on code changes, set up webhooks in your version control system to notify Jenkins. Alternatively, Jenkins can periodically poll the repository for changes.

7) Advanced Steps (Optional):

i) Install Plugins: Jenkins supports numerous plugins that extend its functionality. Depending on your project, you may need plugins for integrations with tools like SonarQube, Docker, etc.

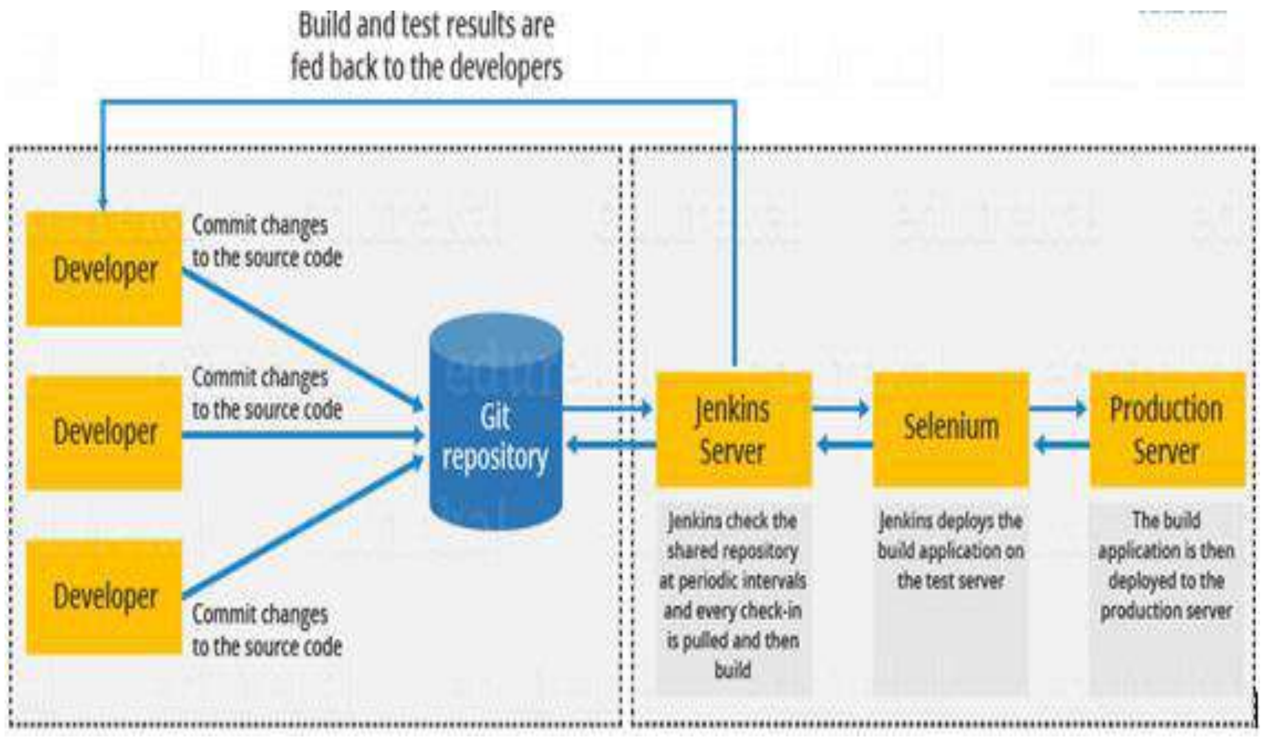
ii) Pipeline as Code (Jenkinsfile): Create a Jenkinsfile to define your CI/CD pipeline as code. This allows you to version control your pipeline alongside your application code.

iii) Integration Testing and Code Quality: Extend your pipeline to include automated integration tests, code quality checks, and any other necessary steps for your project.

iv) Deployments (Optional): If you're practicing Continuous Delivery (CD), integrate deployment steps into your pipeline.

Continuous Integration with Jenkins provides a foundation for automating and streamlining your software development process. As your project evolves, you can enhance your Jenkins pipeline to meet the specific needs of your team and application.

Generic flow diagram of Continuous Integration with Jenkins:


Build and test results are fed back to the developers
Developer
Commit changes to the source code
Developer
Commit changes to the source code
Developer
Commit changes to the source code
Git repository
Jenkins Server
Selenium
Production Server
Jenkins check the shared repository at periodic intervals and every check-in is pulled and then build
Jenkins deploys the build application on the test server
The build application is then deployed to the production server

Let's see how Jenkins works.

1. First of all, a developer commits the code to the source code repository. Meanwhile, the Jenkins checks the repository at regular intervals for changes.

2. Soon after a commit occurs, the Jenkins server finds the changes that have occurred in the source code repository. Jenkins will draw those changes and will start preparing a new build.
3. If the build fails, then the concerned team will be notified.
4. If built is successful, then Jenkins server deploys the built in the test server.
5. After testing, Jenkins server generates feedback and then notifies the developers about the build and test results.
6. It will continue to verify the source code repository for changes made in the source code and the whole process keeps on repeating.

III. Advantages of Jenkins

1. It is an open-source tool.
2. It is free of cost.
3. It does not require additional installations or components. Means it is easy to install.
4. Easily configurable.
5. It supports 1000 or more plugins to ease your work. If a plugin does not exist, you can write the script for it and share with community.
6. It is built in java and hence it is portable.
7. It is platform independent. It is available for all platforms and different operating systems. Like OS X, Windows or Linux.
8. Easy support, since its open source and widely used.
9. Jenkins also supports cloud-based architecture so that we can deploy Jenkins in cloud-based platforms.

IV. Disadvantages of Jenkins

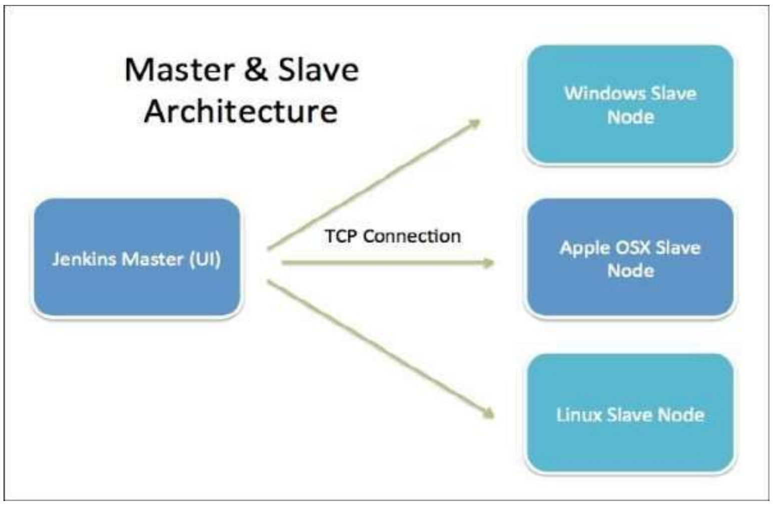
1. Its interface is out dated and not user friendly compared to current user interface trends.
2. Not easy to maintain it because it runs on a server and requires some skills as server administrator to monitor its activity.
3. CI regularly breaks due to some small setting changes. CI will be paused and therefore requires some developer's team attention.

V. Jenkins Architecture

Jenkins follows a distributed architecture that allows it to scale and accommodate various configurations based on the needs of different projects and organizations. The core architecture of Jenkins involves a master-slave setup, plugins, and a web-based user interface. In this architecture, slave and master communicate through TCP/IP protocol.

Jenkins architecture has two components:

1. Jenkins Master/Server
2. Jenkins Slave/Node/Build Server



Jenkins Master

The Jenkins master is the primary server that manages the entire Jenkins environment. It is responsible for handling the scheduling of jobs, distributing tasks to slave nodes, monitoring agents, and serving the Jenkins web interface. It is a web dashboard which is nothing but powered from a war file. By default, it runs on 8080 ports. With the help of Dashboard, we can configure the jobs/projects but the build takes place in Nodes/Slave. By default, one node (slave) is configured and running in Jenkins server.

The server's job or master's job is to handle:

1. Scheduling build jobs.

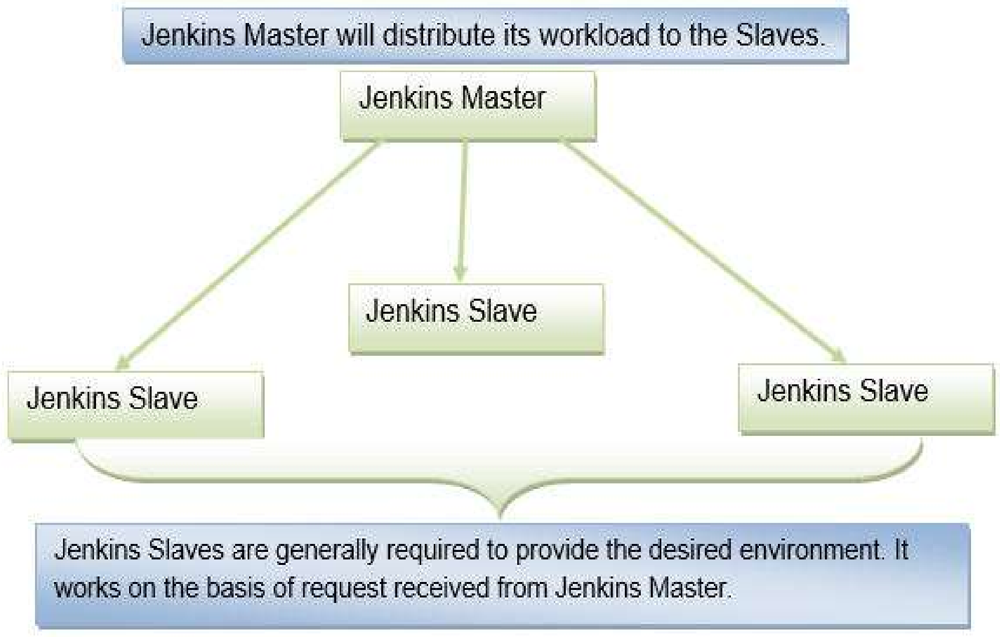
2. Dispatching builds to the nodes/slaves for the actual execution.
3. Monitor the nodes/slaves (possibly taking them online and offline as required).
4. Recording and presenting the build results.
5. A Master/Server instance of Jenkins can also execute build jobs directly.

Jenkins Slave

Jenkins can distribute the workload to one or more slave nodes, allowing concurrent execution of jobs. Slave nodes can be set up on different physical machines or virtual machines, providing the ability to parallelize builds and tasks.

Each slave node has a Jenkins agent running, which communicates with the master to receive tasks and report back the results. Jenkins slave is used to execute the build jobs dispatched by the master. We can configure a project to always run on a particular slave machine, or particular type of slave machine, or simply let the Jenkins to pick the next available slave/node.

As we know Jenkins is developed using Java is platform independent thus Jenkins Master/Servers and Slave/nodes can be configured in any servers including Linux, Windows, and Mac.


Jenkins Master will distribute its workload to the Slaves.
Jenkins Master
Jenkins Slave
Jenkins Slave
Jenkins Slave
Jenkins Slaves are generally required to provide the desired environment. It works on the basis of request received from Jenkins Master.

SOFTWARE ENGINEERING

UNIT – 3

CHAPTER – 6

BUILDING JENKINS PIPELINE

I. Definition of Jenkins Pipeline

Jenkins Pipeline (or simply Pipeline with a capital P) plugins supports implementing and integrating continuous delivery pipelines into Jenkins. Jenkins Pipeline is a suite of plugins for the Jenkins automation server that allows you to define and manage your software delivery pipeline as code. Instead of using the traditional graphical user interface (GUI) to create and configure jobs in Jenkins, you can use a Jenkinsfile to describe your build, test, and deployment process in a script format.

A Jenkinsfile is a text file that defines the steps of your pipeline, including build, test, and deployment phases. It can be written in either Declarative Pipeline Syntax or Scripted Pipeline Syntax. Declarative syntax is a more structured and opinionated way to define pipelines, while scripted syntax provides more flexibility and is based on a Groovy-based domain-specific language.

Approaches to Defining Pipeline Script

In Jenkins, there are two main approaches to defining pipeline scripts: Declarative Pipeline Syntax and Scripted Pipeline Syntax. Each approach has its own style and use cases.

1. Declarative Pipeline Syntax:

Declarative Pipeline Syntax provides a more structured and simplified way of defining pipelines. It is designed to be easy to read and write, making it accessible to users with less scripting experience. It uses a predefined structure with specified sections for defining the pipeline stages, steps, and other configurations.

Example of a Declarative Pipeline:

```
pipeline {
        agent any
        stages {
                stage('Build') {
                                steps {
                                        // Build steps go here
                                        }
                                }
                stage('Test') {
                                steps {
                                        // Test steps go here
                                        }
                                }
                stage('Deploy') {
                                steps {
                                        // Deployment steps go here
                                        }
                                }
                }
        }
```

2. Scripted Pipeline Syntax:

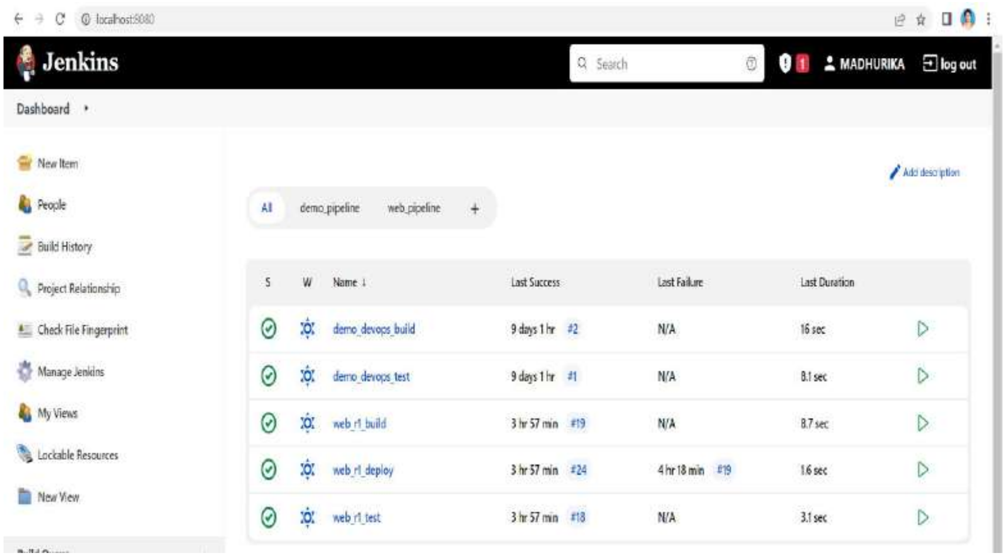
Scripted Pipeline Syntax, on the other hand, is more flexible and allows you to write pipelines using a Groovy-based scripting language. It is suitable for users who are comfortable with programming and provides greater control over the flow of the pipeline.

Example of a Scripted Pipeline:

```
node {
        stage('Build') {
                        // Build steps go here
                        }
        stage('Test') {
                        // Test steps go here
                        }
stage('Deploy') {
                        // Deployment steps go here
                        }
}
```

II. Creating a Simple Pipeline using user interface for Java project without Jenkins Script.

1. Open Jenkins in local host:8080 and create a new item



localhost:8080

Jenkins | Search | 1 | MADHURIKA | log out

Dashboard

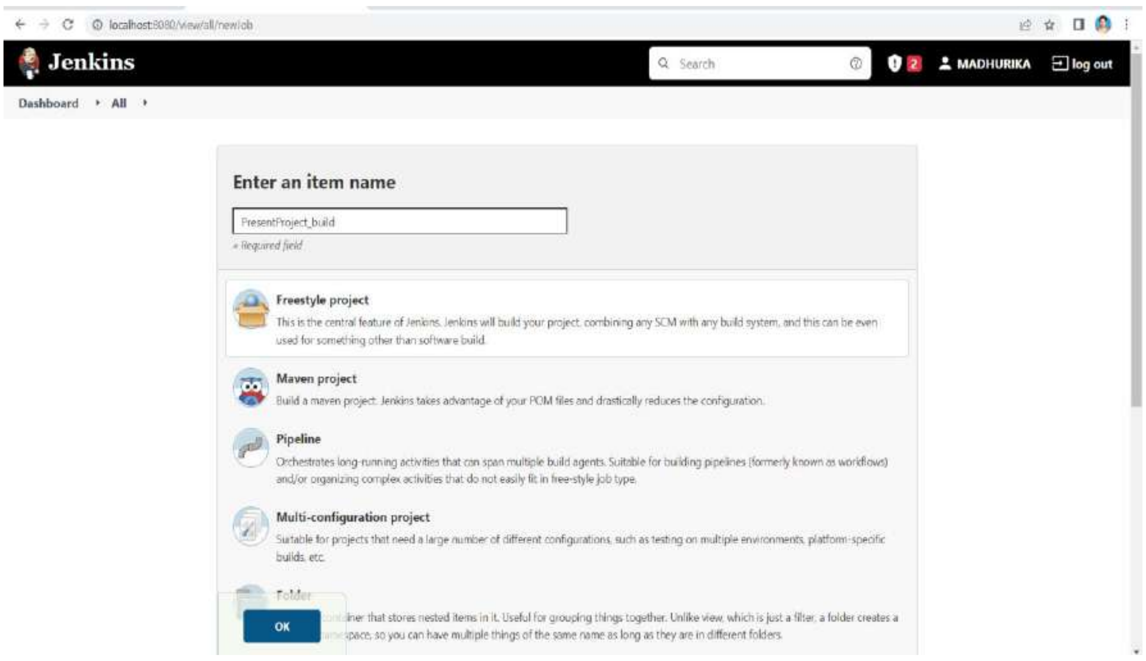
New Item
People
Build History
Project Relationship
Check File Fingerprint
Manage Jenkins
My Views
Lockable Resources
New View

Add description

All | demo_pipeline | web_pipeline | +

S	W	Name ↓	Last Success	Last Failure	Last Duration	
✓	☼	demo_devops_build	9 days 1 hr #2	N/A	16 sec	▷
✓	☼	demo_devops_test	9 days 1 hr #1	N/A	8.1 sec	▷
✓	☼	web_r1_build	3 hr 57 min #19	N/A	8.7 sec	▷
✓	☼	web_r1_deploy	3 hr 57 min #24	4 hr 18 min #19	1.6 sec	▷
✓	☼	web_r1_test	3 hr 57 min #18	N/A	3.1 sec	▷

2. Select a new Freestyle Project give name (eg. PresentProject_build) and then click ok



localhost:8080/view/all/newJob

Jenkins

Dashboard › All ›

Enter an item name

PresentProject_build

» Required field

Freestyle project
This is the central feature of Jenkins. Jenkins will build your project, combining any SCM with any build system, and this can be even used for something other than software build.

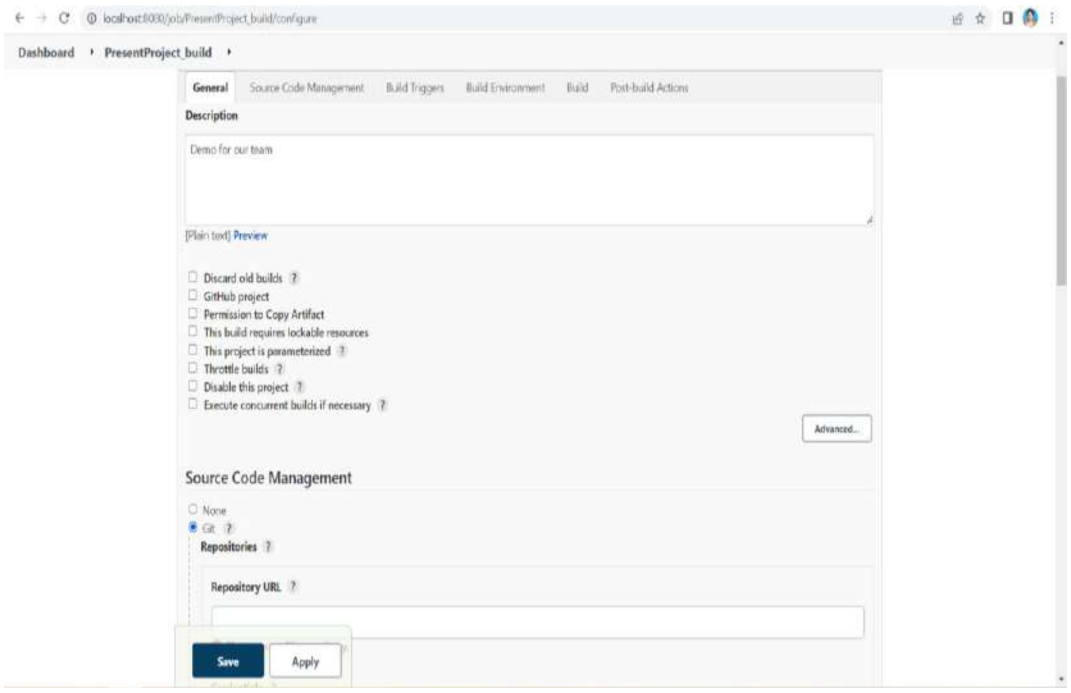
Maven project
Build a maven project. Jenkins takes advantage of your POM files and drastically reduces the configuration.

Pipeline
Orchestrates long-running activities that can span multiple build agents. Suitable for building pipelines (formerly known as workflows) and/or organizing complex activities that do not easily fit in free-style job type.

Multi-configuration project
Suitable for projects that need a large number of different configurations, such as testing on multiple environments, platform-specific builds, etc.

Folder
Creates a container that stores nested items in it. Useful for grouping things together. Unlike view, which is just a filter, a folder creates a separate namespace, so you can have multiple things of the same name as long as they are in different folders.

3. In description type e.g., Build demo
4. Give the Git repository URL of the project to be built



localhost:8080/job/PresentProject_build/configure

Dashboard › PresentProject_build ›

General | Source Code Management | Build Triggers | Build Environment | Build | Post-build Actions

Description

Demo for our team

[Plain text] Preview

- ☐ Discard old builds
- ☐ GitHub project
- ☐ Permission to Copy Artifact
- ☐ This build requires lockable resources
- ☐ This project is parameterized
- ☐ Throttle builds
- ☐ Disable this project
- ☐ Execute concurrent builds if necessary

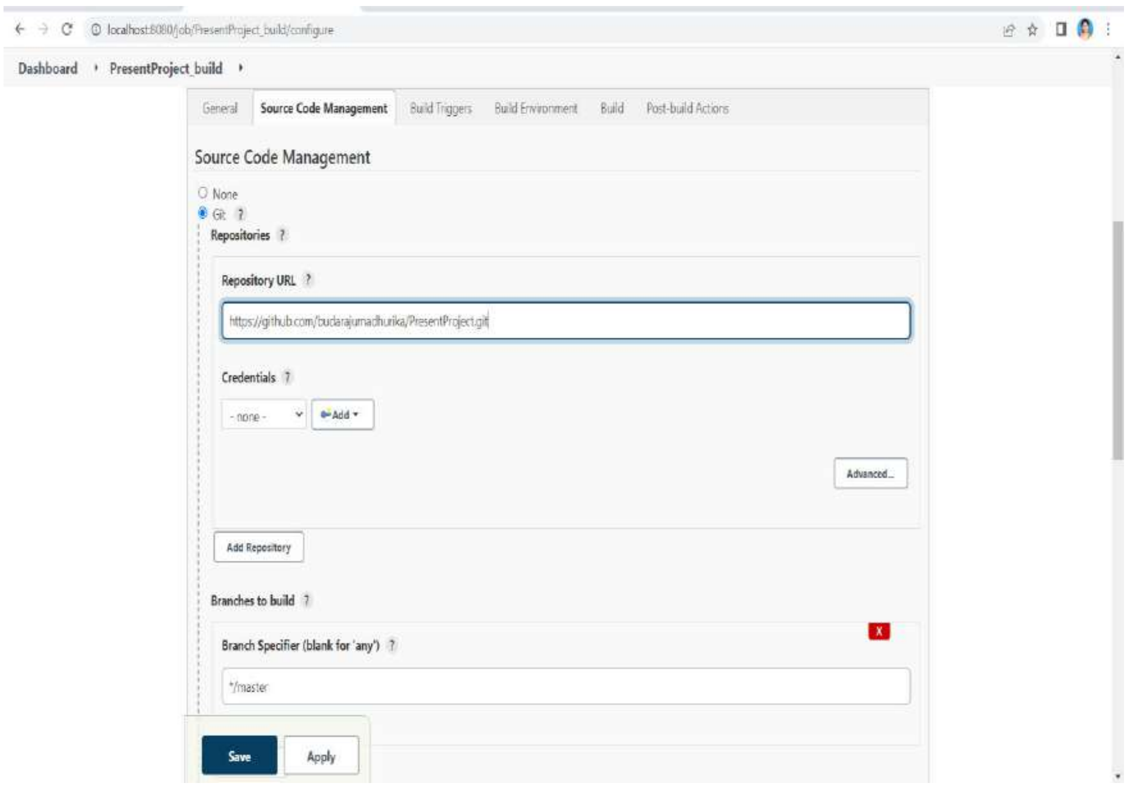
Source Code Management

- ○ None
- ● Git

Repositories

Repository URL

5. After adding the repository URL, Specify the Branch as either Master/Main as it is in the GitHub (eg. My project is in master branch so I have mentioned <u>*/master</u>)


localhost:8080/job/PresentProject_build/configure

Dashboard › PresentProject_build ›

Source Code Management

None
Git

Repositories

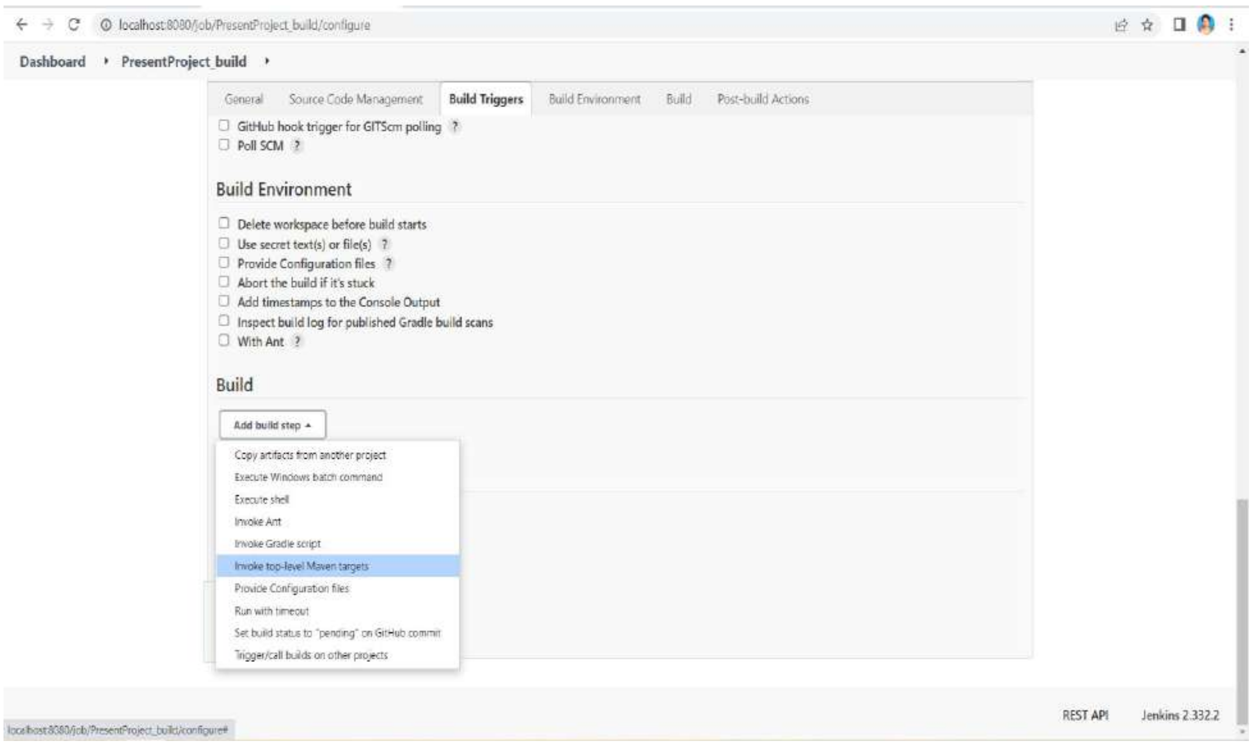
Repository URL: https://github.com/budarajumadhurika/PresentProject.git

Credentials: - none -

Branches to build

Branch Specifier (blank for 'any'): */master

6. Now in Build, select the “Invoke top-level Maven targets”


localhost:8080/job/PresentProject_build/configure

Dashboard › PresentProject_build ›

GitHub hook trigger for GITScm polling
Poll SCM

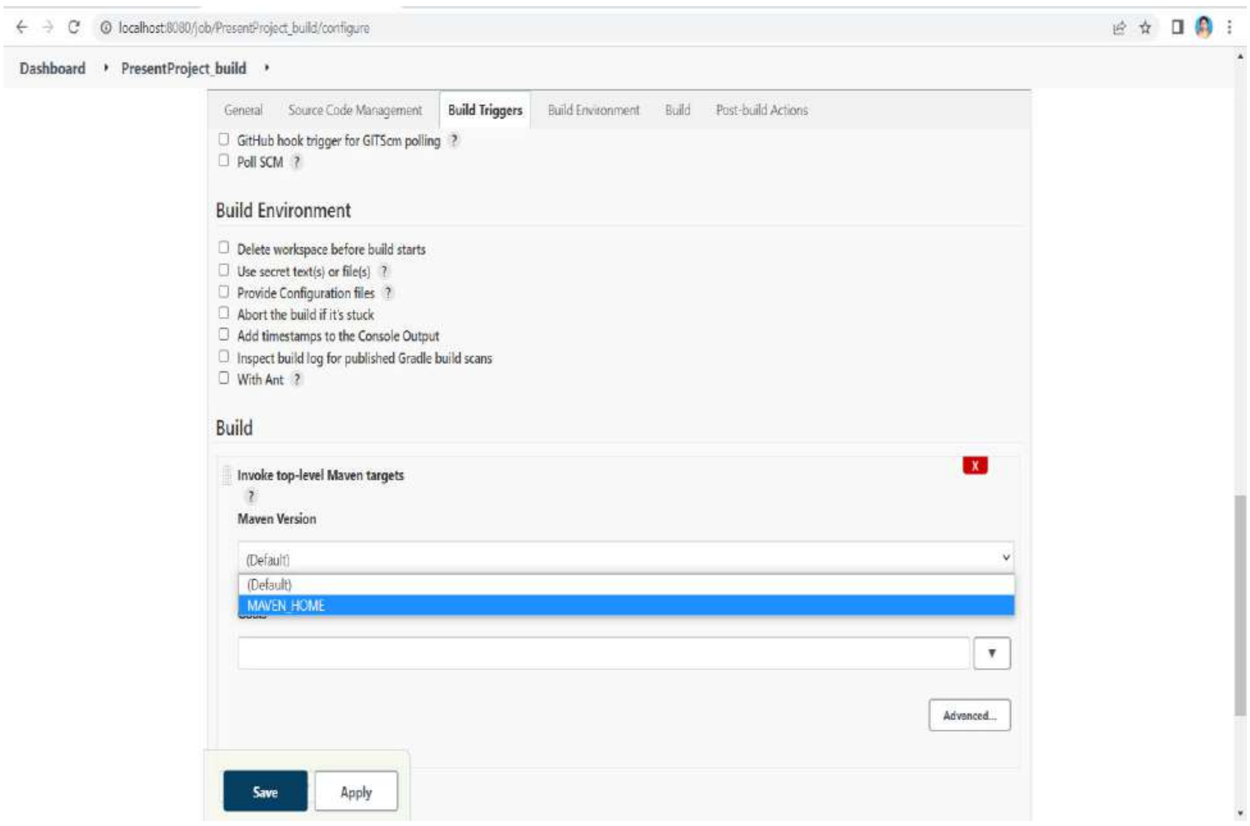
Build Environment

Delete workspace before build starts
Use secret text(s) or file(s)
Provide Configuration files
Abort the build if it's stuck
Add timestamps to the Console Output
Inspect build log for published Gradle build scans
With Ant

Build

General
Source Code Management
Build Triggers
Build Environment
Build
Post-build Actions
Add build step
Copy artifacts from another project
Execute Windows batch command
Execute shell
Invoke Ant
Invoke Gradle script
Invoke top-level Maven targets
Provide Configuration files
Run with timeout
Set build status to "pending" on GitHub commit
Trigger/call builds on other projects
REST API
Jenkins 2.332.2
localhost:8080/job/PresentProject_build/configure#

7. Select the Maven path which is already set in the global credentials in Manage Jenkins (eg MAVEN_HOME)



localhost:8080/job/PresentProject_build/configure

Dashboard › PresentProject_build ›

General | Source Code Management | Build Triggers | Build Environment | Build | Post-build Actions

- [ ] GitHub hook trigger for GITScm polling
- [ ] Poll SCM

Build Environment

- [ ] Delete workspace before build starts
- [ ] Use secret text(s) or file(s)
- [ ] Provide Configuration files
- [ ] Abort the build if it's stuck
- [ ] Add timestamps to the Console Output
- [ ] Inspect build log for published Gradle build scans
- [ ] With Ant

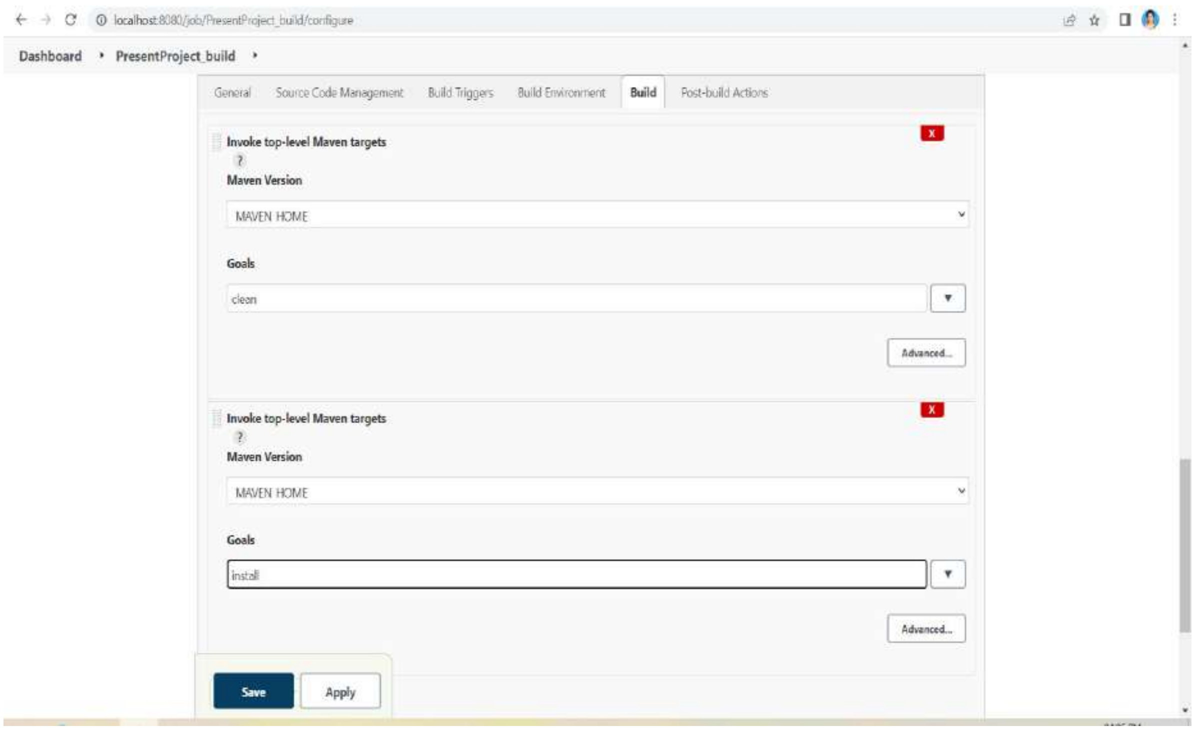
Build

Invoke top-level Maven targets

Maven Version: (Default) — options: (Default), MAVEN_HOME

Advanced... | Save | Apply

8. Set Goals field to clean as done in eclipse



localhost:8080/job/PresentProject_build/configure

Dashboard › PresentProject_build ›

General | Source Code Management | Build Triggers | Build Environment | Build | Post-build Actions

Invoke top-level Maven targets

Maven Version: MAVEN HOME

Goals: clean

Invoke top-level Maven targets

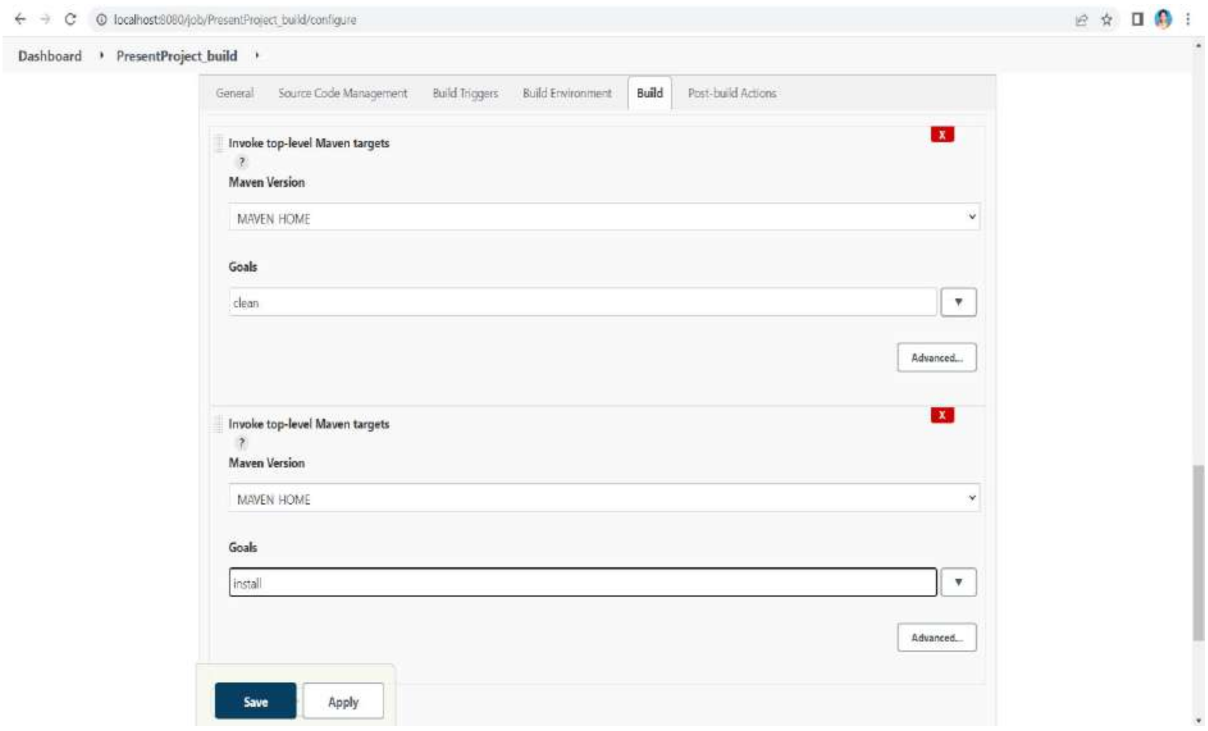
Maven Version: MAVEN HOME

Goals: install

Advanced... | Save | Apply

Again, in Build, select the "Invoke top-level Maven targets"

9. Select the Maven path which is already set in the global credentials in Manage Jenkins (eg MAVEN_HOME)
10. Set Goals field to **Install** as done in eclipse



Invoke top-level Maven targets

Maven Version: MAVEN HOME

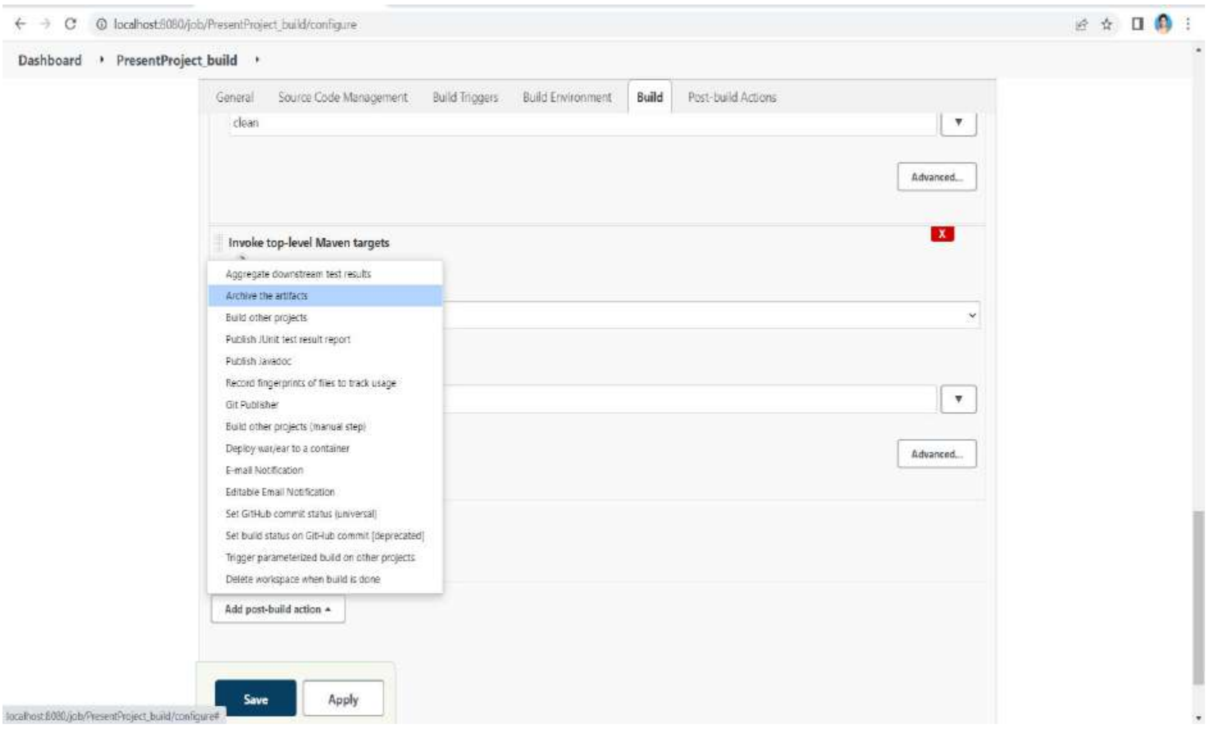
Goals: clean

Invoke top-level Maven targets

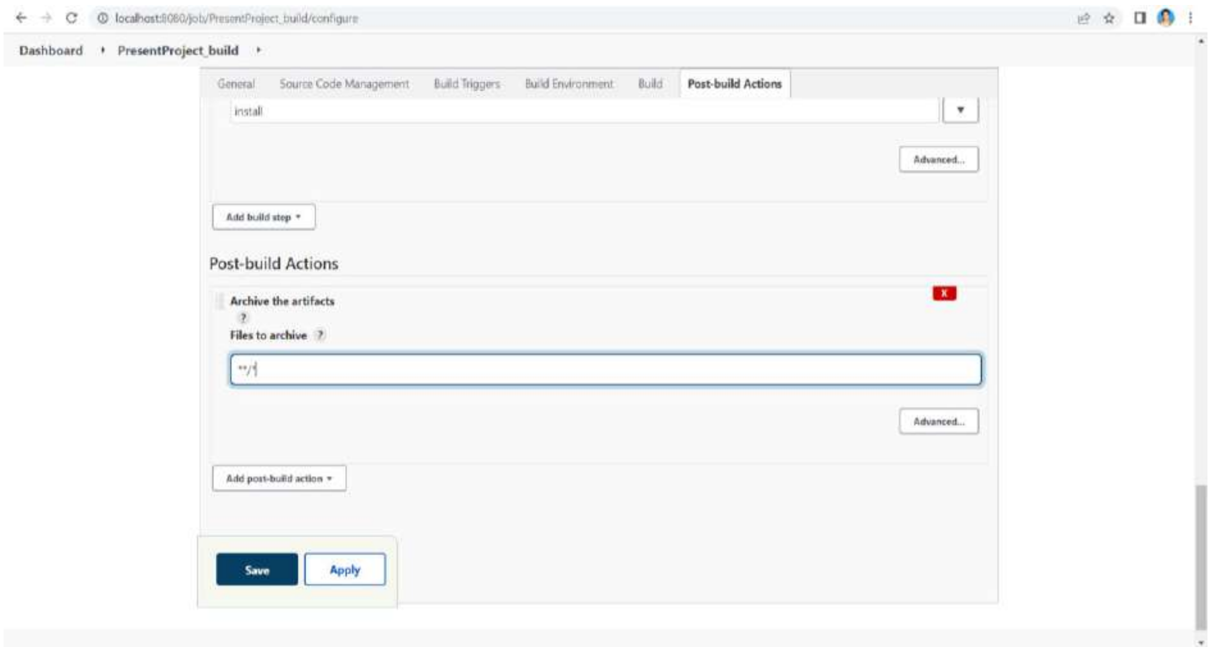
Maven Version: MAVEN HOME

Goals: install

11. Now in post build actions-> select "**Archive the artifacts**", to send the output of build project to the testing team


localhost:8080/job/PresentProject_build/configure
Dashboard
PresentProject_build
General
Source Code Management
Build Triggers
Build Environment
Build
Post-build Actions
clean
Advanced...
Invoke top-level Maven targets
Aggregate downstream test results
Archive the artifacts
Build other projects
Publish JUnit test result report
Publish Javadoc
Record fingerprints of files to track usage
Git Publisher
Build other projects (manual step)
Deploy war/ear to a container
E-mail Notification
Editable Email Notification
Set GitHub commit status (universal)
Set build status on GitHub commit [deprecated]
Trigger parameterized build on other projects
Delete workspace when build is done
Add post-build action
Save
Apply
localhost:8080/job/PresentProject_build/configure#

12. If we want to archive all the artifacts type **/* in Files to Archive



install

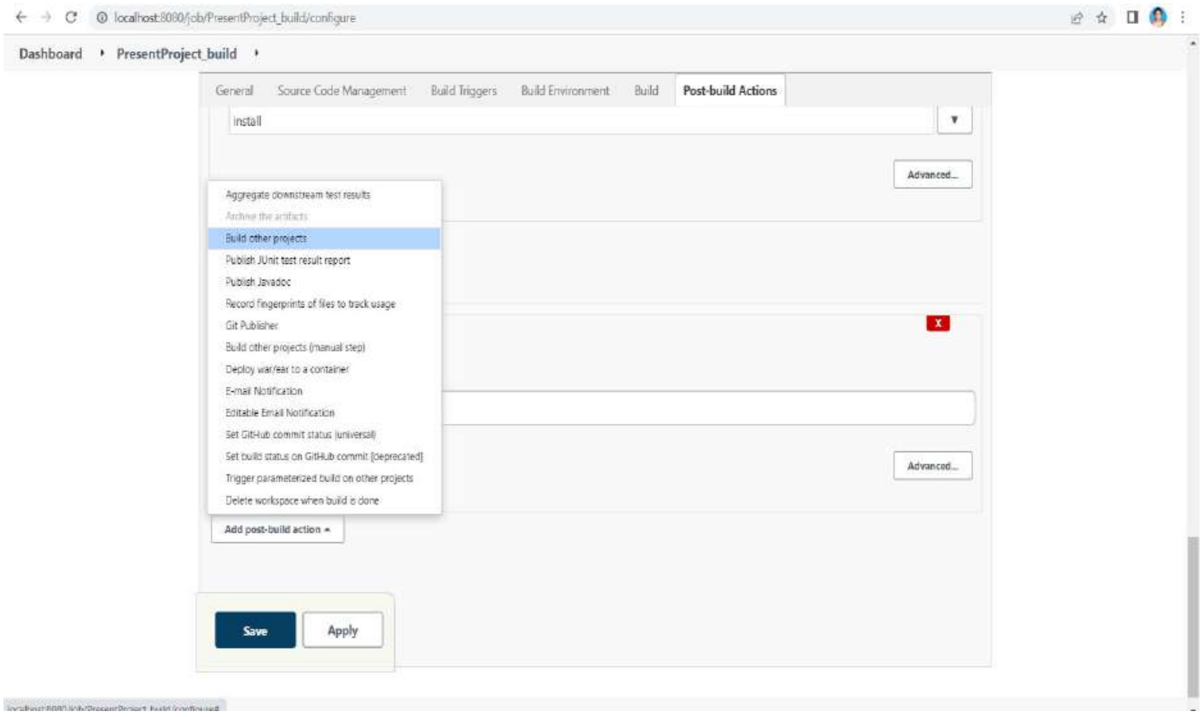
Post-build Actions

Archive the artifacts

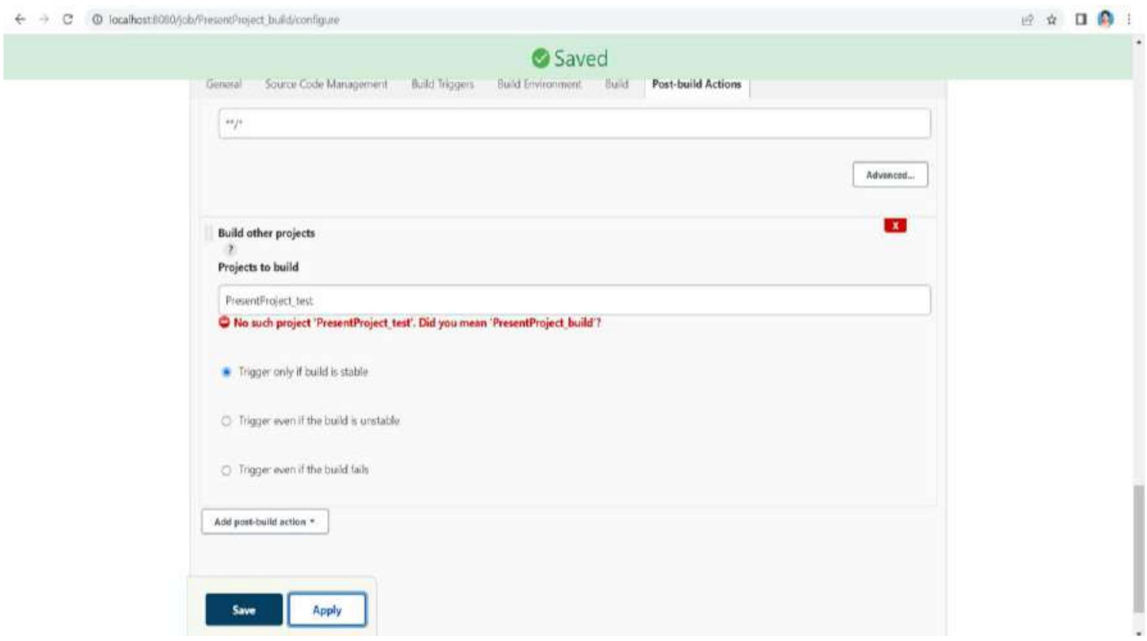
Files to archive: **/*

13. Now the next step is to build other projects, where we will create a test project which will be triggered by the build project.

    For this **in <u>Add Post build Action</u> select "Build other projects"**


localhost:8080/job/PresentProject_build/configure
Dashboard
PresentProject_build
General
Source Code Management
Build Triggers
Build Environment
Build
Post-build Actions
install
Advanced...
Aggregate downstream test results
Archive the artifacts
Build other projects
Publish JUnit test result report
Publish Javadoc
Record fingerprints of files to track usage
Git Publisher
Build other projects (manual step)
Deploy war/ear to a container
E-mail Notification
Editable Email Notification
Set GitHub commit status (universal)
Set build status on GitHub commit [deprecated]
Trigger parameterized build on other projects
Delete workspace when build is done
Add post-build action
Save
Apply

14. In Projects to build enter the next project name as "**PresentProject_test**"
15. Next press on Apply and Save


Saved

General | Source Code Management | Build Triggers | Build Environment | Build | Post-build Actions

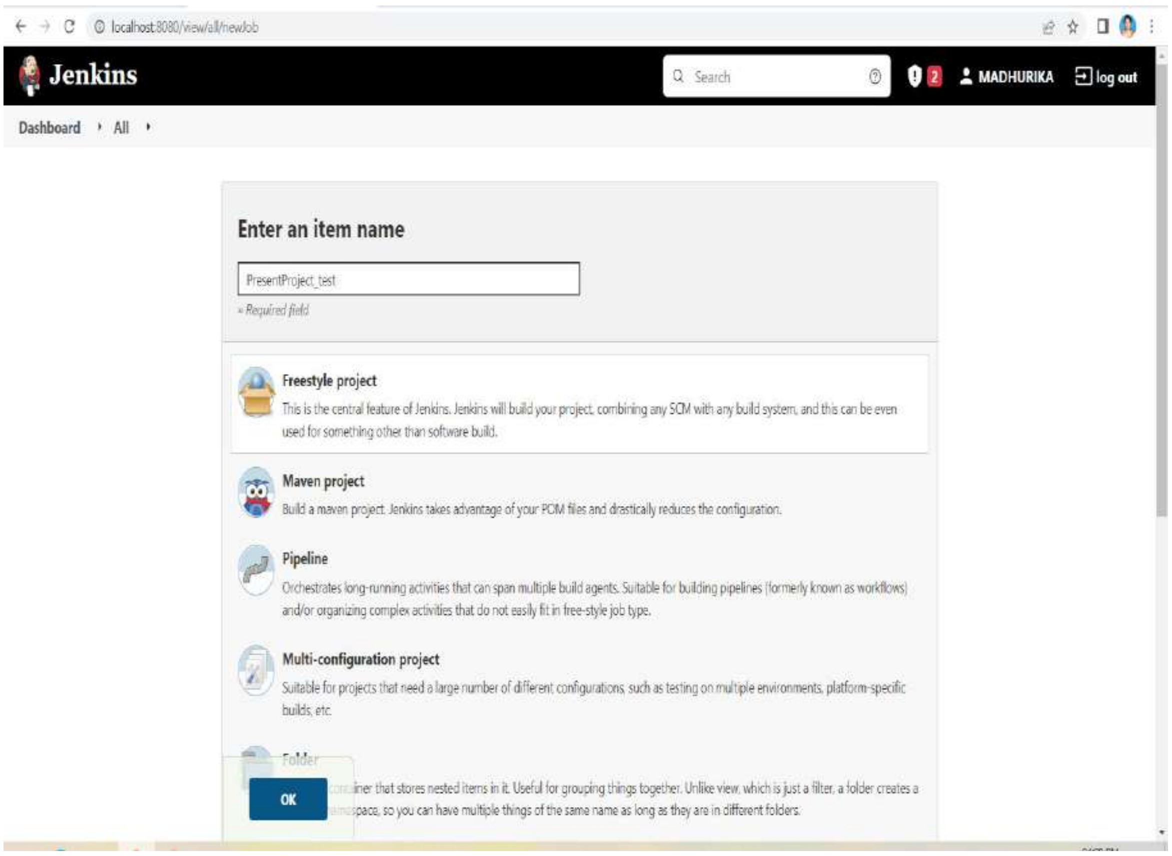
Build other projects

Projects to build: PresentProject_test

No such project 'PresentProject_test'. Did you mean 'PresentProject_build'?

- (●) Trigger only if build is stable
- ( ) Trigger even if the build is unstable
- ( ) Trigger even if the build fails

16. Go to dashboard -> New item-> Freestyle Project, and then give next project name as **PresentProject_test**, then press on OK


Enter an item name

PresentProject_test

» Required field

Freestyle project
This is the central feature of Jenkins. Jenkins will build your project, combining any SCM with any build system, and this can be even used for something other than software build.

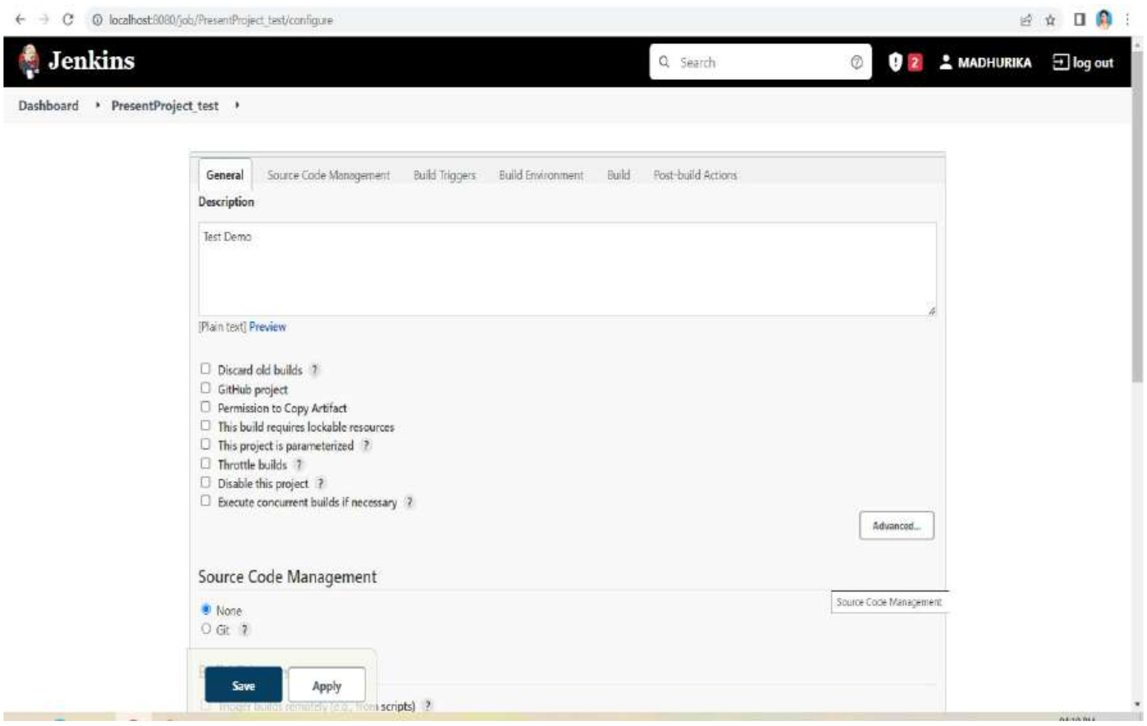
Maven project
Build a maven project. Jenkins takes advantage of your POM files and drastically reduces the configuration.

Pipeline
Orchestrates long-running activities that can span multiple build agents. Suitable for building pipelines (formerly known as workflows) and/or organizing complex activities that do not easily fit in free-style job type.

Multi-configuration project
Suitable for projects that need a large number of different configurations, such as testing on multiple environments, platform-specific builds, etc.

Folder
...iner that stores nested items in it. Useful for grouping things together. Unlike view, which is just a filter, a folder creates a ...space, so you can have multiple things of the same name as long as they are in different folders.

17. In description type eg. Test demo


localhost:8080/job/PresentProject_test/configure

Jenkins

Dashboard › PresentProject_test ›

General | Source Code Management | Build Triggers | Build Environment | Build | Post-build Actions

Description

Test Demo

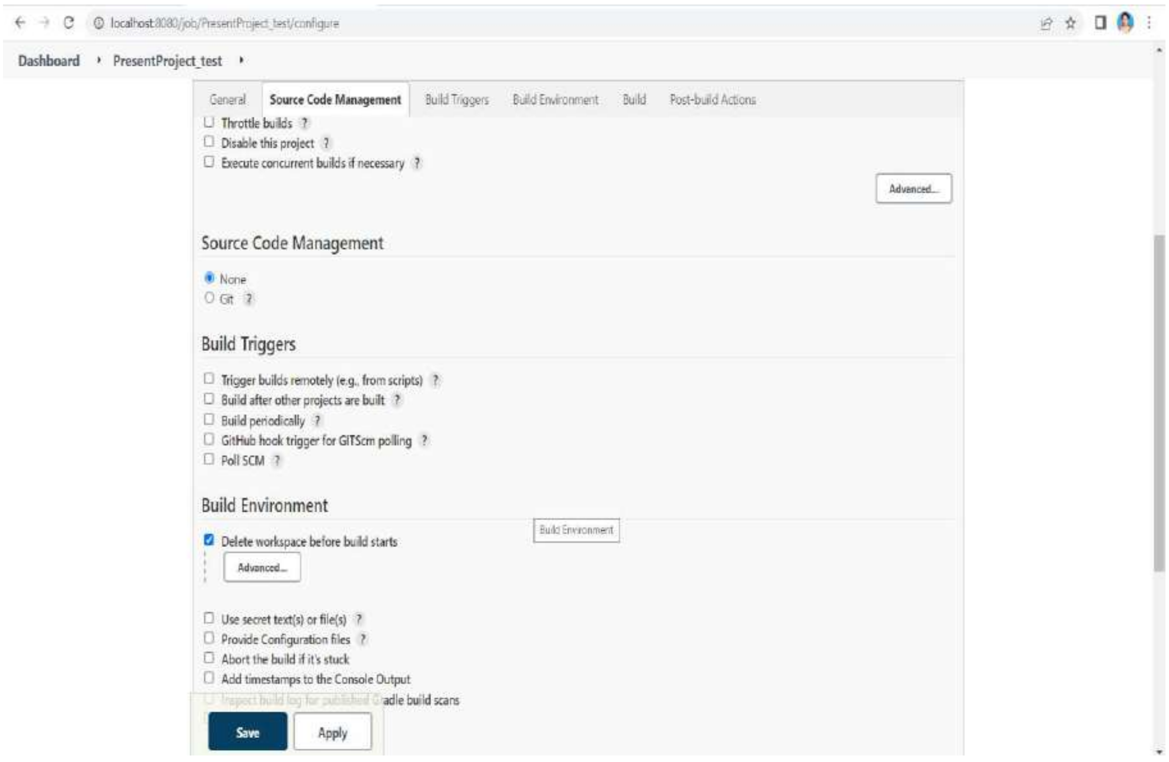
[Plain text] Preview

- [ ] Discard old builds
- [ ] GitHub project
- [ ] Permission to Copy Artifact
- [ ] This build requires lockable resources
- [ ] This project is parameterized
- [ ] Throttle builds
- [ ] Disable this project
- [ ] Execute concurrent builds if necessary

Source Code Management

- (●) None
- ( ) Git

18. In Build environment, check the box with name "Delete the workspace before build starts". This is to discard old builds


localhost:8080/job/PresentProject_test/configure

Dashboard › PresentProject_test ›

General | Source Code Management | Build Triggers | Build Environment | Build | Post-build Actions

- [ ] Throttle builds
- [ ] Disable this project
- [ ] Execute concurrent builds if necessary

Source Code Management

- (●) None
- ( ) Git

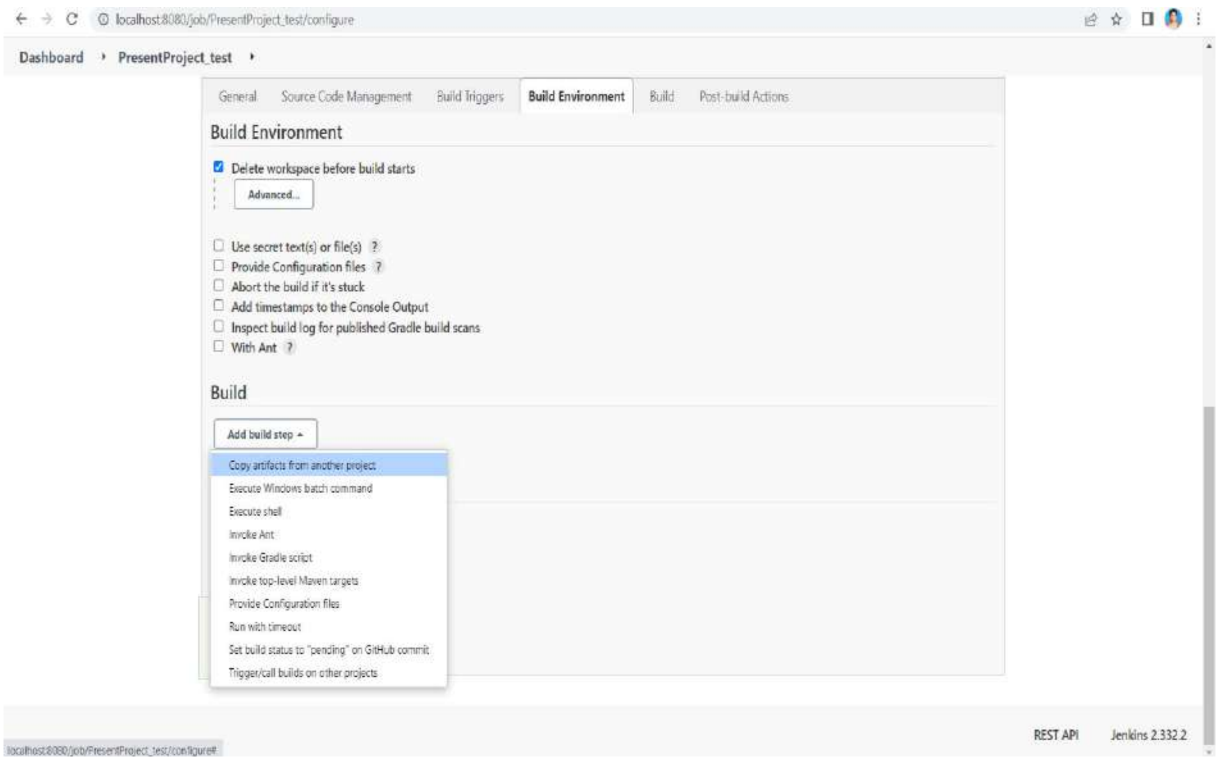
Build Triggers

- [ ] Trigger builds remotely (e.g., from scripts)
- [ ] Build after other projects are built
- [ ] Build periodically
- [ ] GitHub hook trigger for GITScm polling
- [ ] Poll SCM

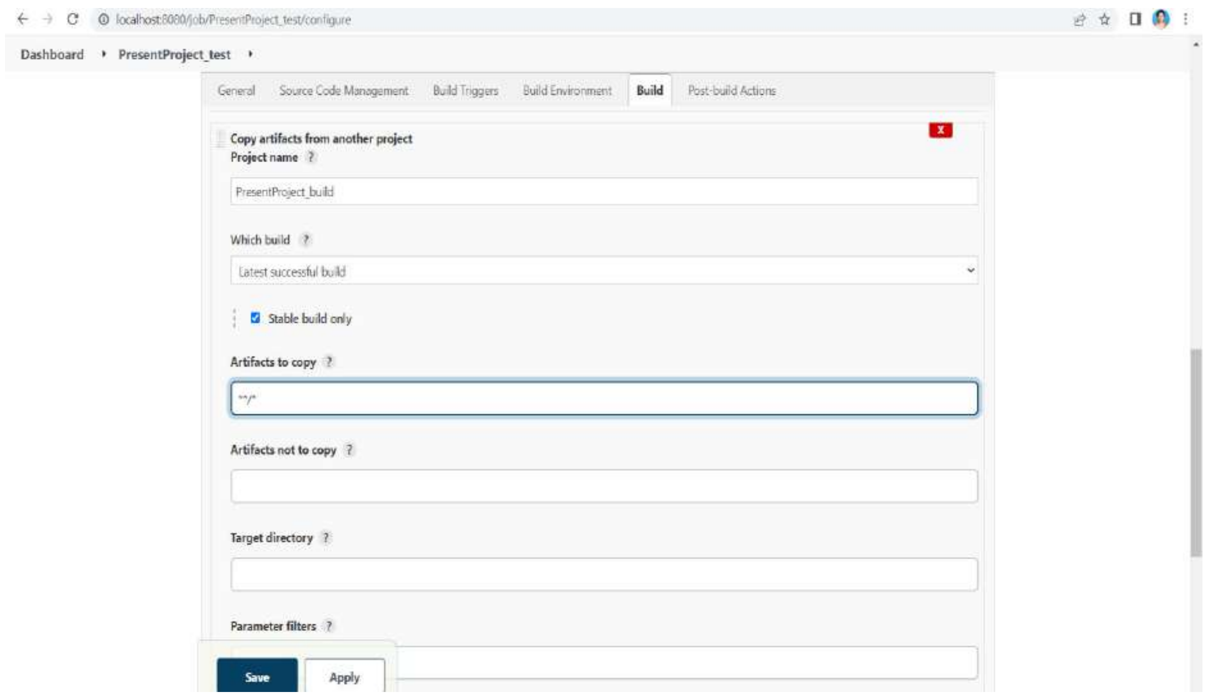
Build Environment

- [✓] Delete workspace before build starts
- [ ] Use secret text(s) or file(s)
- [ ] Provide Configuration files
- [ ] Abort the build if it's stuck
- [ ] Add timestamps to the Console Output
- [ ] Inspect build log for published Gradle build scans

19. In Build select "**copy the artifacts from another project**" to forward the artifacts of the previous project to the current test project i.e., PresentProject_test


localhost:8080/job/PresentProject_test/configure
Dashboard › PresentProject_test ›
General
Source Code Management
Build Triggers
Build Environment
Build
Post-build Actions
Build Environment
☑ Delete workspace before build starts
Advanced...
☐ Use secret text(s) or file(s)
☐ Provide Configuration files
☐ Abort the build if it's stuck
☐ Add timestamps to the Console Output
☐ Inspect build log for published Gradle build scans
☐ With Ant
Build
Add build step
Copy artifacts from another project
Execute Windows batch command
Execute shell
Invoke Ant
Invoke Gradle script
Invoke top-level Maven targets
Provide Configuration files
Run with timeout
Set build status to "pending" on GitHub commit
Trigger/call builds on other projects
REST API
Jenkins 2.332.2

20. Give the name of the project from which we want to copy the artifacts (eg. PresentProject_build) and check the box ->stable build only->to copy all the artifacts type **/*


localhost:8080/job/PresentProject_test/configure
Dashboard › PresentProject_test ›
General
Source Code Management
Build Triggers
Build Environment
Build
Post-build Actions
Copy artifacts from another project
Save
Apply

Project name: PresentProject_build

Which build: Latest successful build

☑ Stable build only

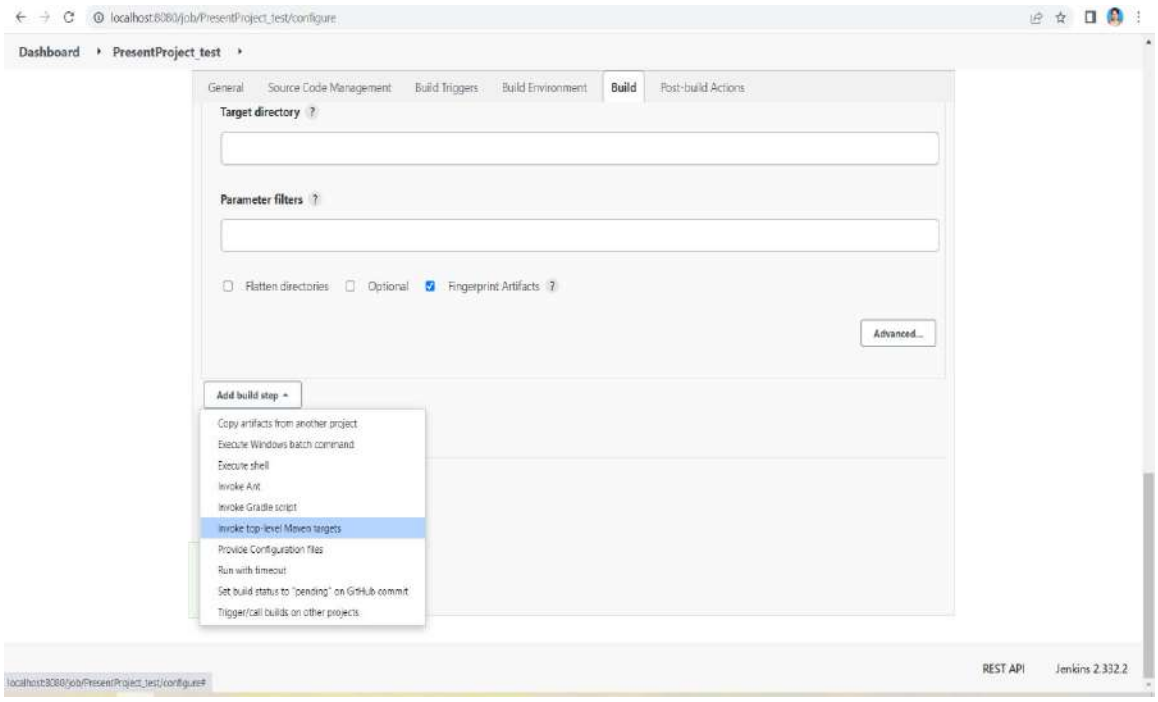
Artifacts to copy: **/*

Artifacts not to copy

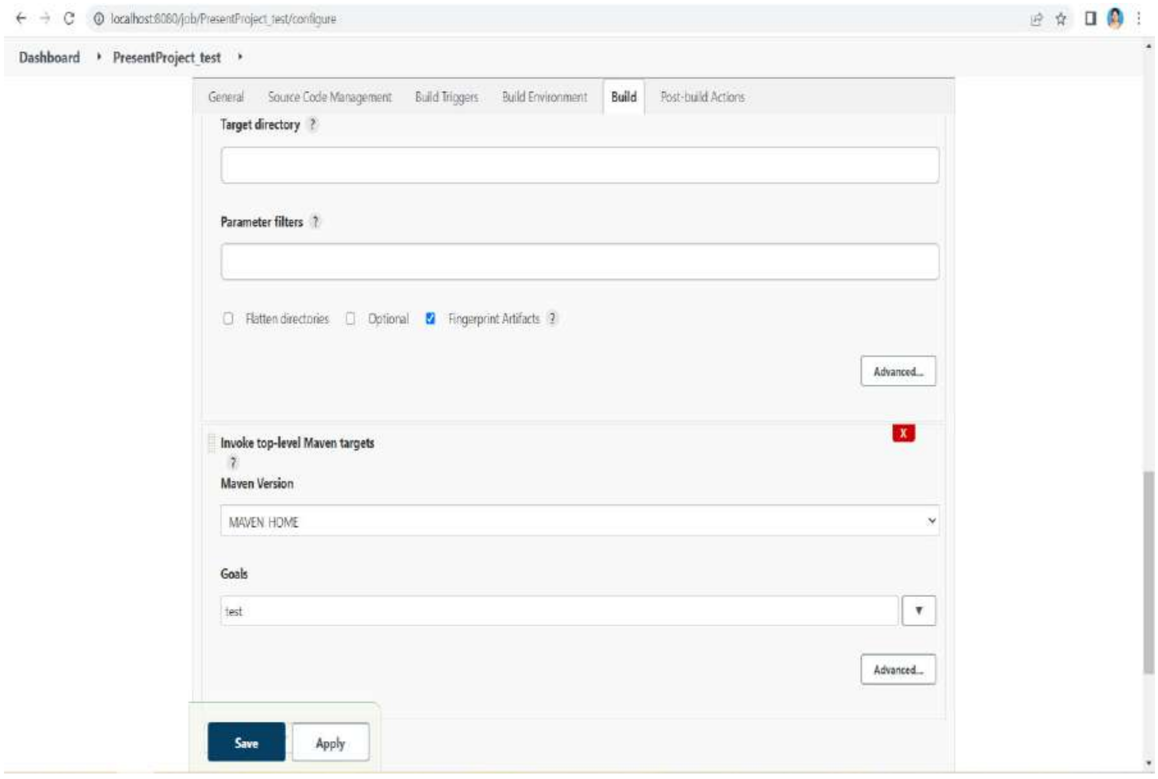
Target directory

Parameter filters

21. Now select Invoke top-level Maven targets in build

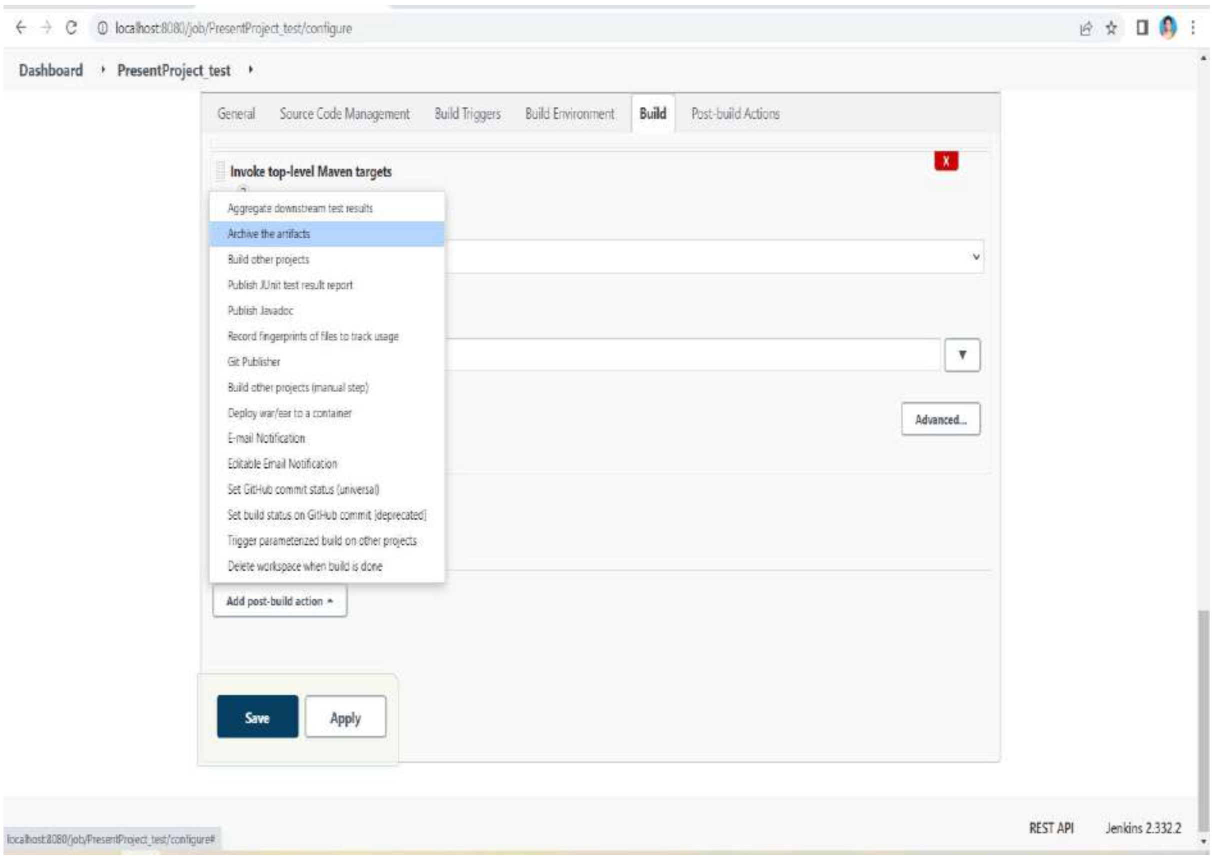


22. This time give the goal as test after selecting the Maven version

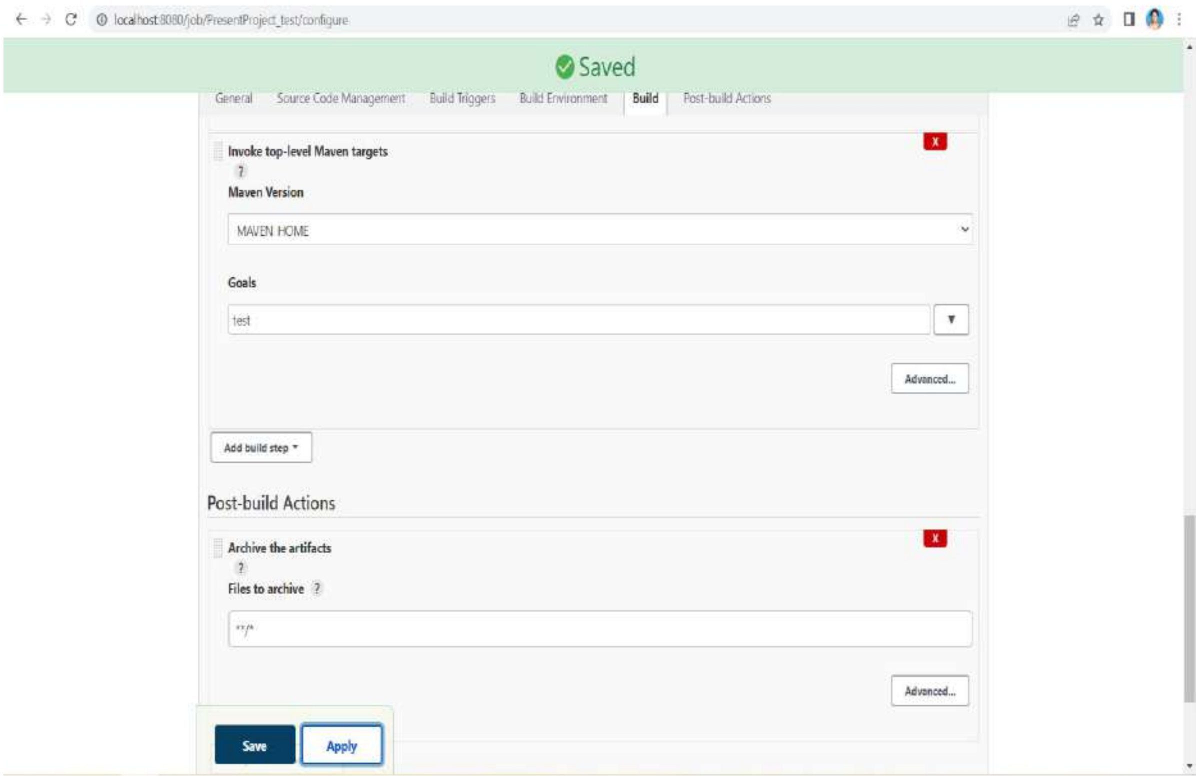


Maven Version: MAVEN HOME
Goals: test

23. In post-build actions->select Archive the artifacts


localhost:8080/job/PresentProject_test/configure
Dashboard
PresentProject_test
General
Source Code Management
Build Triggers
Build Environment
Build
Post-build Actions
Invoke top-level Maven targets
Aggregate downstream test results
Archive the artifacts
Build other projects
Publish JUnit test result report
Publish Javadoc
Record fingerprints of files to track usage
Git Publisher
Build other projects (manual step)
Deploy war/ear to a container
E-mail Notification
Editable Email Notification
Set GitHub commit status (universal)
Set build status on GitHub commit [deprecated]
Trigger parameterized build on other projects
Delete workspace when build is done
Add post-build action
Advanced...
Save
Apply
REST API
Jenkins 2.332.2

24. To save all the artifacts->type **/* and Apply->Save



Maven Version

MAVEN HOME

Goals

test

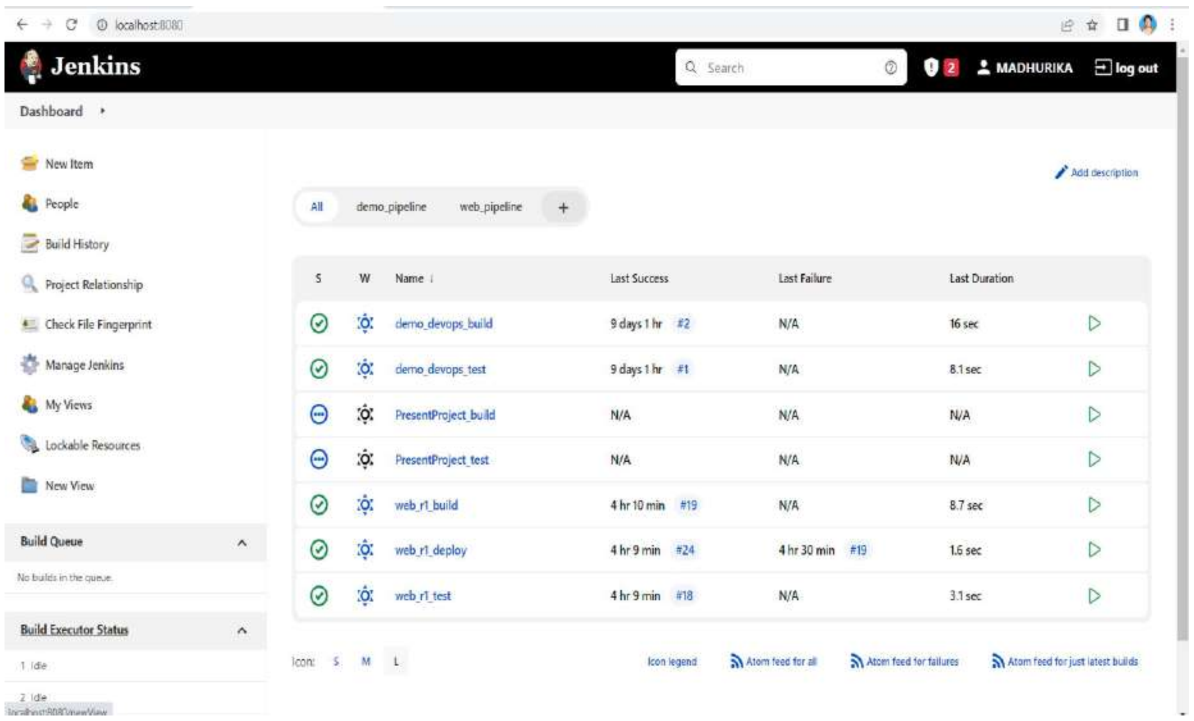
Post-build Actions

Archive the artifacts

Files to archive

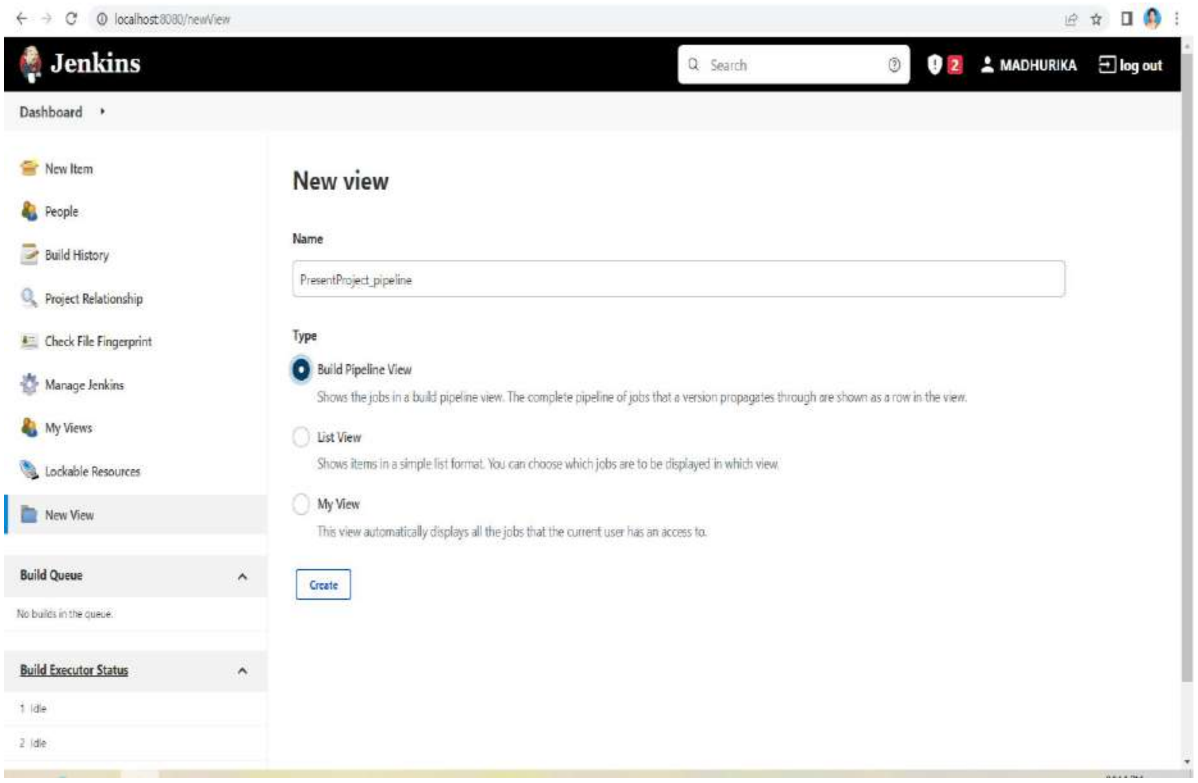
**/*

25. Create a pipeline by clicking on + symbol in the dashboard ->a pipeline is a collection of events or jobs which are interlinked with one another in a sequence



S	W	Name ↓	Last Success	Last Failure	Last Duration
		demo_devops_build	9 days 1 hr #2	N/A	16 sec
		demo_devops_test	9 days 1 hr #1	N/A	8.1 sec
		PresentProject_build	N/A	N/A	N/A
		PresentProject_test	N/A	N/A	N/A
		web_r1_build	4 hr 10 min #19	N/A	8.7 sec
		web_r1_deploy	4 hr 9 min #24	4 hr 30 min #19	1.6 sec
		web_r1_test	4 hr 9 min #18	N/A	3.1 sec

26. Give a name to the pipeline->select Build Pipeline View->create



New view

Name: PresentProject_pipeline

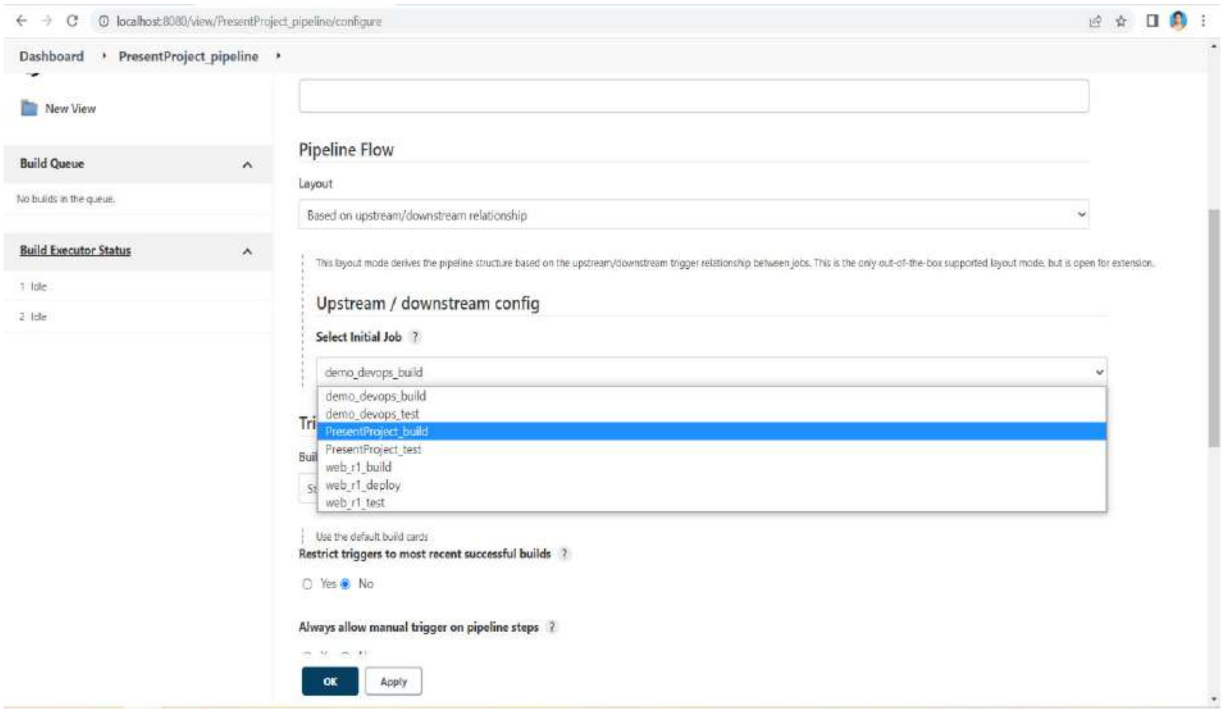
Type

[✓ Build Pipeline View]
Shows the jobs in a build pipeline view. The complete pipeline of jobs that a version propagates through are shown as a row in the view.

List View
Shows items in a simple list format. You can choose which jobs are to be displayed in which view.

My View
This view automatically displays all the jobs that the current user has an access to.

27. Select the first project to trigger the execution->build project


localhost:8080/view/PresentProject_pipeline/configure

Dashboard › PresentProject_pipeline ›

New View

Build Queue

No builds in the queue.

Build Executor Status

1 Idle

2 Idle

Pipeline Flow

Layout

Based on upstream/downstream relationship

This layout mode derives the pipeline structure based on the upstream/downstream trigger relationship between jobs. This is the only out-of-the-box supported layout mode, but is open for extension.

Upstream / downstream config

Select Initial Job ?

demo_devops_build

- demo_devops_build
- demo_devops_test
- PresentProject_build
- PresentProject_test
- web_r1_build
- web_r1_deploy
- web_r1_test

Use the default build cards

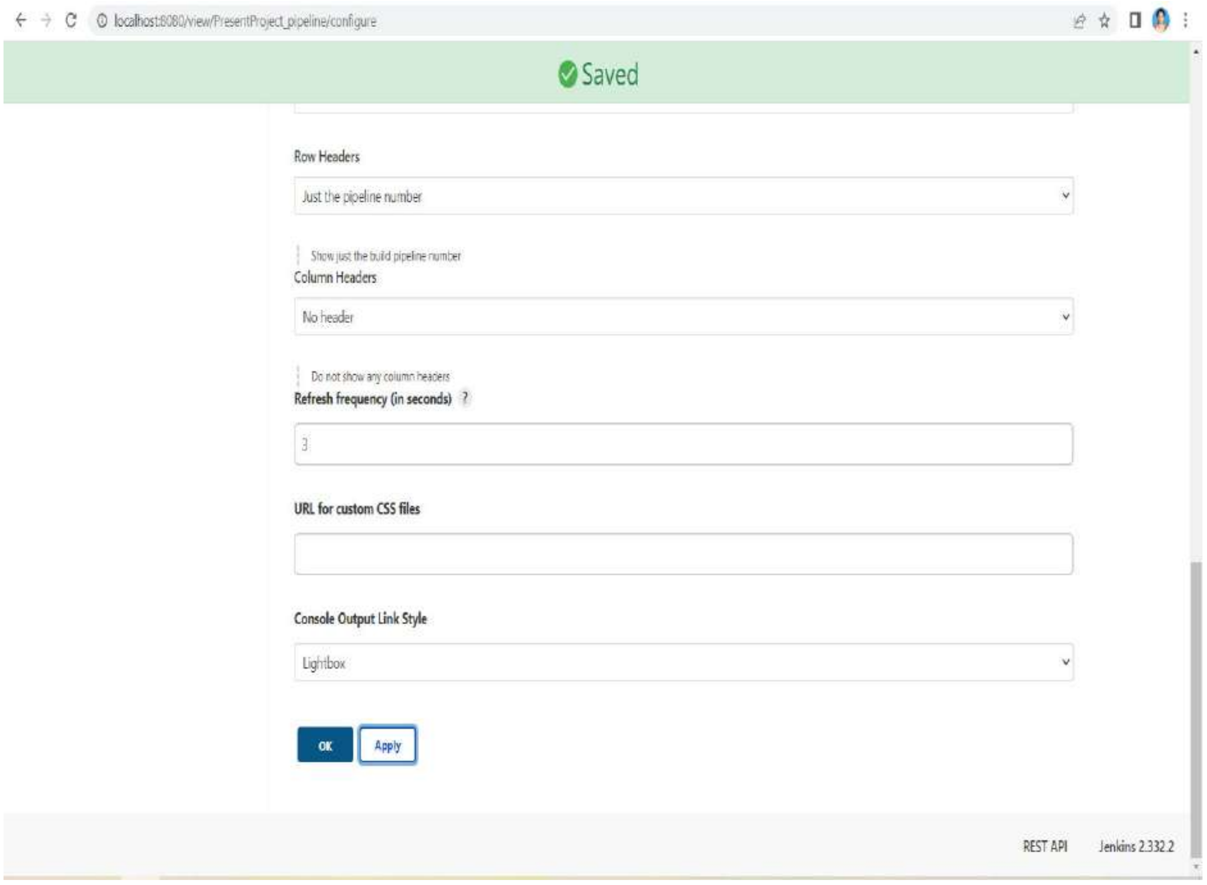
Restrict triggers to most recent successful builds ?

Yes [✓ No]

Always allow manual trigger on pipeline steps ?

OK Apply

28. Press on Apply->Ok


localhost:8080/view/PresentProject_pipeline/configure

Saved

Row Headers

Just the pipeline number

Show just the build pipeline number

Column Headers

No header

Do not show any column headers

Refresh frequency (in seconds) ?

3

URL for custom CSS files

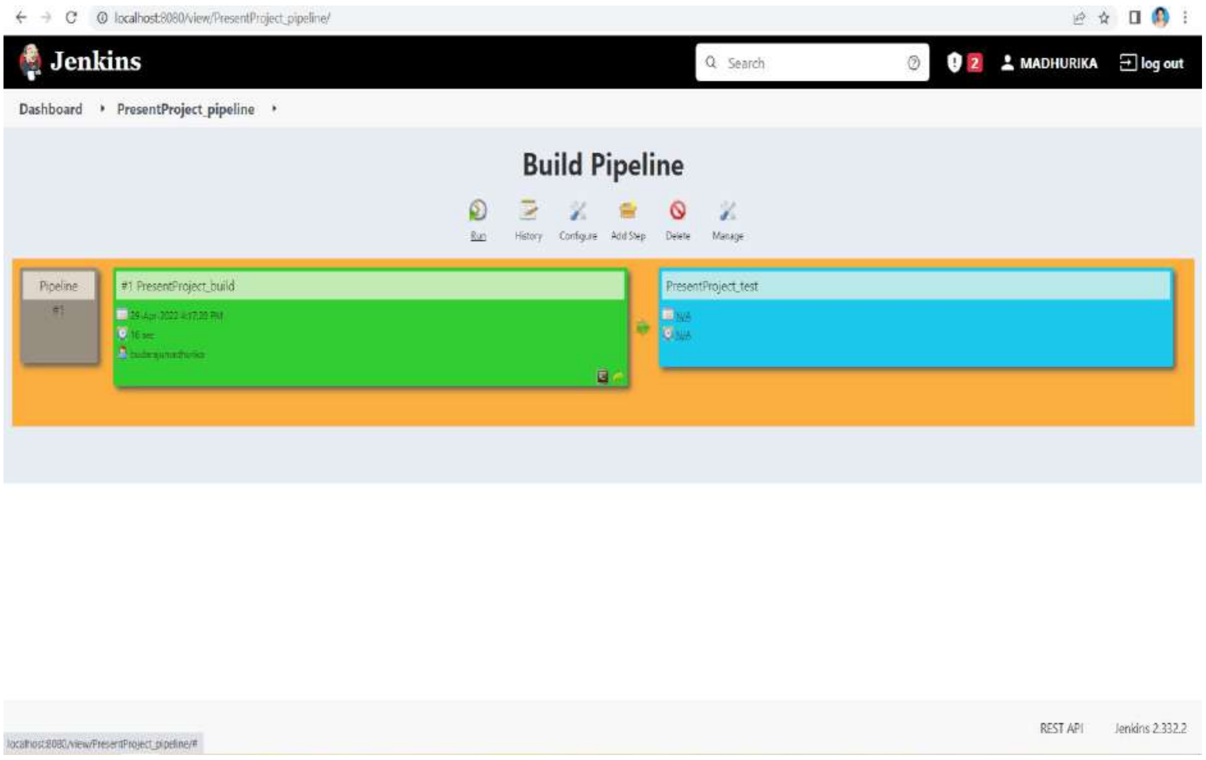
Console Output Link Style

Lightbox

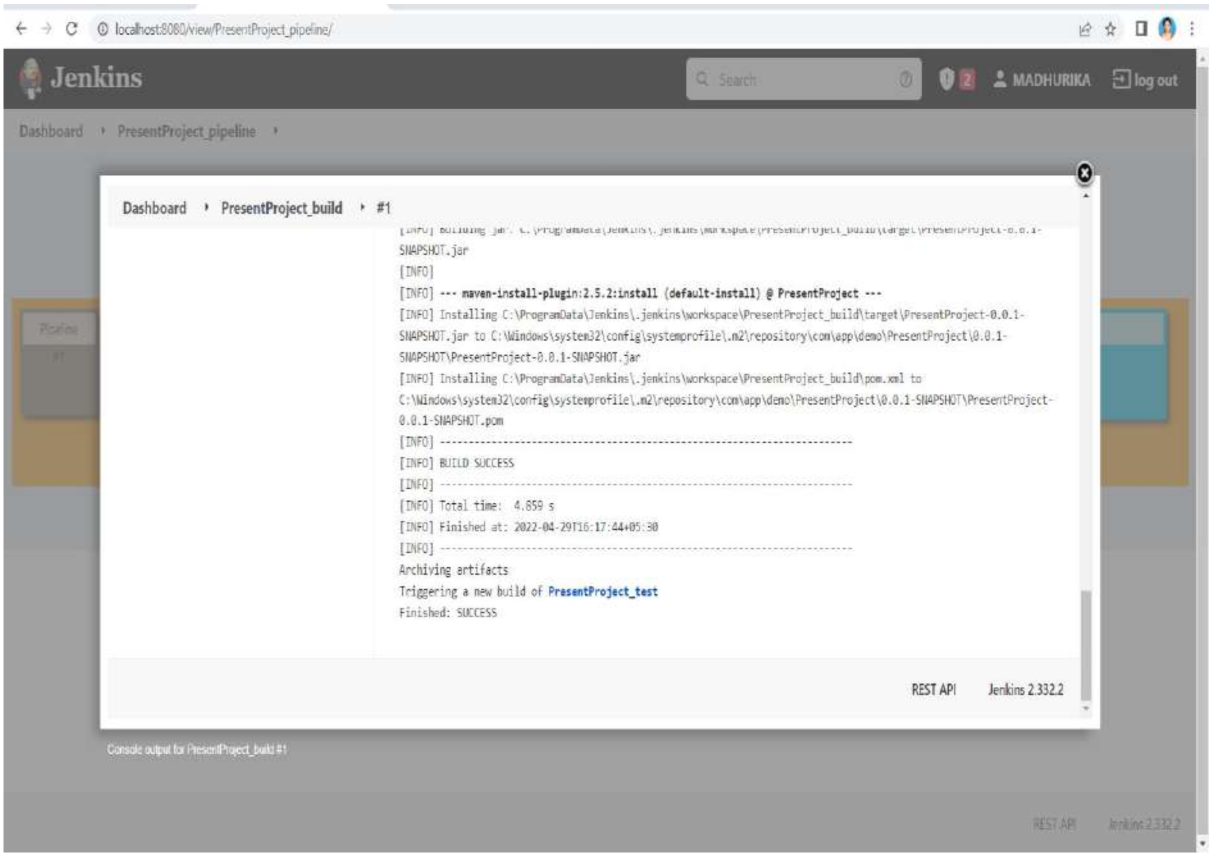
OK Apply

REST API Jenkins 2.332.2

29. Click on Run -> click on the small black box to open the console to check if the build is success

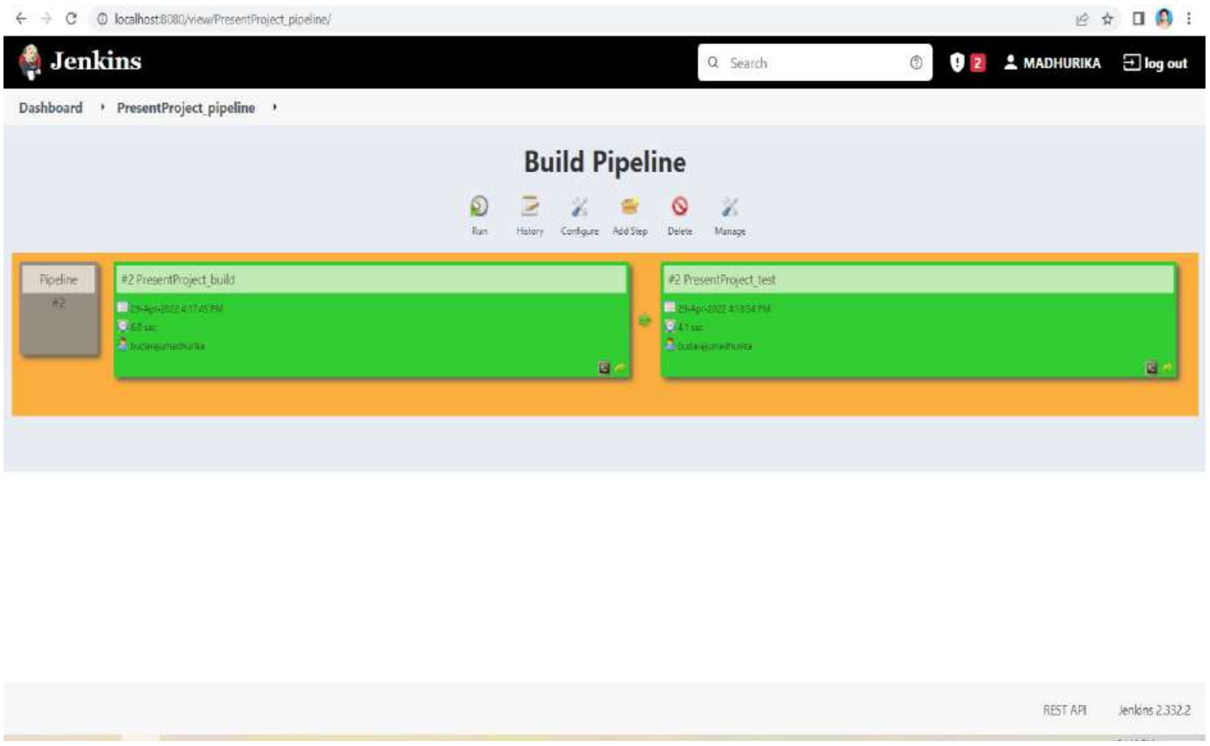


30. We can see that the build is success and the test project is also automatically triggered

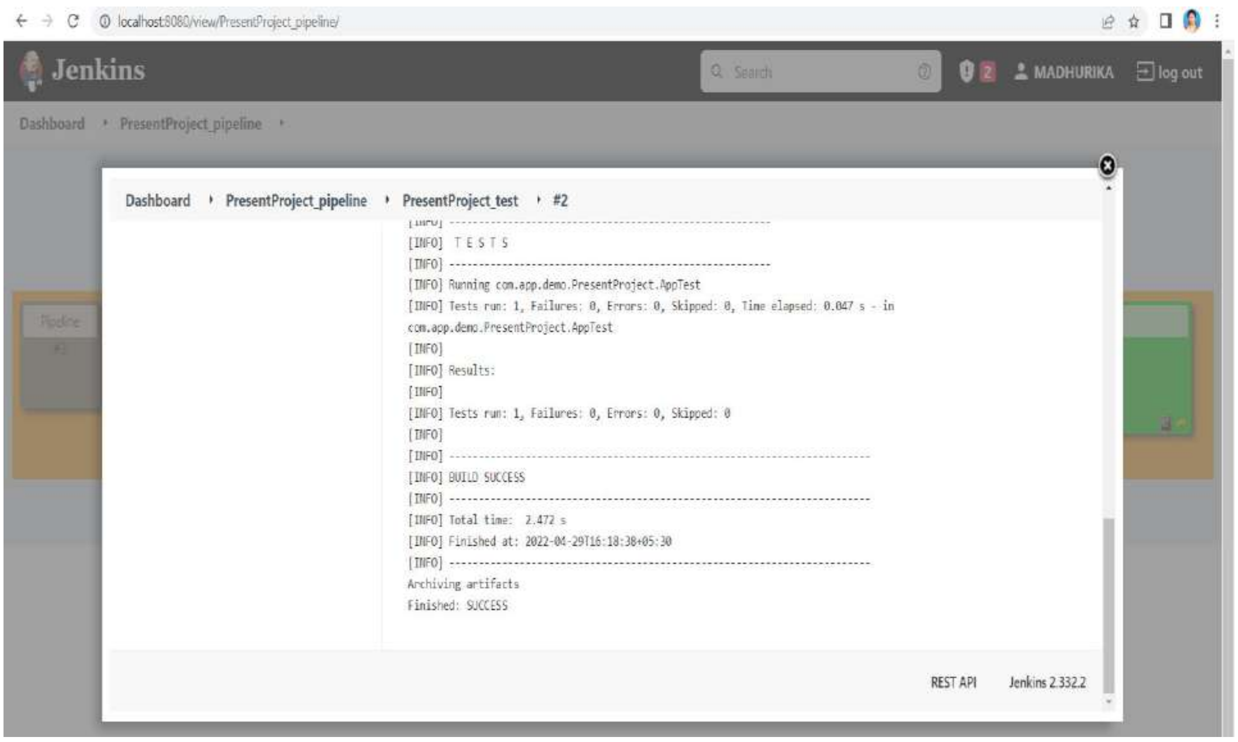


```
[INFO] Building jar: C:\ProgramData\Jenkins\.jenkins\workspace\PresentProject_build\target\PresentProject-0.0.1-
SNAPSHOT.jar
[INFO]
[INFO] --- maven-install-plugin:2.5.2:install (default-install) @ PresentProject ---
[INFO] Installing C:\ProgramData\Jenkins\.jenkins\workspace\PresentProject_build\target\PresentProject-0.0.1-
SNAPSHOT.jar to C:\Windows\system32\config\systemprofile\.m2\repository\com\app\demo\PresentProject\0.0.1-
SNAPSHOT\PresentProject-0.0.1-SNAPSHOT.jar
[INFO] Installing C:\ProgramData\Jenkins\.jenkins\workspace\PresentProject_build\pom.xml to
C:\Windows\system32\config\systemprofile\.m2\repository\com\app\demo\PresentProject\0.0.1-SNAPSHOT\PresentProject-
0.0.1-SNAPSHOT.pom
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time:  4.859 s
[INFO] Finished at: 2022-04-29T16:17:44+05:30
[INFO] ------------------------------------------------------------------------
Archiving artifacts
Triggering a new build of PresentProject_test
Finished: SUCCESS
```

31. The pipeline is successful if it is in green color as shown ->check the console of the test project



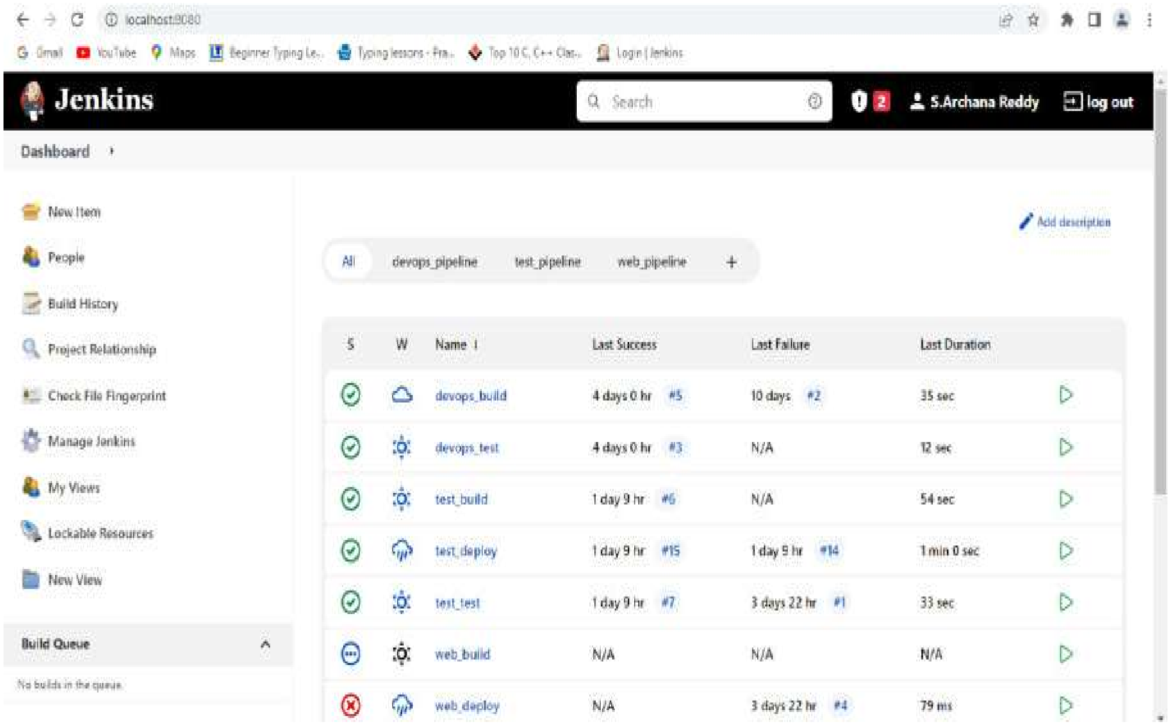
32. The test project is successful and all the artifacts are archived successfully



```
[INFO] -------------------------------------------------------
[INFO]  T E S T S
[INFO] -------------------------------------------------------
[INFO] Running com.app.demo.PresentProject.AppTest
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.047 s - in
com.app.demo.PresentProject.AppTest
[INFO]
[INFO] Results:
[INFO]
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time:  2.472 s
[INFO] Finished at: 2022-04-29T16:18:38+05:30
[INFO] ------------------------------------------------------------------------
Archiving artifacts
Finished: SUCCESS
```

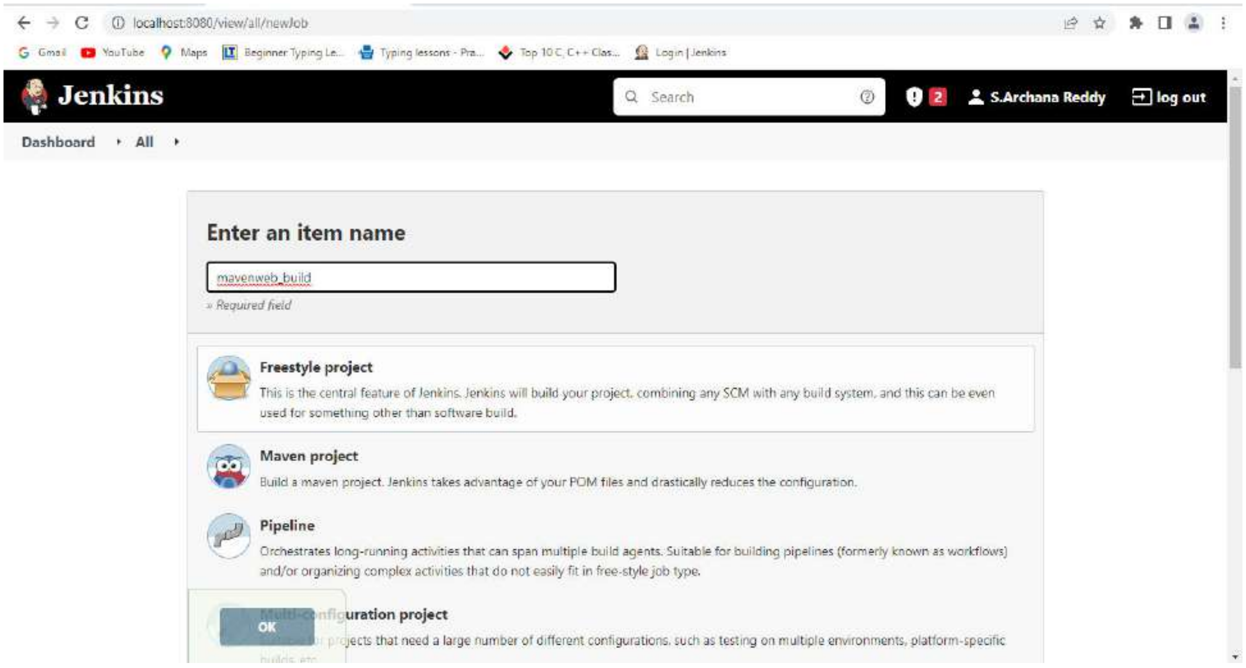
III. Creating a Simple Pipeline using user interface for Web Project

1. Open Jenkins in local host:8080 and **create the new item**



S	W	Name ↓	Last Success	Last Failure	Last Duration
✓	☁	devops_build	4 days 0 hr #5	10 days #2	35 sec
✓	☼	devops_test	4 days 0 hr #3	N/A	12 sec
✓	☼	test_build	1 day 9 hr #6	N/A	54 sec
✓	☂	test_deploy	1 day 9 hr #15	1 day 9 hr #14	1 min 0 sec
✓	☼	test_test	1 day 9 hr #7	3 days 22 hr #1	33 sec
⋯	☼	web_build	N/A	N/A	N/A
✗	☂	web_deploy	N/A	3 days 22 hr #4	79 ms

2. Select a new Freestyle Project give name (eg. mavenweb_build) and then click ok
3. In description type eg. Build demo



Enter an item name

mavenweb_build

» Required field

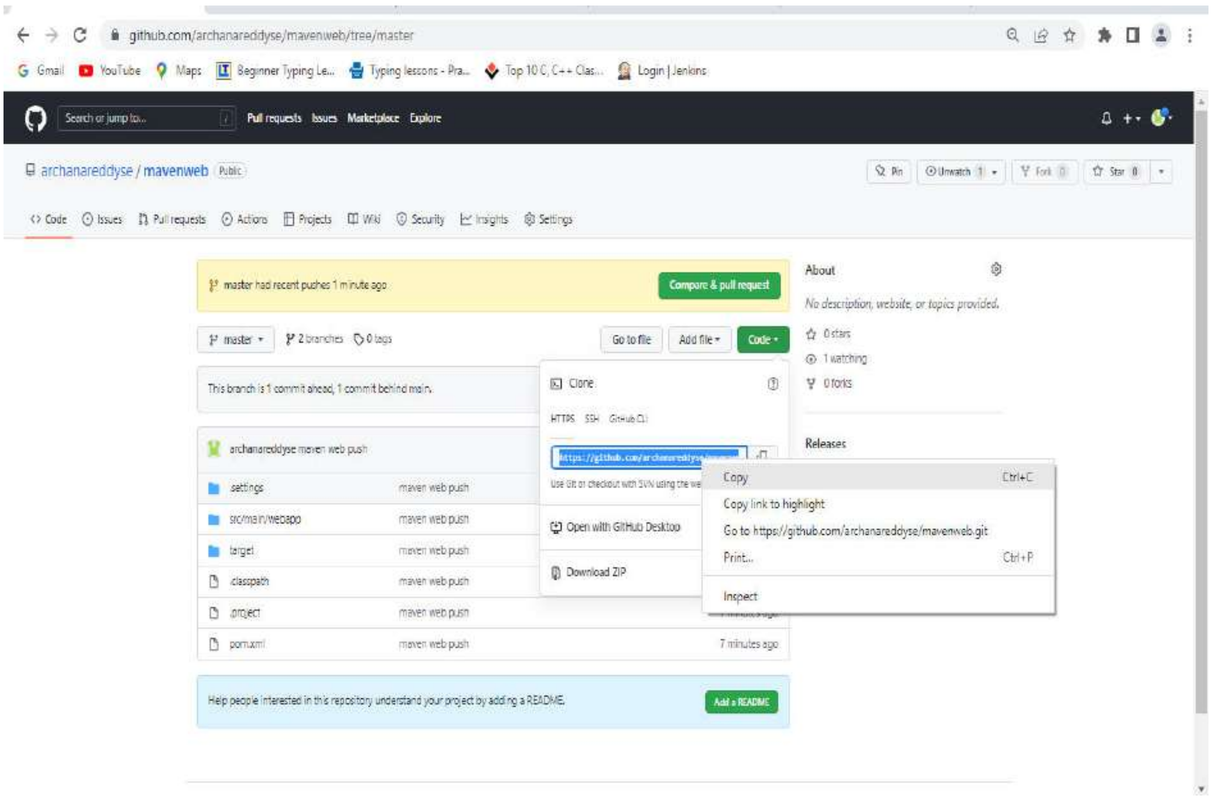
Freestyle project
This is the central feature of Jenkins. Jenkins will build your project, combining any SCM with any build system, and this can be even used for something other than software build.

Maven project
Build a maven project. Jenkins takes advantage of your POM files and drastically reduces the configuration.

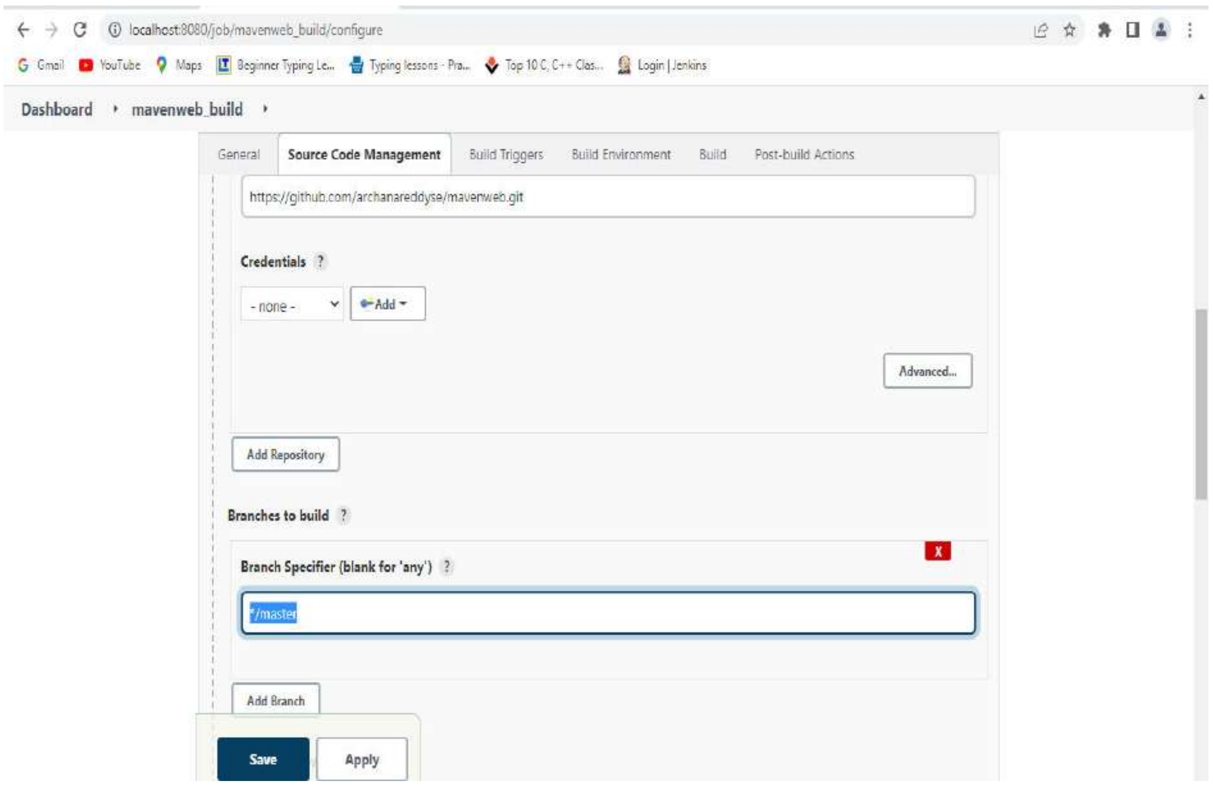
Pipeline
Orchestrates long-running activities that can span multiple build agents. Suitable for building pipelines (formerly known as workflows) and/or organizing complex activities that do not easily fit in free-style job type.

Multi-configuration project
Suitable for projects that need a large number of different configurations, such as testing on multiple environments, platform-specific builds, etc.

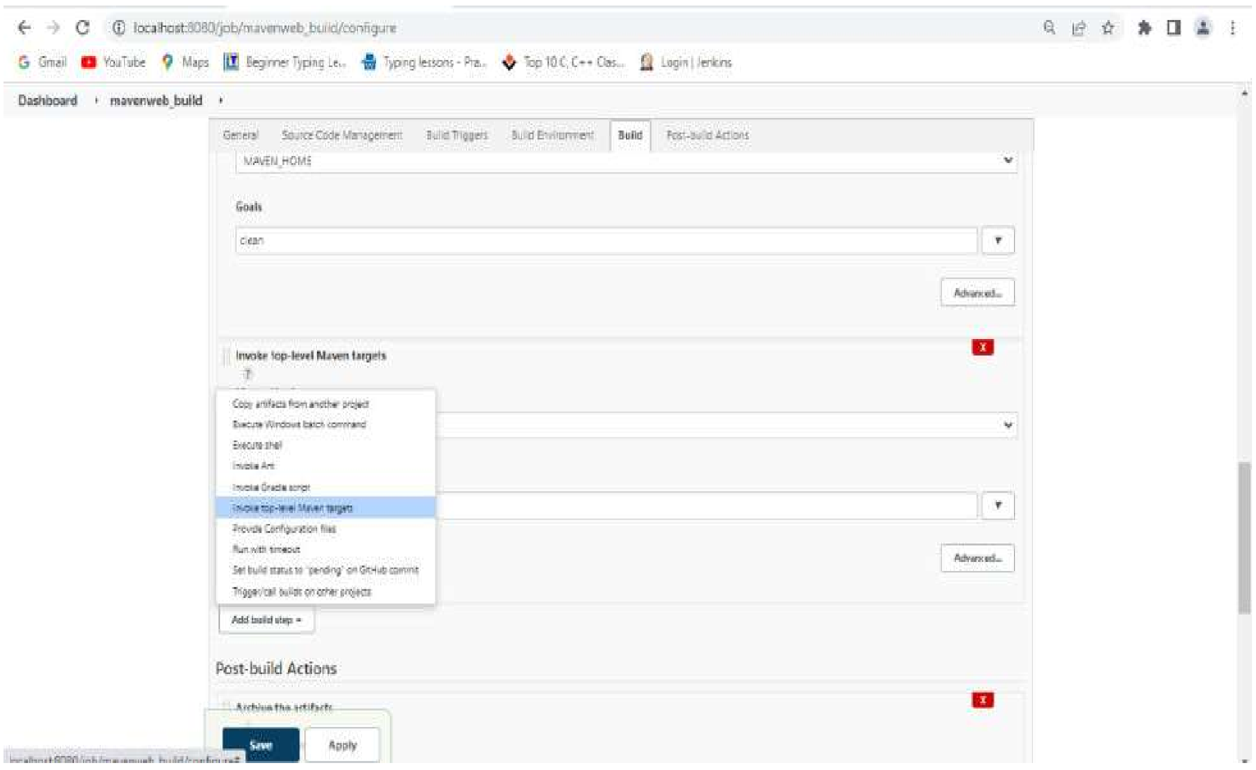
4. Copy the git hub http code


github.com/archanareddyse/mavenweb/tree/master
archanareddyse / mavenweb Public
master had recent pushes 1 minute ago
Compare & pull request
Code
Clone
HTTPS SSH GitHub CLI
https://github.com/archanareddyse/maven...
Copy Ctrl+C
Copy link to highlight
Go to https://github.com/archanareddyse/mavenweb.git
Print... Ctrl+P
Inspect
Open with GitHub Desktop
Download ZIP

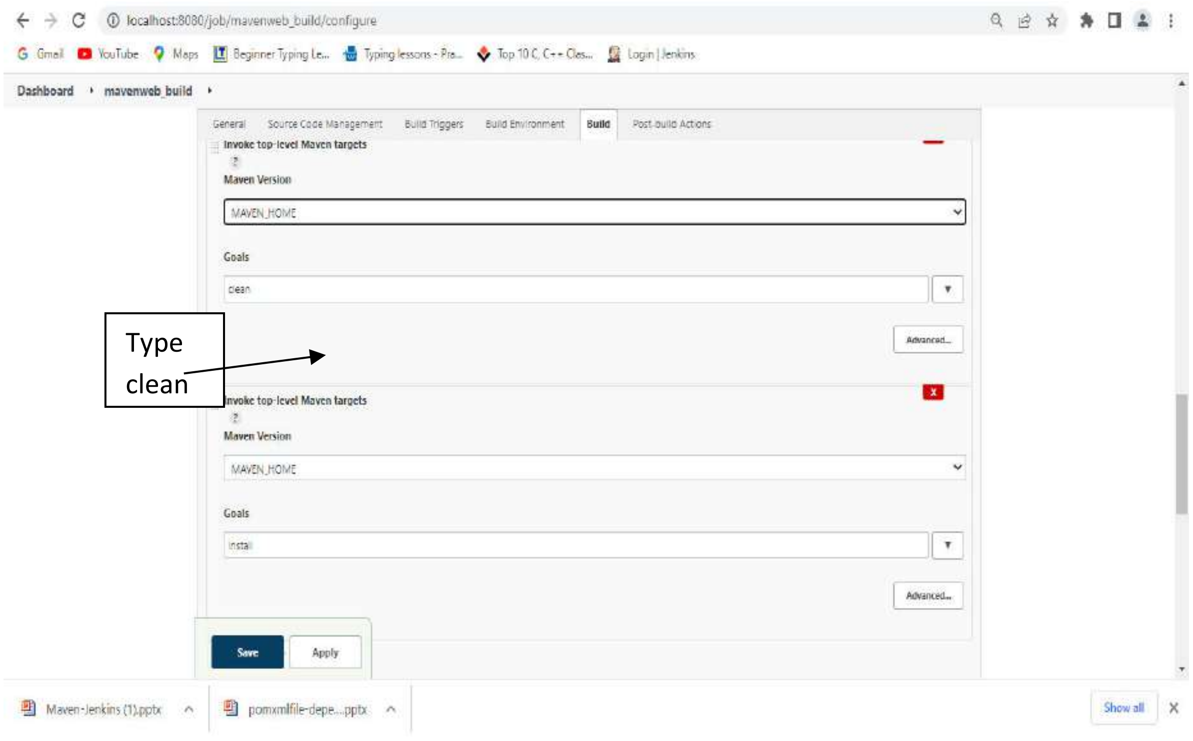
5. Paste the code in GIT URL, check for */master or */main.



6. Now in Build, Invoke top-level Maven targets

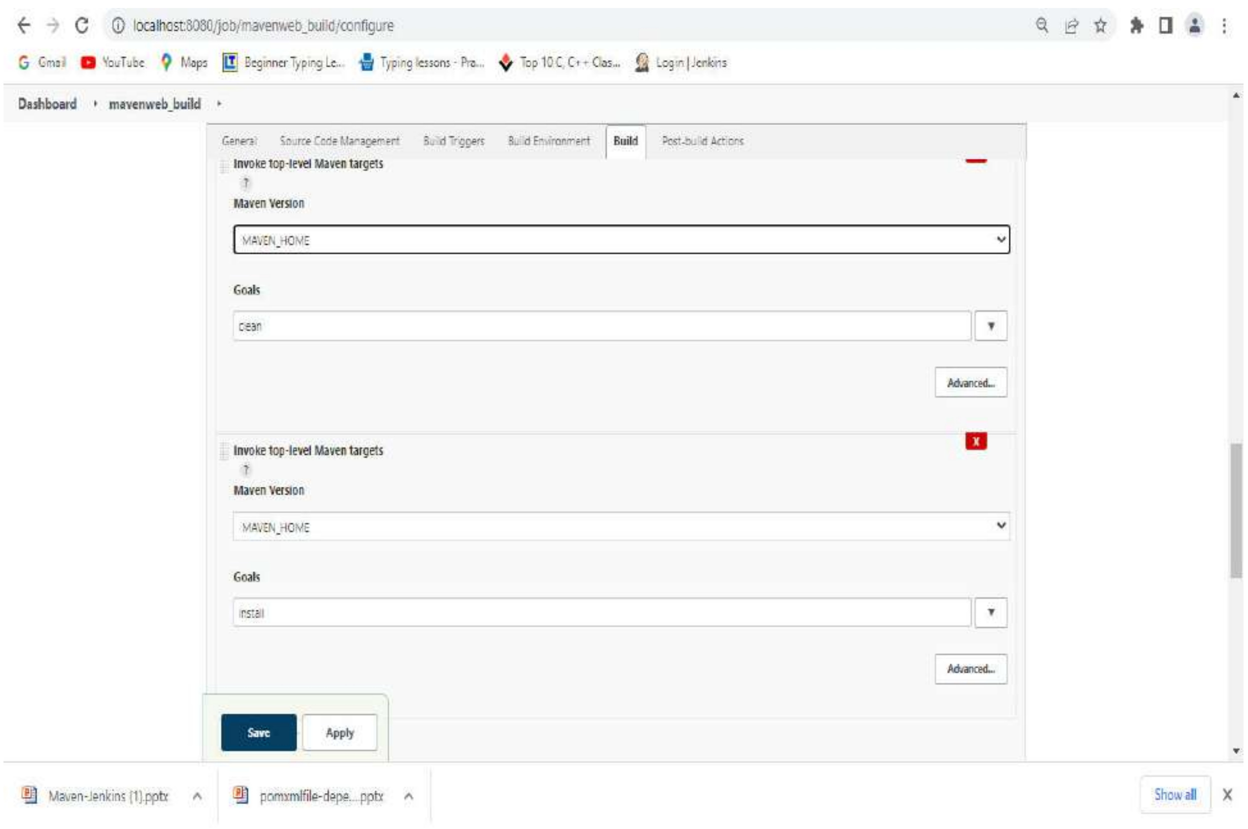

localhost:8080/job/mavenweb_build/configure
Dashboard › mavenweb_build ›
General
Source Code Management
Build Triggers
Build Environment
Build
Post-build Actions
MAVEN_HOME
Goals
clean
Advanced...
Invoke top-level Maven targets
Copy artifacts from another project
Execute Windows batch command
Execute shell
Invoke Ant
Invoke Gradle script
Invoke top-level Maven targets
Provide Configuration files
Run with timeout
Set build status to "pending" on GitHub commit
Trigger/call builds on other projects
Add build step
Post-build Actions
Archive the artifacts
Save
Apply

7. Select the Maven path which is already set in the global credentials in Manage Jenkins
8. Set Goals field to clean as done in eclipse



9. Again, in Build, select the "Invoke top-level Maven targets"

10. Select the Maven path which is already set in the global credentials in Manage Jenkins (eg MAVEN_HOME)
11. Set Goals field to **Install** as done in eclipse


localhost:8080/job/mavenweb_build/configure

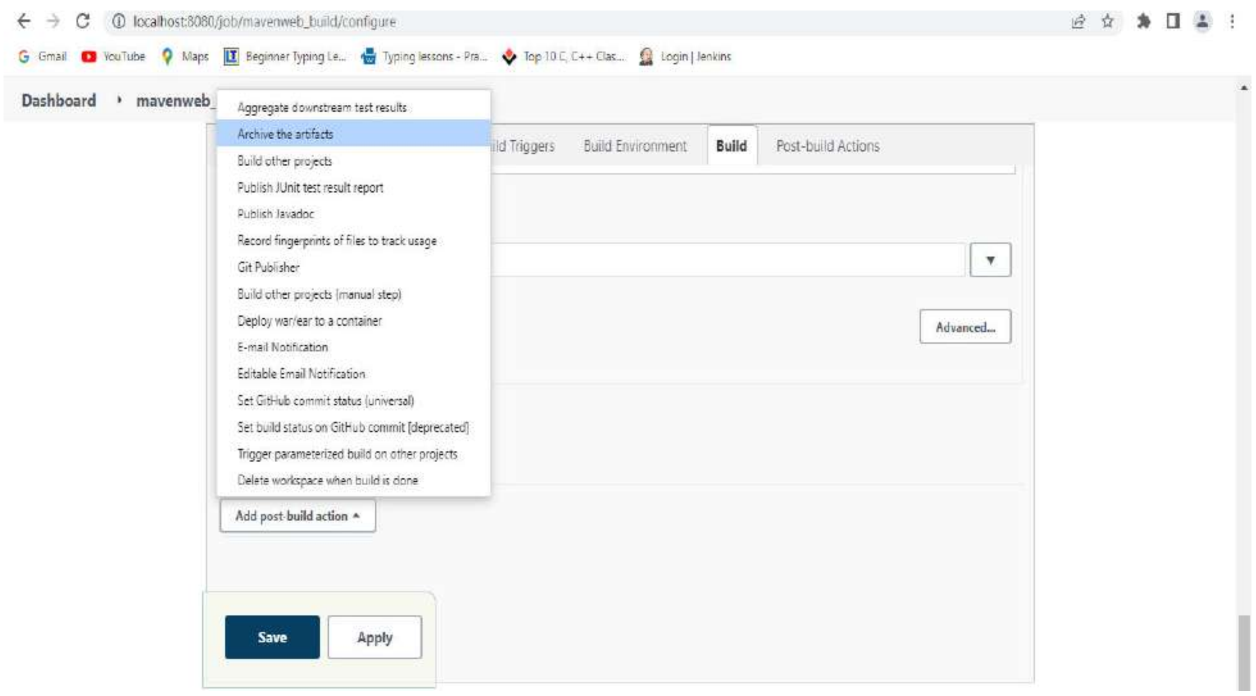
Dashboard › mavenweb_build ›

General | Source Code Management | Build Triggers | Build Environment | Build | Post-build Actions

Invoke top-level Maven targets
Maven Version: MAVEN_HOME
Goals: clean

Invoke top-level Maven targets
Maven Version: MAVEN_HOME
Goals: install

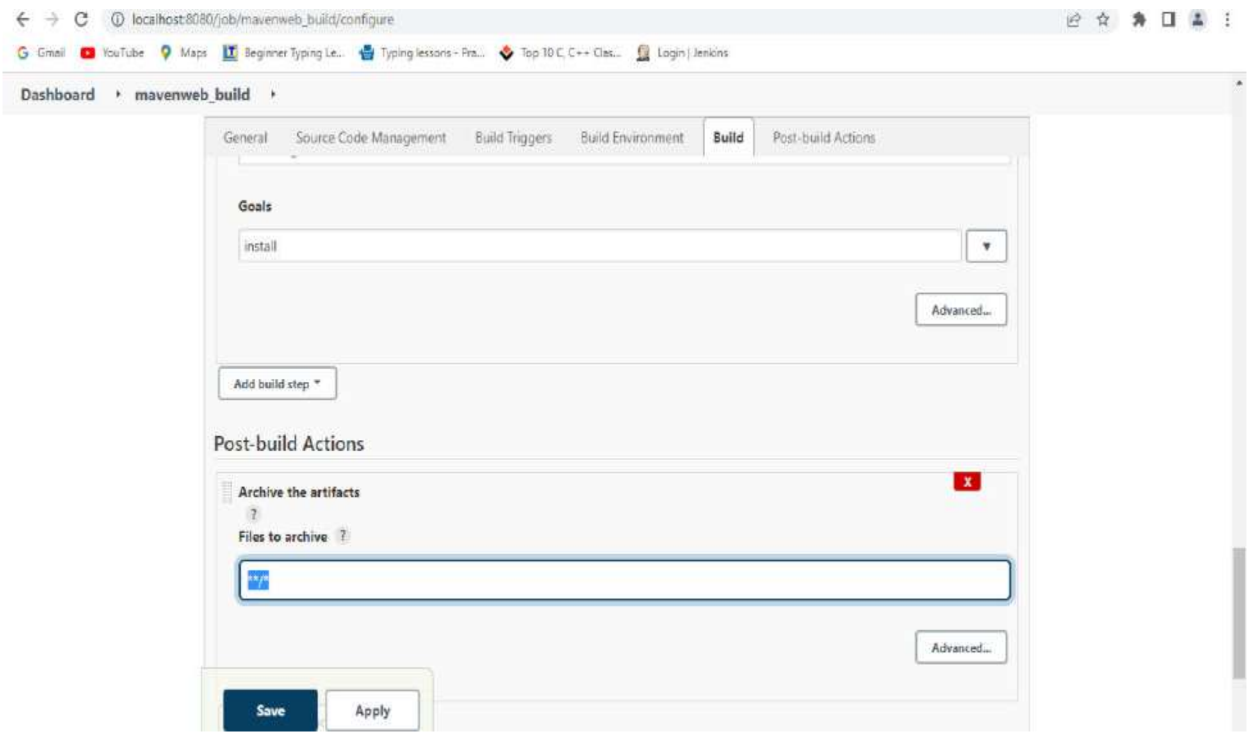
12. Now in post build actions-> select "**Archive the artifacts**", to send the output of build project to the testing team


localhost:8080/job/mavenweb_build/configure

Dashboard › mavenweb_

Aggregate downstream test results
Archive the artifacts
Build other projects
Publish JUnit test result report
Publish Javadoc
Record fingerprints of files to track usage
Git Publisher
Build other projects (manual step)
Deploy war/ear to a container
E-mail Notification
Editable Email Notification
Set GitHub commit status (universal)
Set build status on GitHub commit [deprecated]
Trigger parameterized build on other projects
Delete workspace when build is done
Add post-build action
Build Triggers
Build Environment
Build
Post-build Actions
Advanced...
Save
Apply

13. If we want to archive all the artifacts type **/* in Files to Archive



localhost:8080/job/mavenweb_build/configure

Dashboard › mavenweb_build ›

General | Source Code Management | Build Triggers | Build Environment | Build | Post-build Actions

Goals

install

Post-build Actions

Archive the artifacts

Files to archive

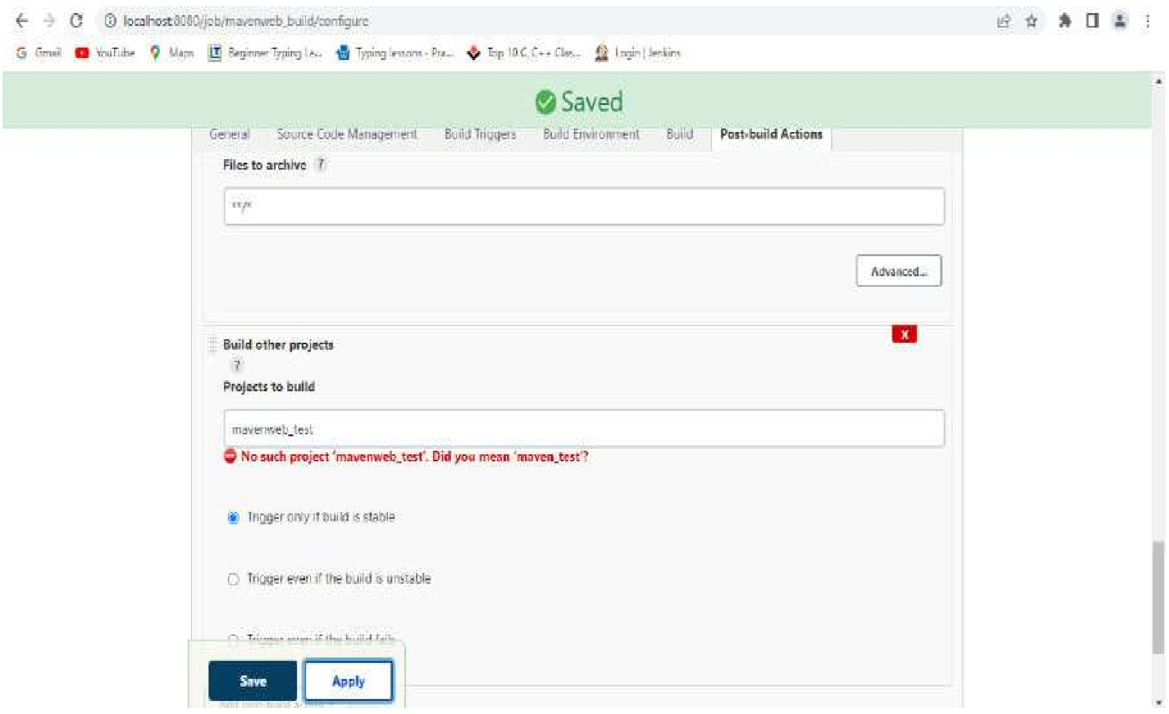
**/*

14. Now the next step is to build other projects, where we will create a test project which will be triggered by the build project.

    For this **in <u>Add Post build Action</u> select "Build other projects"**

15. In "Projects to build" enter the next project name as "**mavenweb_test**"

16. Next press on Apply and Save



localhost:8080/job/mavenweb_build/configure

Saved

General | Source Code Management | Build Triggers | Build Environment | Build | Post-build Actions

Files to archive

**/*

Build other projects

Projects to build

mavenweb_test

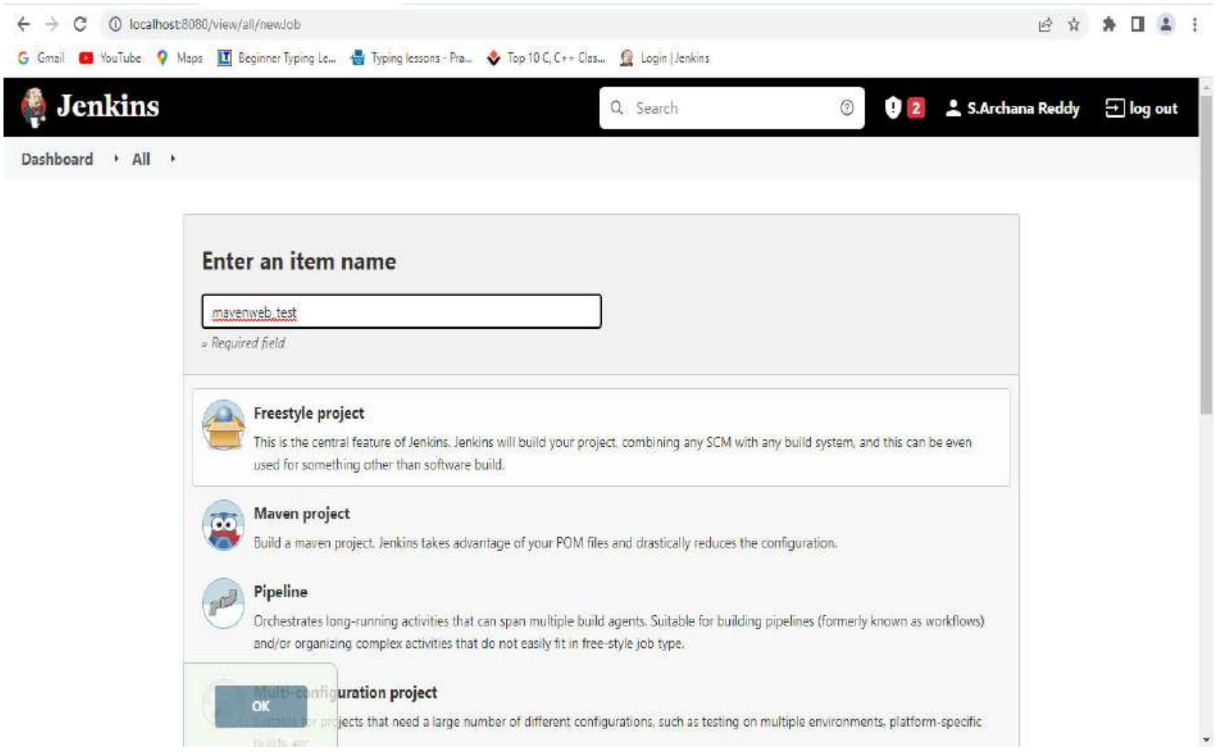
No such project 'mavenweb_test'. Did you mean 'maven_test'?

● Trigger only if build is stable

○ Trigger even if the build is unstable

○ Trigger even if the build fails

17. Go to dashboard -> New item-> Freestyle Project, and then give next project name as **MavenJavaProject_test**, then press on OK



Enter an item name

mavenweb_test

» Required field

Freestyle project

This is the central feature of Jenkins. Jenkins will build your project, combining any SCM with any build system, and this can be even used for something other than software build.

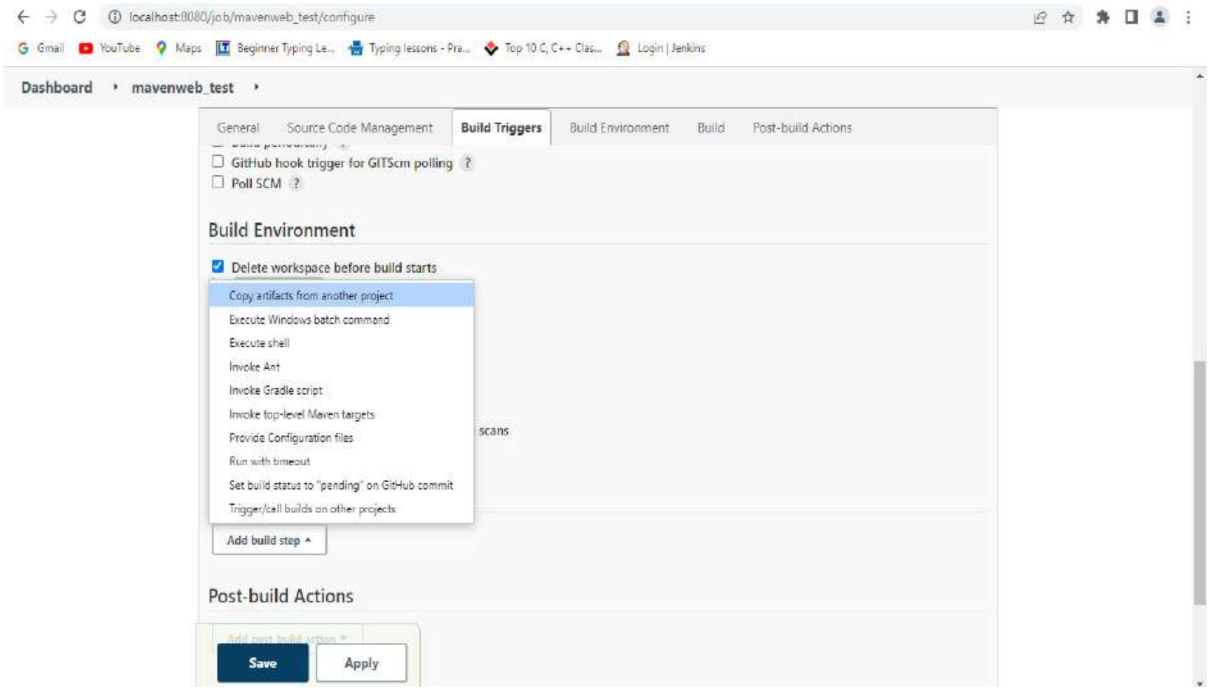
Maven project

Build a maven project. Jenkins takes advantage of your POM files and drastically reduces the configuration.

Pipeline

Orchestrates long-running activities that can span multiple build agents. Suitable for building pipelines (formerly known as workflows) and/or organizing complex activities that do not easily fit in free-style job type.

18. In description type eg. Test demo
19. In Build environment, check the box with name "Delete the workspace before build starts". This is to discard old builds
20. In Build select "**copy the artifacts from another project**" to forward the artifacts of the previous project to the current test project i.e. mavenweb_test

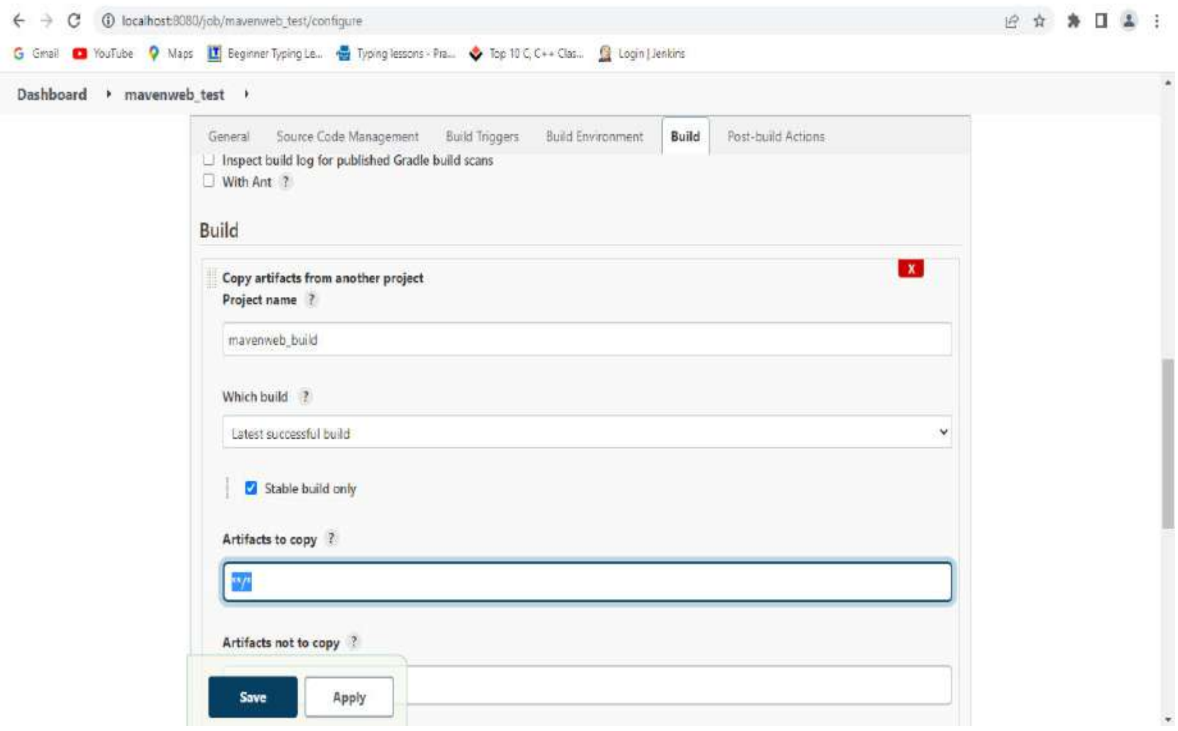


Build Environment

☑ Delete workspace before build starts

localhost:8080/job/mavenweb_test/configure
Dashboard
mavenweb_test
General
Source Code Management
Build Triggers
Build Environment
Build
Post-build Actions
GitHub hook trigger for GITScm polling
Poll SCM
Copy artifacts from another project
Execute Windows batch command
Execute shell
Invoke Ant
Invoke Gradle script
Invoke top-level Maven targets
Provide Configuration files
Run with timeout
Set build status to "pending" on GitHub commit
Trigger/call builds on other projects
scans
Add build step
Post-build Actions
Save
Apply

21. Give the name of the project from which we want to copy the artifacts (eg. Type mavenweb_build) and check the box ->stable build only->to copy all the artifacts type **/*


localhost:8080/job/mavenweb_test/configure

Dashboard › mavenweb_test ›

General | Source Code Management | Build Triggers | Build Environment | Build | Post-build Actions

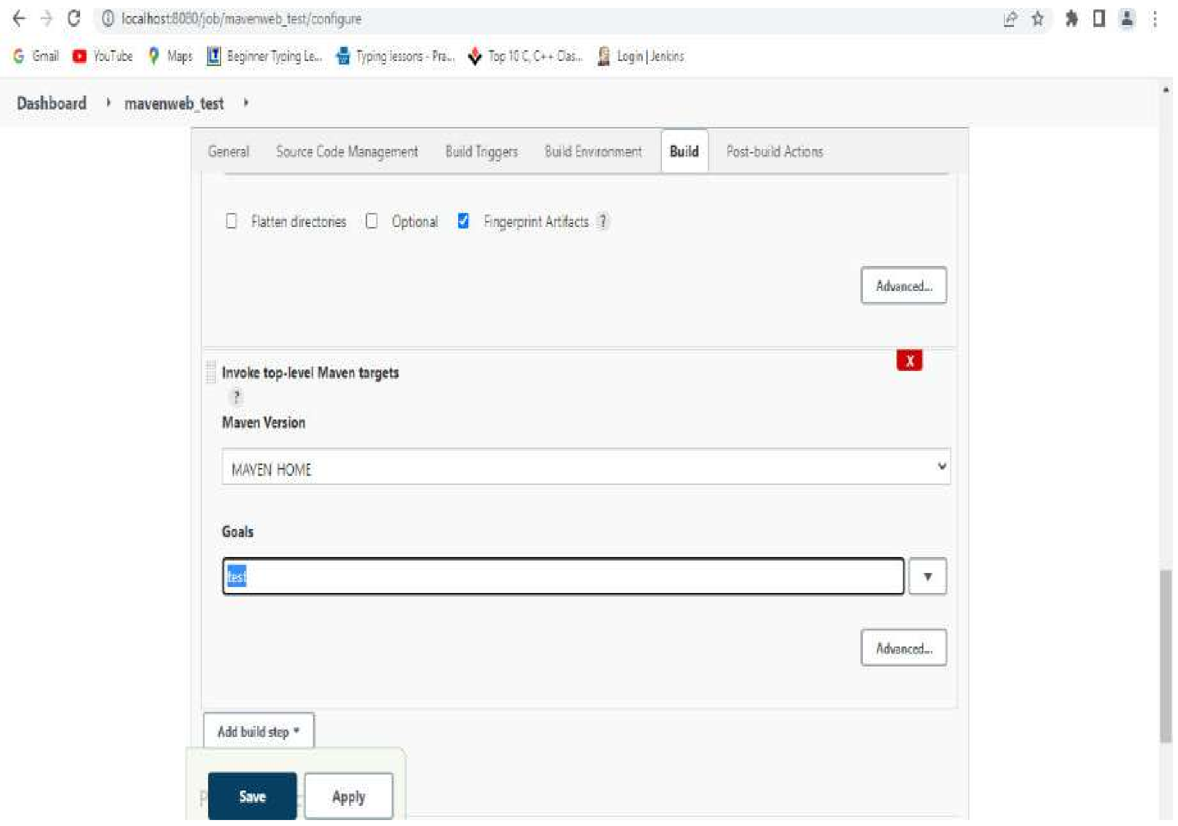
☐ Inspect build log for published Gradle build scans
☐ With Ant

Build

Copy artifacts from another project
Project name: mavenweb_build
Which build: Latest successful build
☑ Stable build only
Artifacts to copy: **/*
Artifacts not to copy:

Save | Apply

22. Now select Invoke top-level Maven targets in build
23. This time give the goal as test after selecting the Maven version


localhost:8080/job/mavenweb_test/configure

Dashboard › mavenweb_test ›

General | Source Code Management | Build Triggers | Build Environment | Build | Post-build Actions

☐ Flatten directories ☐ Optional ☑ Fingerprint Artifacts

Advanced...

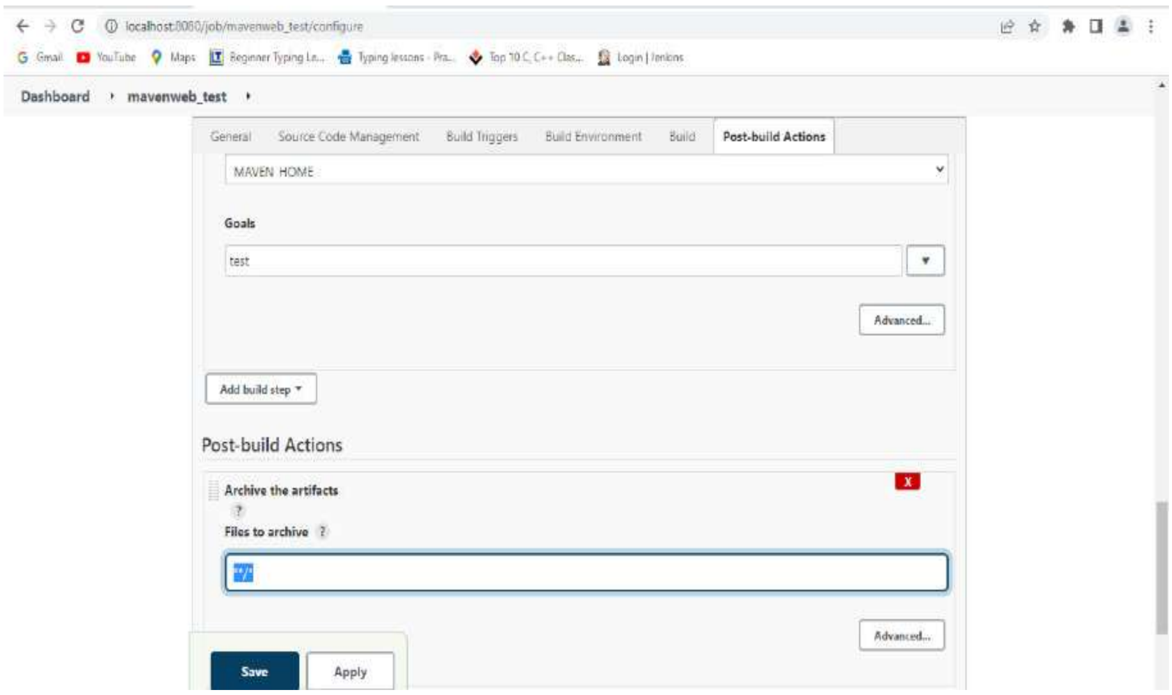
Invoke top-level Maven targets
Maven Version: MAVEN HOME
Goals: test

Advanced...

Add build step

Save | Apply

24. In post-build actions->select Archive the artifacts
25. To save all the artifacts->type **/* and Apply->Save



localhost:8080/job/mavenweb_test/configure

Dashboard › mavenweb_test ›

General | Source Code Management | Build Triggers | Build Environment | Build | Post-build Actions

MAVEN HOME

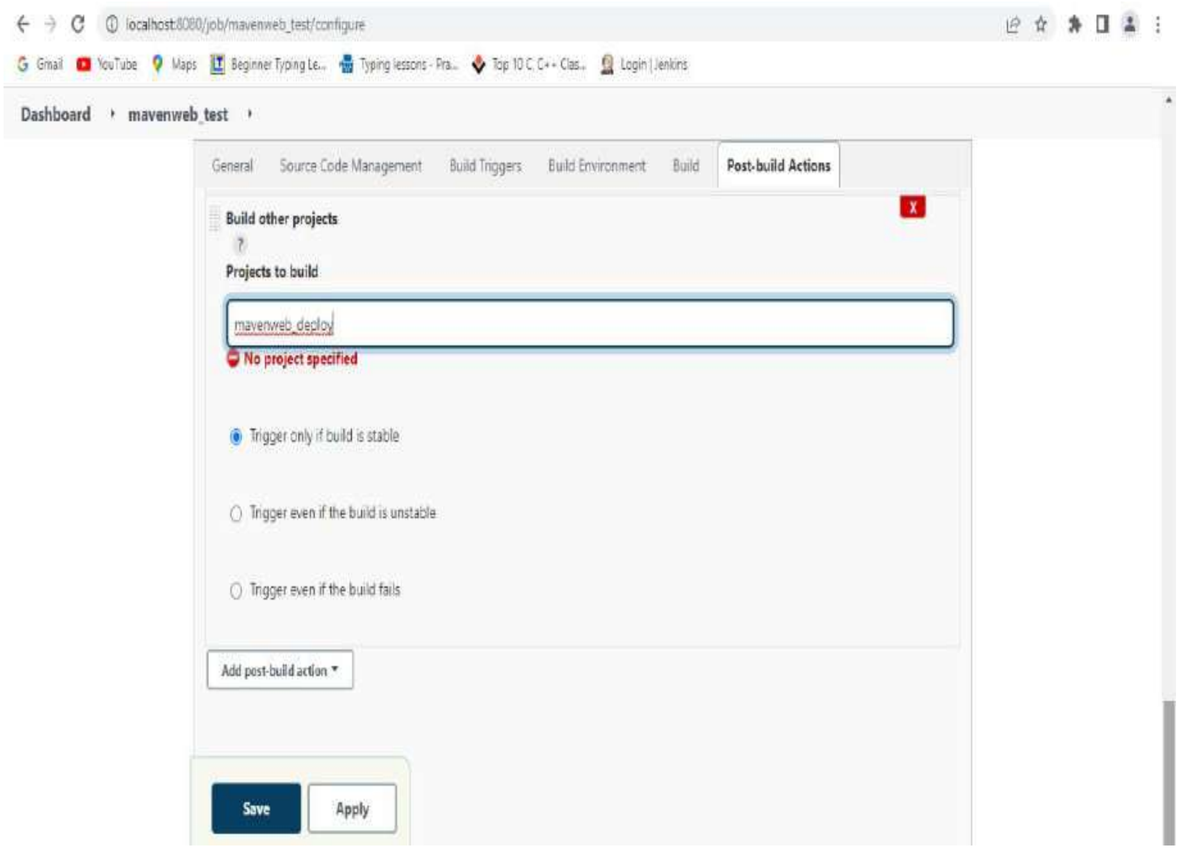
Goals: test

Post-build Actions

Archive the artifacts

Files to archive: **/*

26. Now we here select the build other projects, where we will create a deploy project which will be triggered by the test project, click Apply and Save



localhost:8080/job/mavenweb_test/configure

Dashboard › mavenweb_test ›

General | Source Code Management | Build Triggers | Build Environment | Build | Post-build Actions

Build other projects

Projects to build: mavenweb_deploy

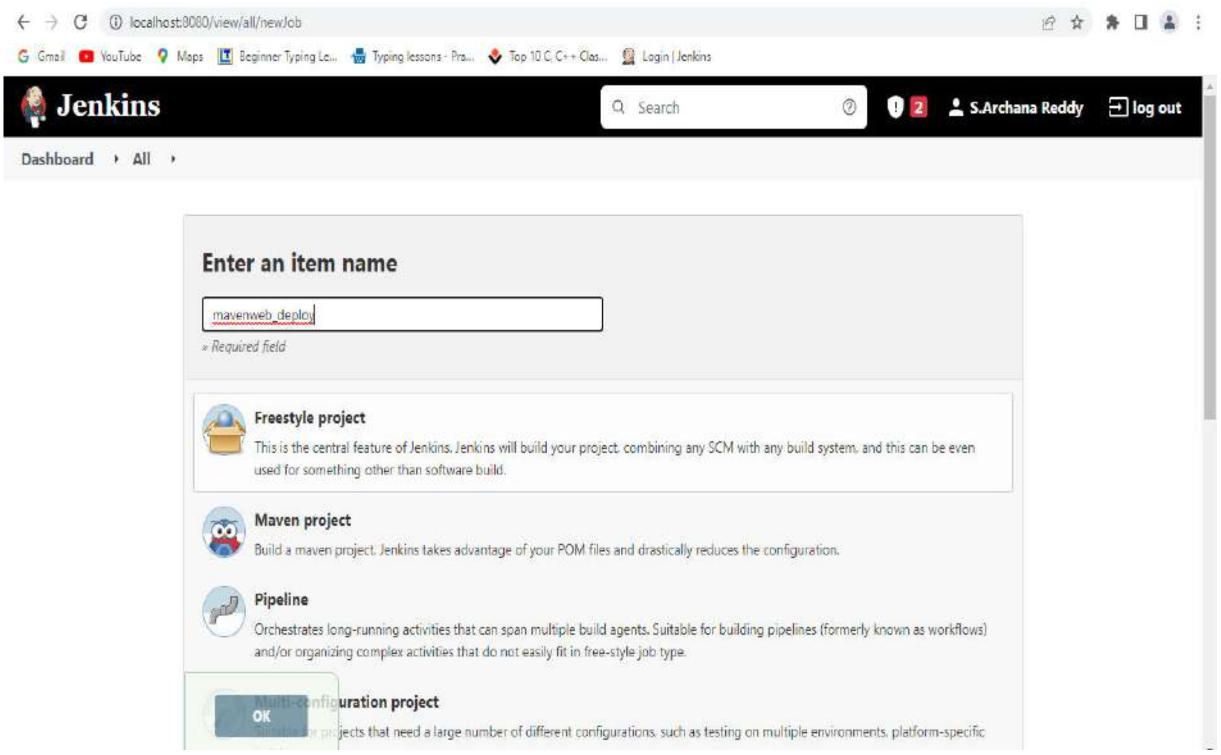
No project specified

● Trigger only if build is stable

○ Trigger even if the build is unstable

○ Trigger even if the build fails

27. Create a new freestyle project test as shown and click ok


localhost:8080/view/all/newJob

Jenkins

Dashboard › All ›

Enter an item name

mavenweb_deploy

» Required field

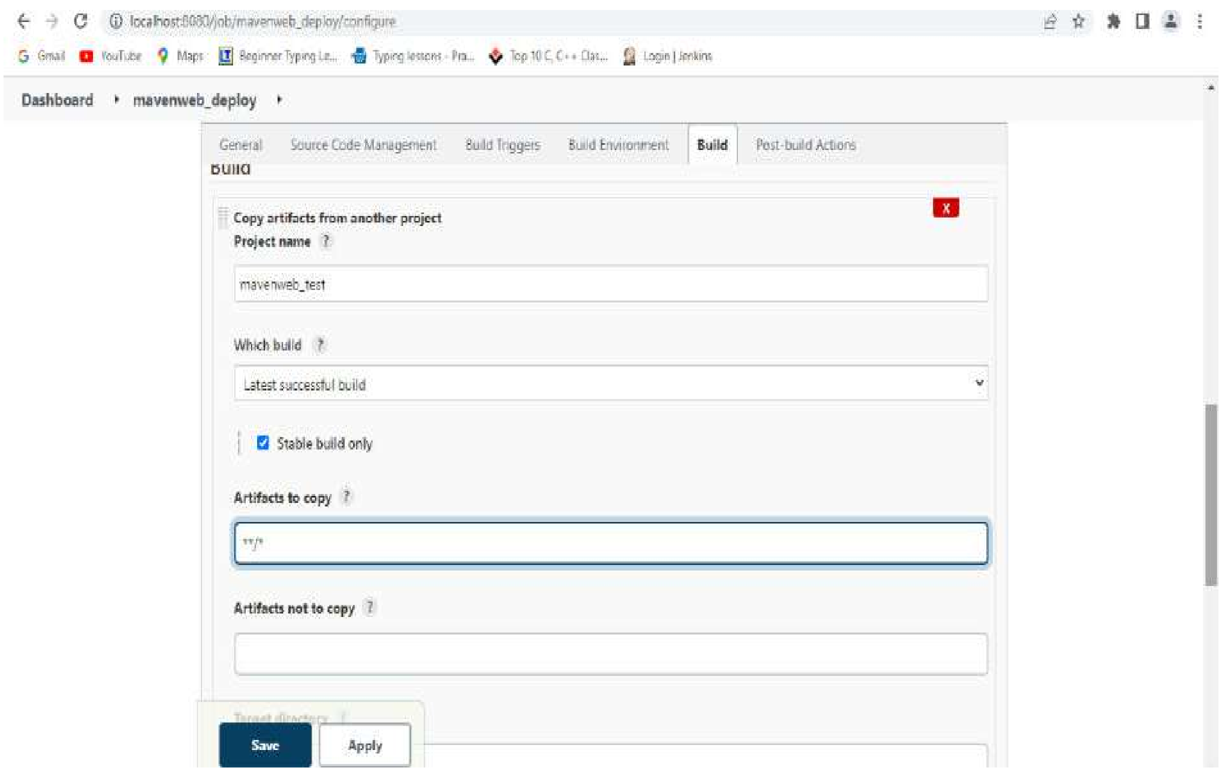
Freestyle project
This is the central feature of Jenkins. Jenkins will build your project, combining any SCM with any build system, and this can be even used for something other than software build.

Maven project
Build a maven project. Jenkins takes advantage of your POM files and drastically reduces the configuration.

Pipeline
Orchestrates long-running activities that can span multiple build agents. Suitable for building pipelines (formerly known as workflows) and/or organizing complex activities that do not easily fit in free-style job type.

Multi-configuration project
Suitable for projects that need a large number of different configurations, such as testing on multiple environments, platform-specific

28. Give the name of the project from which we want to copy the artifacts and check the box ->stable build only->to copy all the artifacts type **/*


localhost:8080/job/mavenweb_deploy/configure

Dashboard › mavenweb_deploy ›

General | Source Code Management | Build Triggers | Build Environment | Build | Post-build Actions

Build

Copy artifacts from another project

Project name: mavenweb_test

Which build: Latest successful build

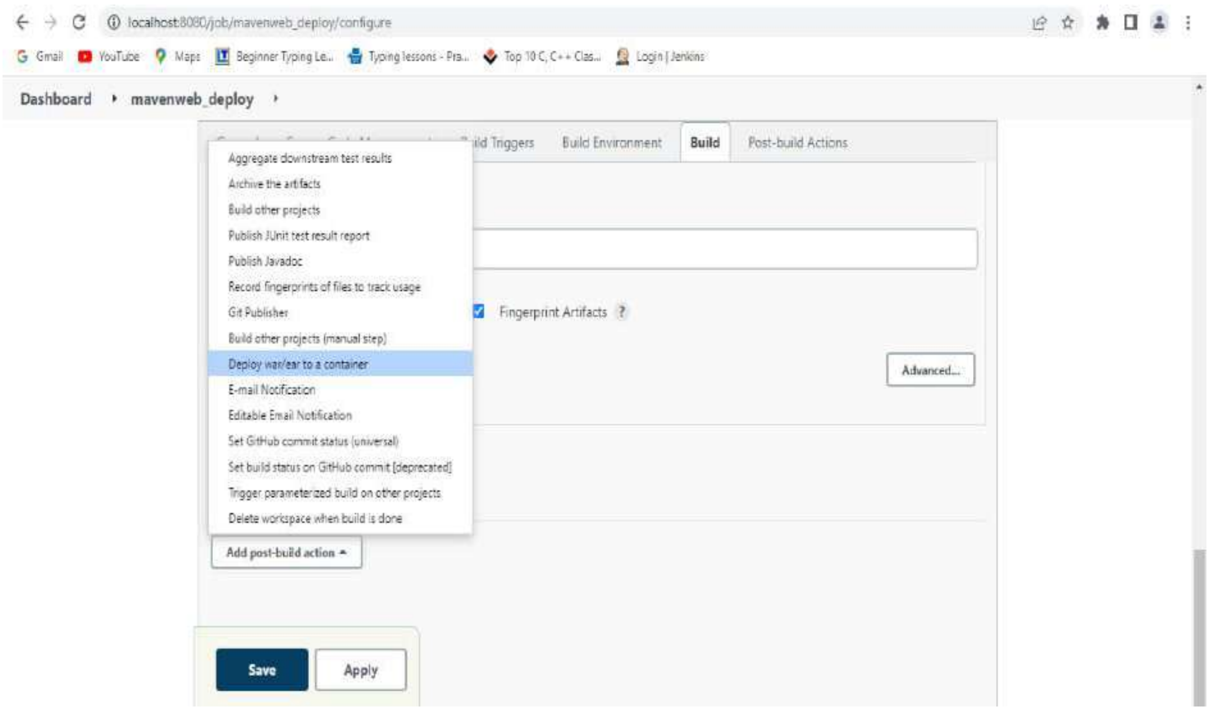
☑ Stable build only

Artifacts to copy: **/*

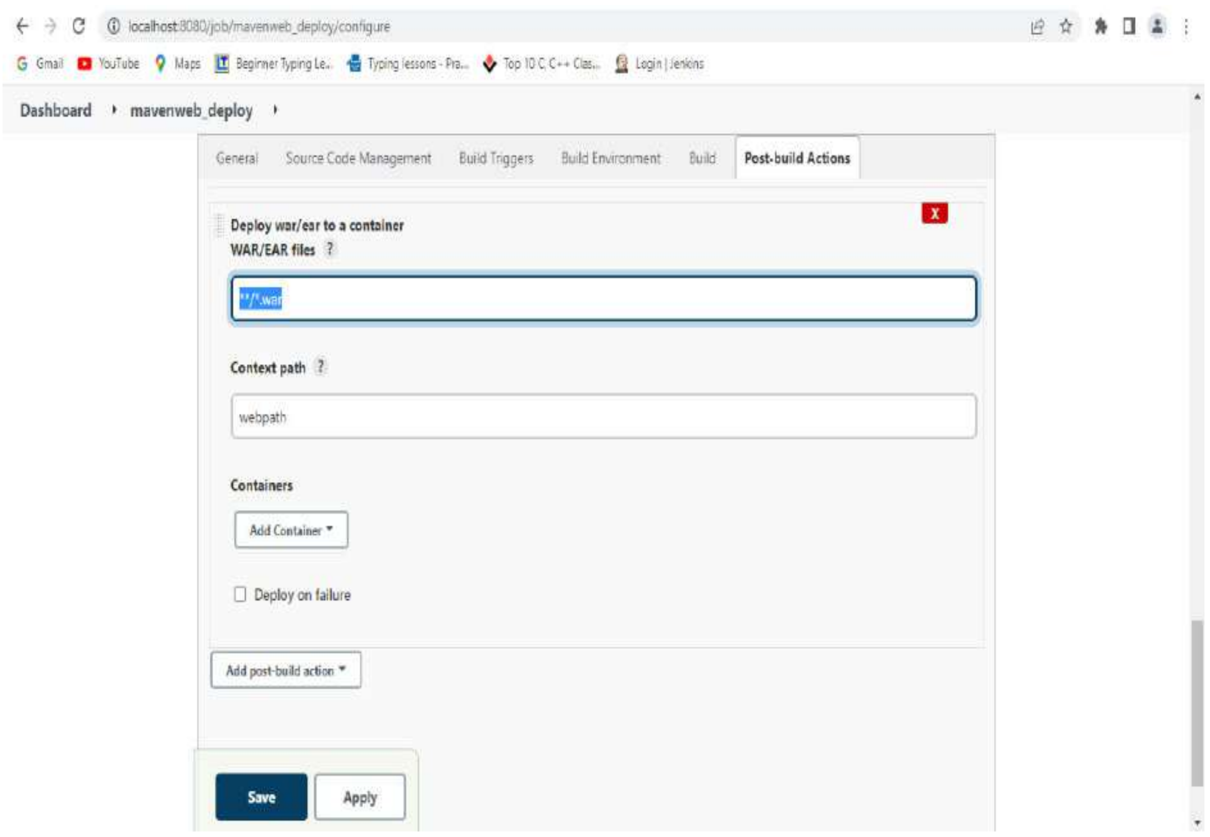
Artifacts not to copy:

29. Now here we go for Add post –build action where we select the Deploy war/ear to a container.

    This **plugin** takes a **war/ear** file and deploys that to a running remote application server at the end of a build.


localhost:8080/job/mavenweb_deploy/configure
Dashboard
mavenweb_deploy
Build Triggers
Build Environment
Build
Post-build Actions
Aggregate downstream test results
Archive the artifacts
Build other projects
Publish JUnit test result report
Publish Javadoc
Record fingerprints of files to track usage
Git Publisher
Build other projects (manual step)
Deploy war/ear to a container
E-mail Notification
Editable Email Notification
Set GitHub commit status (universal)
Set build status on GitHub commit [deprecated]
Trigger parameterized build on other projects
Delete workspace when build is done
Fingerprint Artifacts
Advanced...
Add post-build action
Save
Apply

30. Deploy war/ear to a container takes the artifacts as **/*.war.


localhost:8080/job/mavenweb_deploy/configure

Dashboard › mavenweb_deploy ›

General | Source Code Management | Build Triggers | Build Environment | Build | Post-build Actions

Deploy war/ear to a container

WAR/EAR files: **/*.war

Context path: webpath

Containers

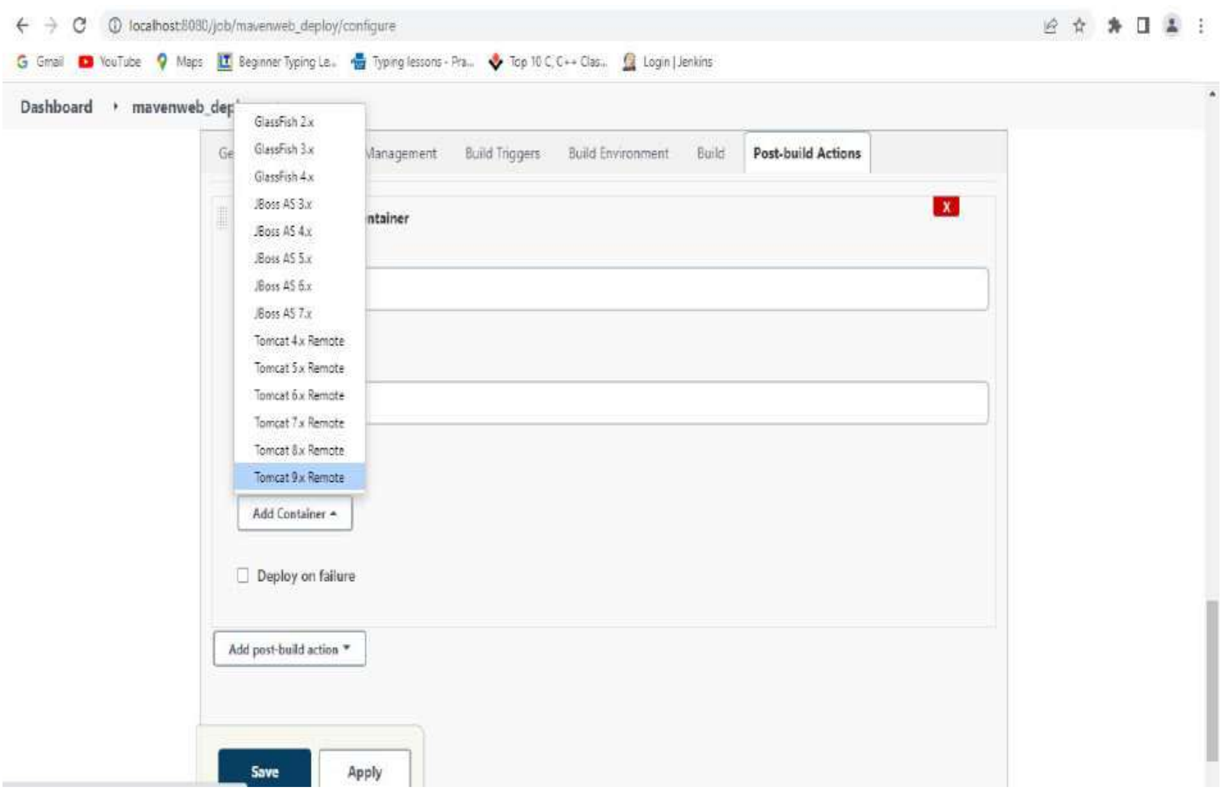
Add Container

☐ Deploy on failure

Add post-build action

Save Apply

31. Select the tomcat version.


localhost:8080/job/mavenweb_deploy/configure

GlassFish 2.x
GlassFish 3.x
GlassFish 4.x
JBoss AS 3.x
JBoss AS 4.x
JBoss AS 5.x
JBoss AS 6.x
JBoss AS 7.x
Tomcat 4.x Remote
Tomcat 5.x Remote
Tomcat 6.x Remote
Tomcat 7.x Remote
Tomcat 8.x Remote
Tomcat 9.x Remote

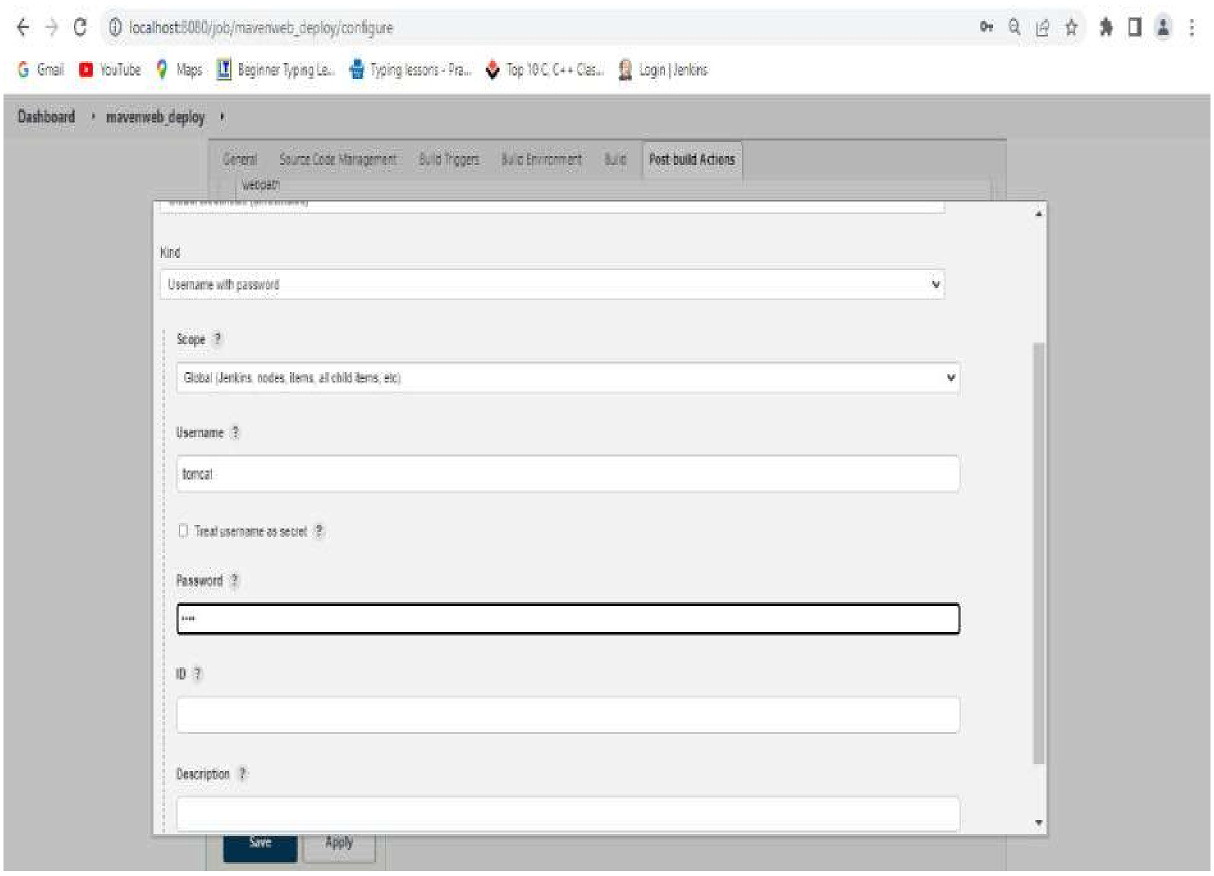
Add Container

☐ Deploy on failure

Add post-build action

Save Apply

32. Here we add the credentials of tomcat


localhost:8080/job/mavenweb_deploy/configure

Dashboard › mavenweb_deploy ›

Kind: Username with password

Scope: Global (Jenkins, nodes, items, all child items, etc)

Username: tomcat

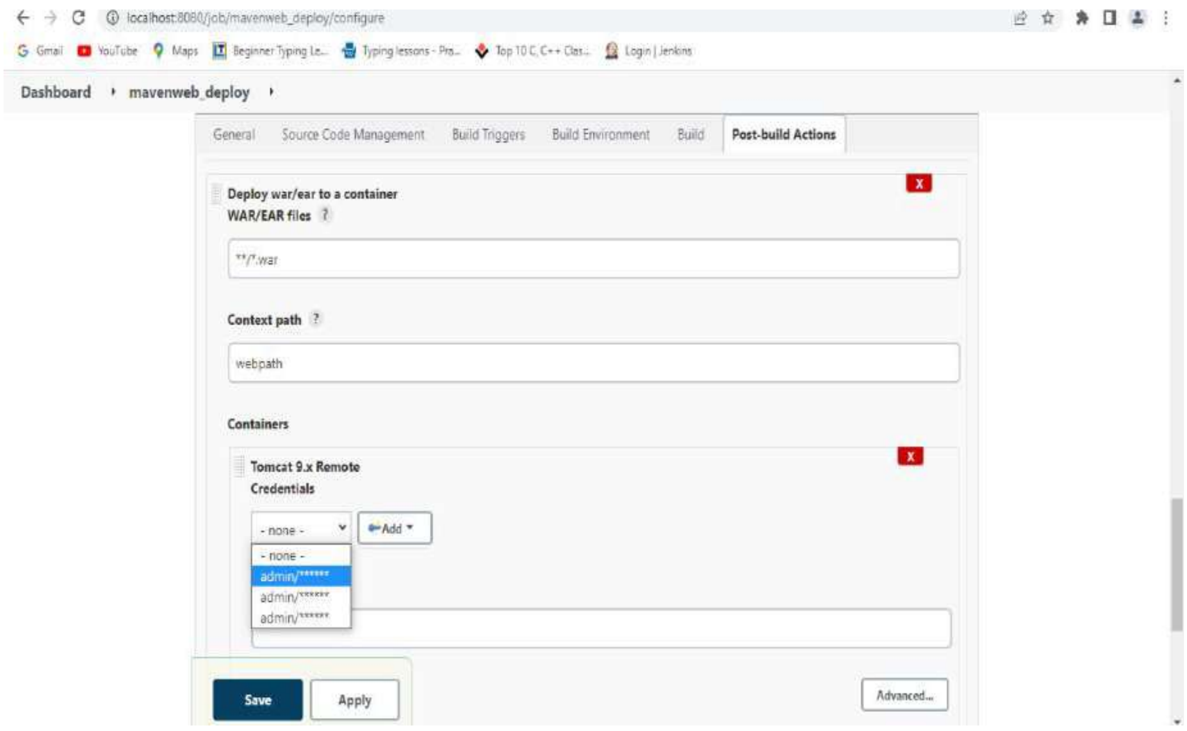
Treat username as secret

Password: ****

ID

Description

33. Here we added the credentials of tomcat and tomcat URL also


localhost:8080/job/mavenweb_deploy/configure

Dashboard › mavenweb_deploy ›

Deploy war/ear to a container

WAR/EAR files: **/*.war

Context path: webpath

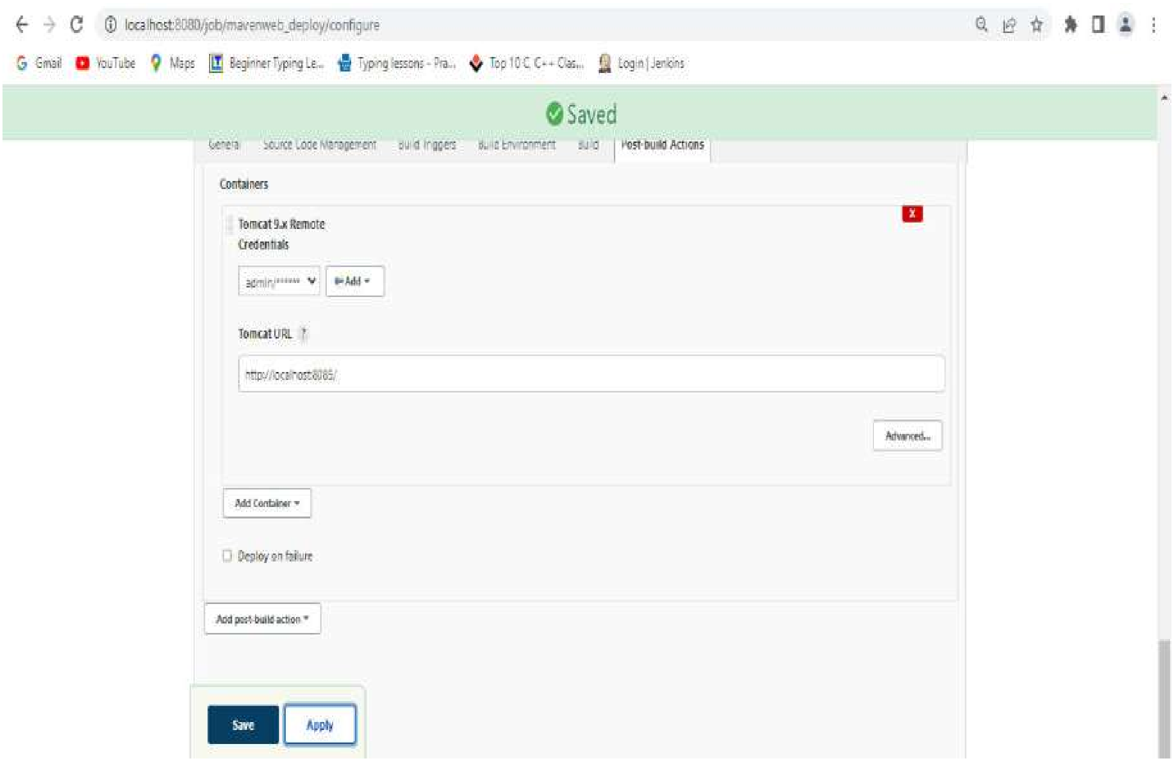
Containers

Tomcat 9.x Remote

Credentials: - none -

- none -
admin/******
admin/******
admin/******

34. Click on Apply and Save.



localhost:8080/job/mavenweb_deploy/configure

Saved

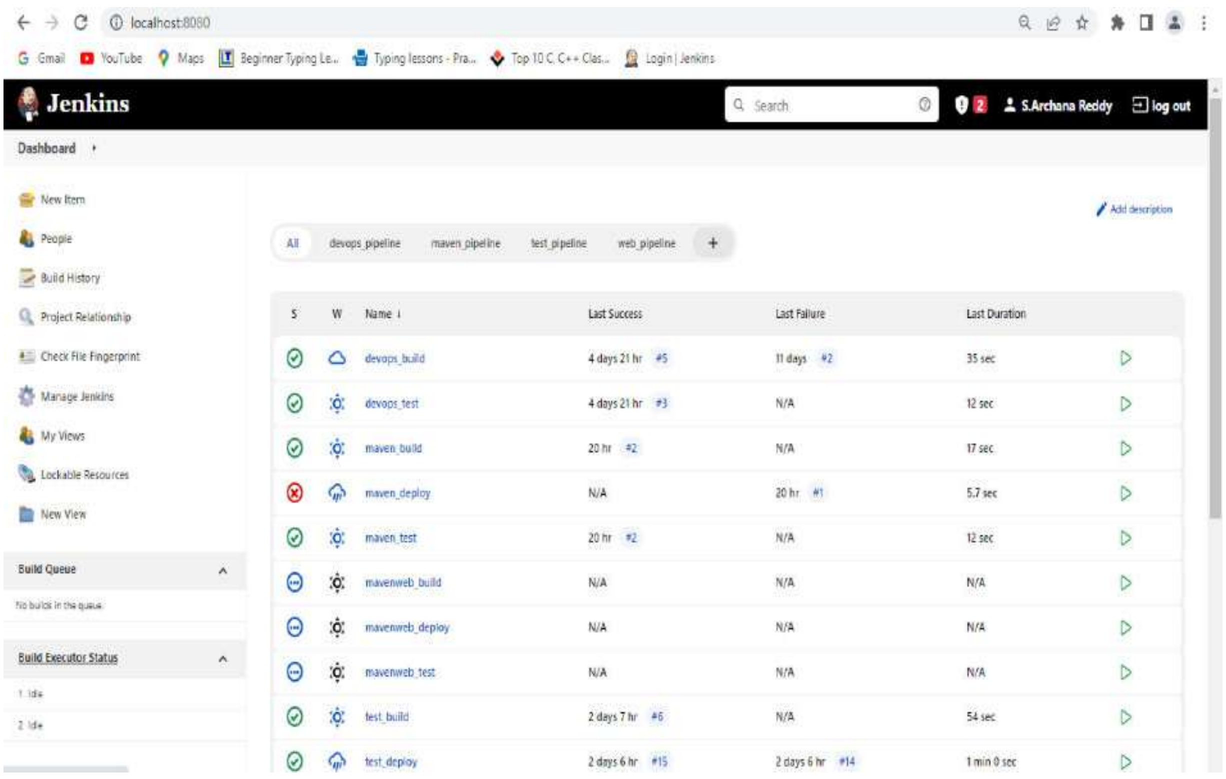
Containers

Tomcat 9.x Remote

Credentials: admin/******

Tomcat URL: http://localhost:8085/

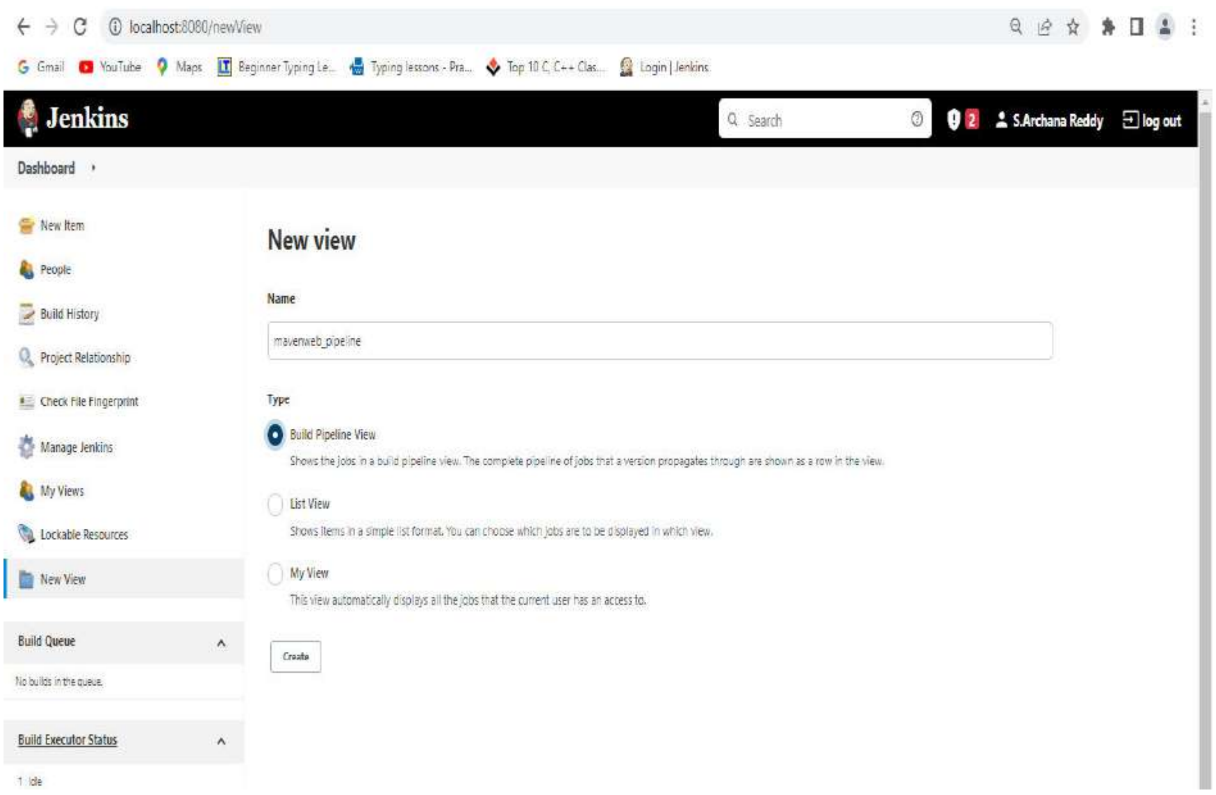
35. We Create a pipeline by clicking on + symbol in the dashboard ->a pipeline is a collection of events or jobs which are interlinked with one another in a sequence.



localhost:8080

S	W	Name ↓	Last Success	Last Failure	Last Duration
✓		devops_build	4 days 21 hr #5	11 days #2	35 sec
✓		devops_test	4 days 21 hr #3	N/A	12 sec
✓		maven_build	20 hr #2	N/A	17 sec
✗		maven_deploy	N/A	20 hr #1	5.7 sec
✓		maven_test	20 hr #2	N/A	12 sec
⊝		mavenweb_build	N/A	N/A	N/A
⊝		mavenweb_deploy	N/A	N/A	N/A
⊝		mavenweb_test	N/A	N/A	N/A
✓		test_build	2 days 7 hr #6	N/A	54 sec
✓		test_deploy	2 days 6 hr #15	2 days 6 hr #14	1 min 0 sec

36. Give a name to the pipeline->select Build Pipeline View->create


New view

Name: mavenweb_pipeline

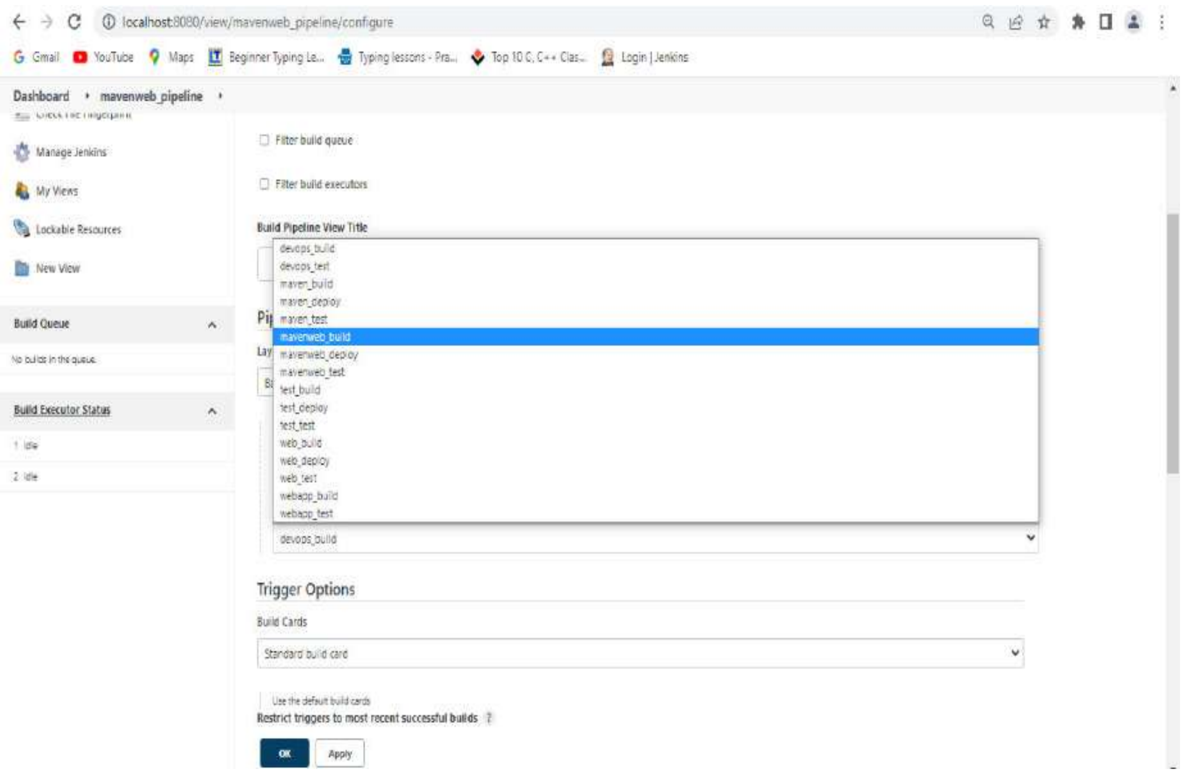
Type

[✓ Build Pipeline View] Shows the jobs in a build pipeline view. The complete pipeline of jobs that a version propagates through are shown as a row in the view.

List View — Shows items in a simple list format. You can choose which jobs are to be displayed in which view.

My View — This view automatically displays all the jobs that the current user has an access to.

37. Select the first project to trigger the execution->build session of your project


Filter build queue

Filter build executors

Build Pipeline View Title

devops_build
devops_test
maven_build
maven_deploy
maven_test
mavenweb_build
mavenweb_deploy
mavenweb_test
test_build
test_deploy
test_test
web_build
web_deploy
web_test
webapp_build
webapp_test

devops_build

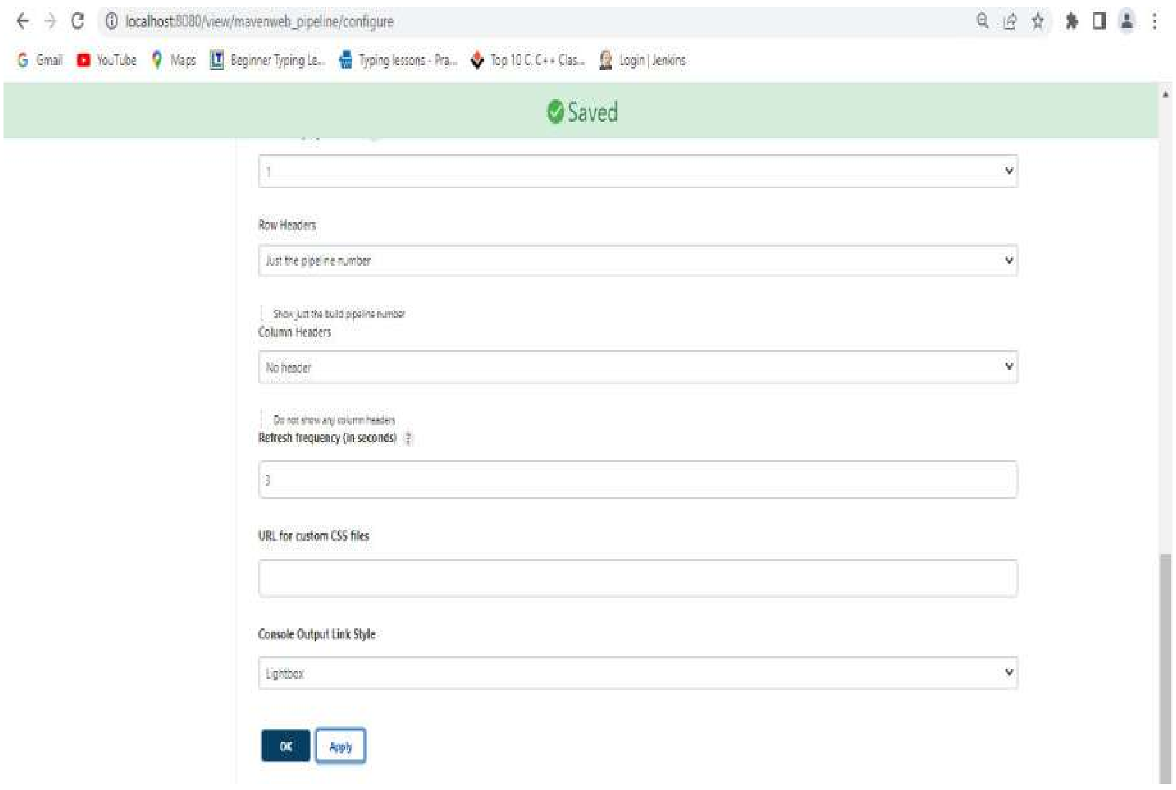
Trigger Options

Build Cards: Standard build card

Use the default build cards

Restrict triggers to most recent successful builds

38. Apply and Save



localhost:8080/view/mavenweb_pipeline/configure

Saved

1

Row Headers

Just the pipeline number

Show just the build pipeline number

Column Headers

No header

Do not show any column headers

Refresh frequency (in seconds)

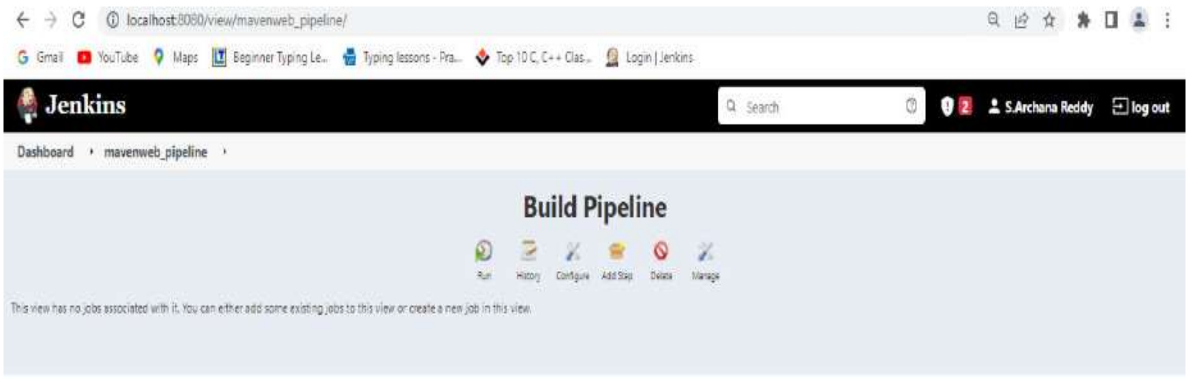
3

URL for custom CSS files

Console Output Link Style

Lightbox

39. This is the console after save. Now click on run.

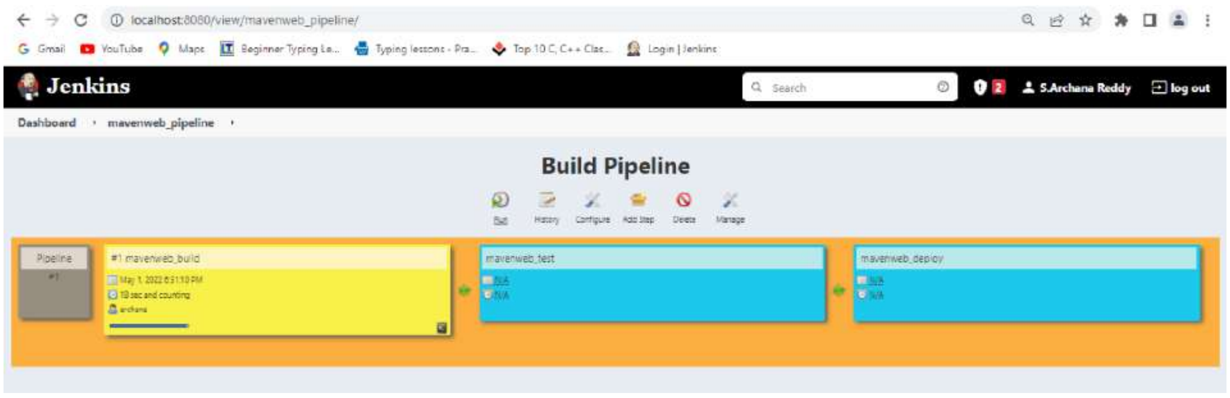


localhost:8080/view/mavenweb_pipeline/

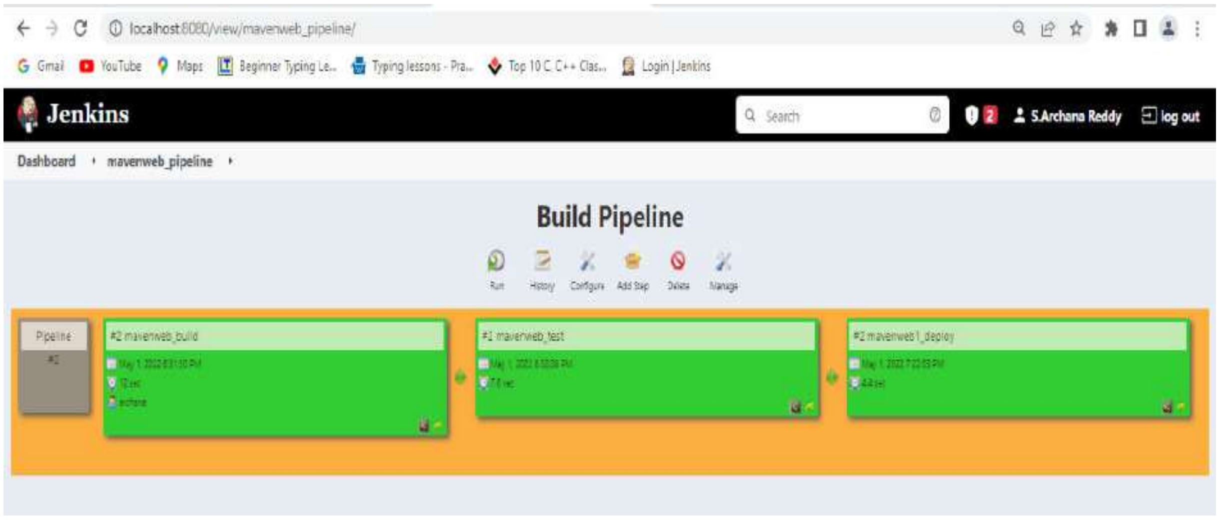
Build Pipeline

This view has no jobs associated with it. You can either add some existing jobs to this view or create a new job in this view.

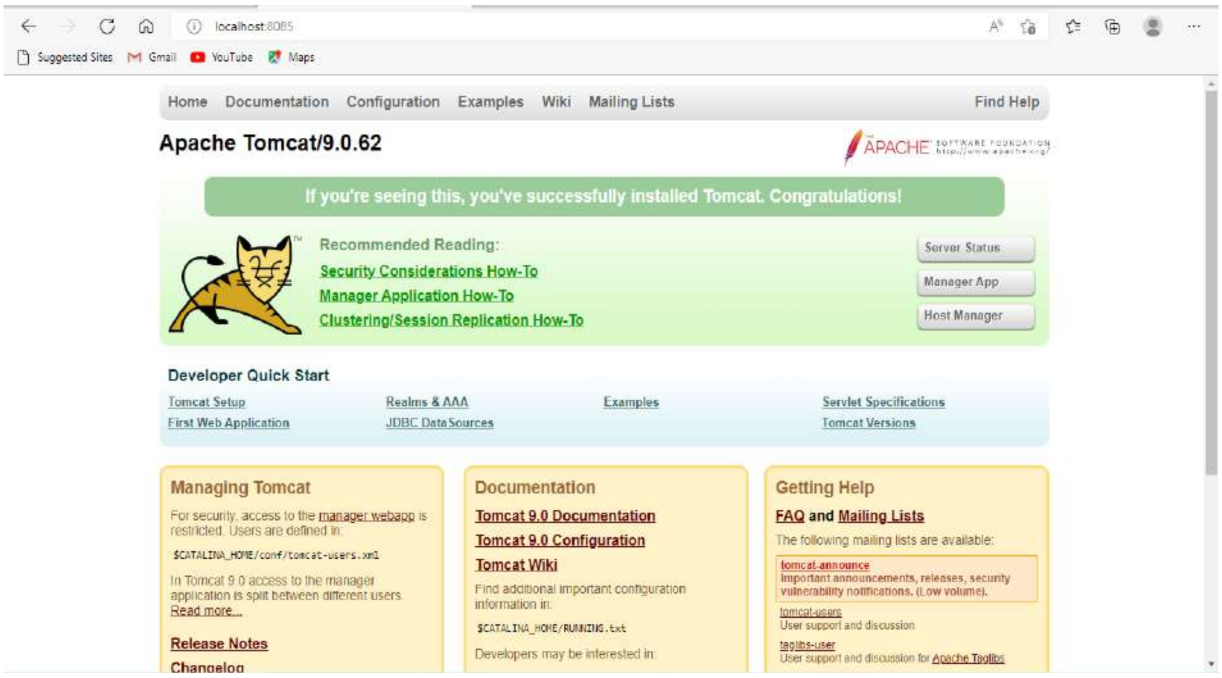
40. After Click on Run -> click on the small black box to open the console to check if the build is success



41. Now we see all the build has success.



42. Now we can run the local host of tomcat, click -> manager App



Apache Tomcat/9.0.62

If you're seeing this, you've successfully installed Tomcat. Congratulations!

Recommended Reading:
Security Considerations How-To
Manager Application How-To
Clustering/Session Replication How-To

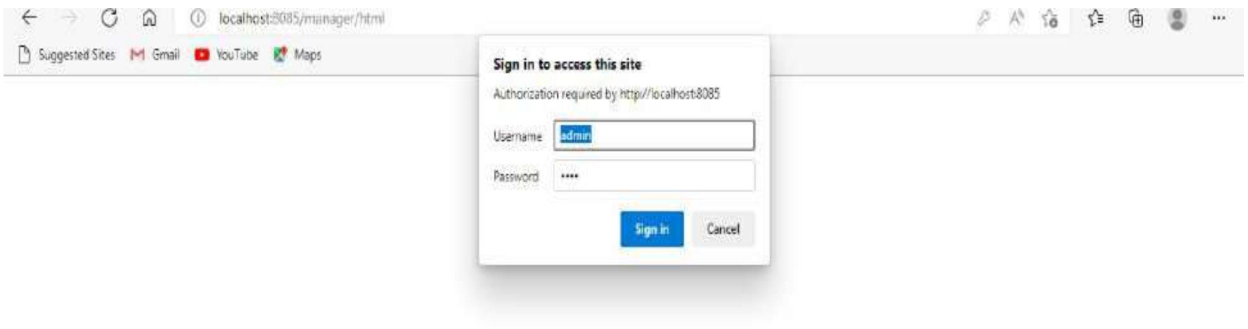
Developer Quick Start
Tomcat Setup
First Web Application
Realms & AAA
JDBC DataSources
Examples
Servlet Specifications
Tomcat Versions

Managing Tomcat
For security, access to the manager webapp is restricted. Users are defined in:
$CATALINA_HOME/conf/tomcat-users.xml
In Tomcat 9.0 access to the manager application is split between different users.
Read more...
Release Notes
Changelog

Documentation
Tomcat 9.0 Documentation
Tomcat 9.0 Configuration
Tomcat Wiki
Find additional important configuration information in:
$CATALINA_HOME/RUNNING.txt
Developers may be interested in:

Getting Help
FAQ and Mailing Lists
The following mailing lists are available:
tomcat-announce
Important announcements, releases, security vulnerability notifications. (Low volume).
tomcat-users
User support and discussion
taglibs-user
User support and discussion for Apache Taglibs

43. It ask for user credentials for login ,provide the credentials of tomcat.



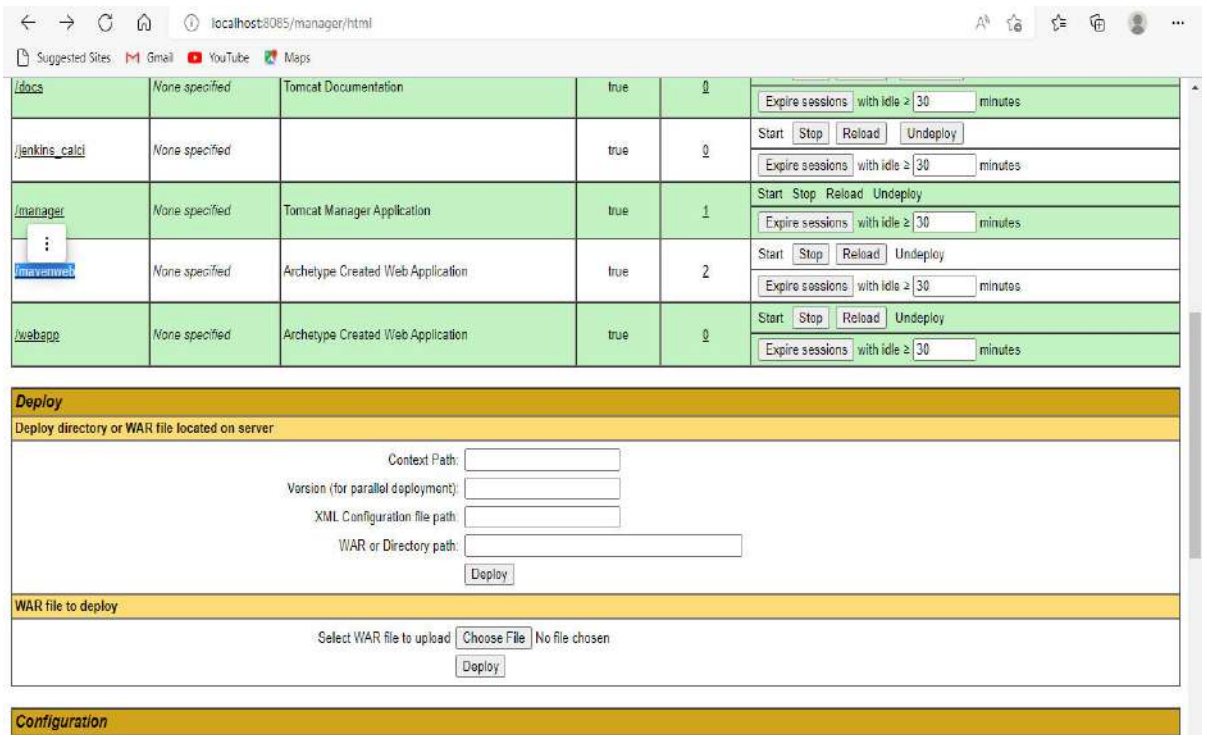
Sign in to access this site

Authorization required by http://localhost:8085

Username: admin

Password: ••••

44. It provide the page without project name which is highlighted.



/docs	None specified	Tomcat Documentation	true	0	Expire sessions with idle ≥ 30 minutes
/jenkins_calci	None specified		true	0	Start Stop Reload Undeploy Expire sessions with idle ≥ 30 minutes
/manager	None specified	Tomcat Manager Application	true	1	Start Stop Reload Undeploy Expire sessions with idle ≥ 30 minutes
/mavenweb	None specified	Archetype Created Web Application	true	2	Start Stop Reload Undeploy Expire sessions with idle ≥ 30 minutes
/webapp	None specified	Archetype Created Web Application	true	0	Start Stop Reload Undeploy Expire sessions with idle ≥ 30 minutes

Deploy

Deploy directory or WAR file located on server

Context Path:

Version (for parallel deployment):

XML Configuration file path:

WAR or Directory path:

Deploy

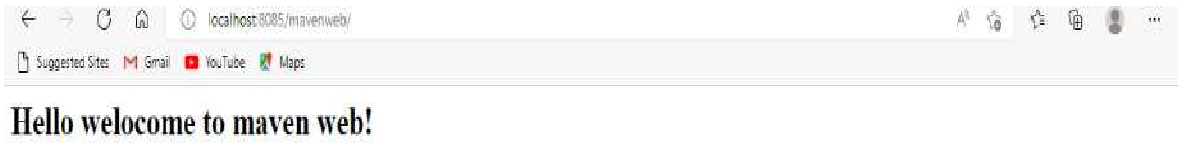
WAR file to deploy

Select WAR file to upload Choose File No file chosen

Deploy

Configuration

45. After clicking on our project, we can see our output here.



Hello welocome to maven web!