Introduction to Maven

Maven is a build automation tool primarily used for Java projects. It simplifies the build process, dependency management, and project management. Maven uses a project object model (POM) to manage project configurations, dependencies, and plugins.

Key Features of Maven:

- **Dependency Management**: Automatically downloads and manages project dependencies.
- **Build Automation**: Standardizes the build process across different environments.
- **Project Management**: Provides a clear structure for project organization.
- **Extensibility**: Supports plugins to extend its capabilities.

Folder Structure

A typical Maven project follows a standard directory structure:

““

```
my-app
|-- pom.xml
`-- src
    |-- main
    |   |-- java
    |   |   `-- com
    |   |       `-- example
    |   |           `-- app
    |   |               `-- App.java
    |   `-- resources
    |       `-- application.properties
    `-- test
        |-- java
        |   `-- com
        |       `-- example
        |           `-- app
        |               `-- AppTest.java
        `-- resources
```

Explanation:

- **pom.xml**: The Project Object Model file that contains configuration details.

- **src/main/java**: Contains the main application source code.
- **src/main/resources**: Contains configuration files and other resources.
- **src/test/java**: Contains test source code.
- **src/test/resources**: Contains resources for testing.

The pom.xml

The **pom.xml** file is the core of a Maven project. It defines the project structure, dependencies, plugins, and other configurations.

Basic Structure of pom.xml:

xml

““

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <groupId>com.example</groupId>
  <artifactId>my-app</artifactId>
  <version>1.0-SNAPSHOT</version>

    <dependencies>
        <!-- Dependencies go here -->
    </dependencies>

    <build>
        <plugins>
            <!-- Plugins go here -->
        </plugins>
    </build>
</project>
```

Key Elements:

- **groupId**: Unique identifier for the group or organization.

- **artifactId**: Unique identifier for the project.
- **version**: Version of the project.
- **dependencies**: List of project dependencies.
- **build**: Configuration for build plugins.

Dependencies

Dependencies are external libraries that your project needs to compile and run. You can specify dependencies in the **pom.xml** file.

Example of Adding Dependencies:

xml

““

```
<dependencies>
  <dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.12</version>
    <scope>test</scope>
  </dependency>
  <dependency>
    <groupId>org.apache.commons</groupId>
    <artifactId>commons-lang3</artifactId>
    <version>3.12.0</version>
  </dependency>
</dependencies>
```

Dependency Scope:

- **compile**: Default scope, available in all classpaths.
- **provided**: Available at compile time but not at runtime (e.g., servlet API).
- **runtime**: Not needed for compilation but required for execution.
- **test**: Only available for testing.
- **system**: Similar to provided but requires an explicit path.

Goals

Maven goals are specific tasks that can be executed. Each goal is associated with a plugin and can be executed from the command line.

Common Goals:

- **clean**: Removes the **target** directory.
- **compile**: Compiles the source code.
- **test**: Runs the tests.
- **package**: Packages the compiled code into a JAR/WAR file.
- **install**: Installs the package into the local repository.

Example Command:

bash

““

```
mvn clean install
```

Scopes

Scopes define the visibility and availability of dependencies in different phases of the build lifecycle. The most common scopes are:

- **compile**: Default scope, available in all phases.
- **provided**: Available during compilation but not packaged.
- **runtime**: Required for execution but not for compilation.
- **test**: Only available during the test phase.
- **system**: Similar to provided but requires an explicit path.

The Compiler Plugin

You

““

““

The Compiler Plugin

The Compiler Plugin is responsible for compiling the source code of your project. By default, it compiles Java source files located in the **src/main/java** directory.

Configuration Example:

You can configure the Compiler Plugin in your **pom.xml** to specify the Java version and other options.

xml

““

```xml
<build>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-compiler-plugin</artifactId>
      <version>3.8.1</version>
      <configuration>
        <source>1.8</source>
        <target>1.8</target>
      </configuration>
    </plugin>
  </plugins>
</build>
```

Key Configuration Options:

- **source**: Specifies the version of the source code.
- **target**: Specifies the version of the bytecode to be generated.

The Source Plugin

The Source Plugin is used to generate source JARs from your project. This is useful for distributing your source code along with your binaries.

Configuration Example:

xml

““

```xml
<build>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-source-plugin</artifactId>
      <version>3.2.1</version>
      <executions>
        <execution>
          <id>attach-sources</id>
          <goals>
            <goal>jar</goal>
          </goals>
        </execution>
      </executions>
    </plugin>
  </plugins>
</build>
```

Key Features:

- Generates a JAR file containing the source code.
- Can be configured to run during specific phases of the build lifecycle.

The Jar Plugin

The Jar Plugin is responsible for packaging your compiled code into a JAR file. This is typically done during the **package** phase of the Maven lifecycle.

Configuration Example:

xml

““

```
<build>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-jar-plugin</artifactId>
      <version>3.2.0</version>
      <configuration>
        <archive>
          <manifest>
            <addClasspath>true</addClasspath>
            <mainClass>com.example.app.App</mainClass>
          </manifest>
        </archive>
      </configuration>
```

```
        </plugin>
    </plugins>
</build>
```

Key Configuration Options:

- **addClasspath**: Adds the classpath to the manifest file.
- **mainClass**: Specifies the main class to be executed when the JAR is run.

Maven Life Cycle

Maven has a defined lifecycle that consists of a series of phases. Each phase represents a stage in the build process. The three main lifecycles are:

1. **Default Lifecycle**: Handles the project deployment.
2. **Clean Lifecycle**: Handles project cleaning.
3. **Site Lifecycle**: Handles the creation of project documentation.

Default Lifecycle Phases:

- **validate**: Validates the project structure and configuration.
- **compile**: Compiles the source code.
- **test**: Runs tests using a suitable testing framework.
- **package**: Packages the compiled code into a distributable format (e.g., JAR).
- **install**: Installs the package into the local repository.
- **deploy**: Copies the final package to a remote repository for sharing.

Clean Lifecycle Phases:

- **pre-clean**: Executes actions before cleaning.
- **clean**: Removes the **target** directory.
- **post-clean**: Executes actions after cleaning.

Site Lifecycle Phases:

- **pre-site**: Executes actions before generating the site.
- **site**: Generates the project's site documentation.
- **post-site**: Executes actions after generating the site.
- **site-deploy**: Deploys the generated site to a web server.

Example Command to Execute a Lifecycle Phase:

To execute the default lifecycle up to the **install** phase, you can run:

bash

““

```
mvn install
```

This command will execute all phases from **validate** to **install**.

Conclusion

Maven is a powerful tool for managing Java projects, providing a standardized way to handle dependencies, build processes, and project structures. By understanding the key components such as the **pom.xml**, folder structure, plugins, and lifecycle phases, you can effectively manage your projects and streamline your development workflow.

Validations in spring boot

n Spring Boot, you can implement validation using Java Bean Validation (JSR 380) annotations. This allows you to enforce constraints on your model classes and automatically validate input data in your controllers. Below is a step-by-step tutorial on how to implement validation using validation annotations in a Spring Boot application.

Step 1: Set Up Your Spring Boot Project

You can create a new Spring Boot project using Spring Initializr (https://start.spring.io/) or your favorite IDE. Make sure to include the following dependencies:

- **Spring Web**: For building web applications.
- **Spring Boot DevTools**: For easier development.
- **Spring Validation**: For validation support (this is included with Spring Web).

Step 2: Create a Model Class

Create a model class that represents the data you want to validate. Use validation annotations from the **javax.validation.constraints** package.

java

““

```
package com.example.demo.model;

import javax.validation.constraints.Email;
import javax.validation.constraints.NotBlank;
import javax.validation.constraints.Size;

public class User {

```

```java
@NotBlank(message = "Name is mandatory")
private String name;

@Email(message = "Email should be valid")
private String email;

@Size(min = 6, message = "Password must be at least 6 characters long")
private String password;

// Getters and Setters
public String getName() {
   return name;
}

public void setName(String name) {
   this.name = name;
}

public String getEmail() {
   return email;
}

public void setEmail(String email) {
   this.email = email;
}

public String getPassword() {
   return password;
}

public void setPassword(String password) {
        this.password = password;
    }
}
```

Step 3: Create a Controller

Create a controller that handles HTTP requests and uses the **@Valid** annotation to trigger validation.

java

““

```
package com.example.demo.controller;

import com.example.demo.model.User;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.validation.BindingResult;
import org.springframework.web.bind.annotation.*;

import javax.validation.Valid;

@RestController
@RequestMapping("/api/users")
public class UserController {

    @PostMapping
    public ResponseEntity<String> createUser (@Valid @RequestBody User user, BindingResult bindingResult) {
        if (bindingResult.hasErrors()) {
            return ResponseEntity.badRequest().body(bindingResult.getAllErrors().toString());
        }
        // Here you would typically save the user to the database
        return ResponseEntity.status(HttpStatus.CREATED).body("User  created successfully");
    }
}
```

Step 4: Handle Validation Errors

In the **createUser** method, we check for validation errors using the **BindingResult** parameter. If there are errors, we return a bad request response with the error messages.

Step 5: Run Your Application

Make sure your application is set up correctly and run it. You can use tools like Postman or cURL to test your API.

Example Request

You can test the validation by sending a POST request to **http://localhost:8080/api/users** with the following JSON body:

Valid Request

json

““

```json
{
  "name": "John Doe",
  "email": "john.doe@example.com",
  "password": "password123"
}
```

Invalid Request

json

““

```json
{
  "name": "",
  "email": "invalid-email",
  "password": "123"
}
```

Expected Responses

- For the valid request, you should receive a **201 Created** response with the message "User created successfully".
- For the invalid request, you should receive a **400 Bad Request** response with a list of validation error messages, such as:

““

1[Field error in object 'user' on field 'name': rejected value ''; codes [NotBlank.user.name,NotBlank.name,NotBlank.java.lang.String,NotBlank]; arguments []; default message [Name is mandatory],

2Field error in object 'user' on field 'email': rejected value 'invalid-email'; codes [Email.user.email,Email.email,Email.java.lang.String,Email]; arguments []; default message [Email should be valid],

3Field error in object 'user' on field 'password': rejected value '123'; codes [Size.user.password,Size.password,Size.java.lang.String,Size]; arguments [6]; default message [Password must be at least 6 characters long]]

Step 6: Customize Error Messages (Optional)

You can customize error messages by creating a **messages.properties** file in the **src/main/resources** directory. For example:

properties

1 NotBlank.user.name=Please provide your name

2 Email.user.email=Please provide a valid email address

3 Size.user.password=Password must be at least {min} characters long

---

Customer Validator in angular

Create a Custom Validator

You can create a custom validator as a function that returns either **null** (if the validation passes) or an object (if the validation fails).

Example: Custom Validator for Password Match

Let's create a custom validator that checks if two password fields match.

1. Create a new file for the validator, e.g., **password-match.validator.ts** in the **src/app/validators** directory.

typescript

““

1// src/app/validators/password-match.validator.ts

2import { AbstractControl, ValidationErrors, ValidatorFn } from '@angular/forms';

3

```
export function passwordMatchValidator(controlName: string, matchingControlName: string): ValidatorFn {
  return (formGroup: AbstractControl): ValidationErrors | null => {
    const control = formGroup.get(controlName);
    const matchingControl = formGroup.get(matchingControlName);

    if (matchingControl?.errors && !matchingControl.errors['passwordMismatch']) {
      return null; // if another validator has already found an error on the matchingControl
    }

    if (control?.value !== matchingControl?.value) {
      matchingControl?.setErrors({ passwordMismatch: true });
      return { passwordMismatch: true };
    } else {
      matchingControl?.setErrors(null);
      return null;
    }
  };
}
```

Step 3: Create a Reactive Form

Now, let's create a reactive form that uses this custom validator.

1. Open the **app.module.ts** file and import **ReactiveFormsModule**.

typescript

““

```
// src/app/app.module.ts
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { ReactiveFormsModule } from '@angular/forms'; // Import ReactiveFormsModule
import { AppComponent } from './app.component';

@NgModule({
  declarations: [AppComponent],
  imports: [BrowserModule, ReactiveFormsModule], // Add ReactiveFormsModule here
  providers: [],
  bootstrap: [AppComponent],
})
export class AppModule {}
```

2. Open the **app.component.ts** file and create a form with password fields.

typescript

“‘

```
// src/app/app.component.ts
import { Component, OnInit } from '@angular/core';
import { FormBuilder, FormGroup, Validators } from '@angular/forms';
import { passwordMatchValidator } from './validators/password-match.validator';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css'],
})
export class AppComponent implements OnInit {
  form: FormGroup;

  constructor(private fb: FormBuilder) {
    this.form = this.fb.group(
      {
        password: ['', [Validators.required, Validators.minLength(6)]],
        confirmPassword: ['', Validators.required],
      },
      { validators: passwordMatchValidator('password', 'confirmPassword') } // Apply custom validator
    );
  }

  ngOnInit(): void {}

  onSubmit(): void {
    if (this.form.valid) {
      console.log('Form Submitted!', this.form.value);
    } else {
      console.log('Form is invalid');
    }
  }
}
```

Step 4: Create the Template

Now, create the template for the form in **app.component.html**.

html

““

```
<!-- src/app/app.component.html -->
<div style="text-align:center">
  <h1>Custom Validation Example</h1>
  <form [formGroup]="form" (ngSubmit)="onSubmit()">
    <div>
      <label for="password">Password:</label>
      <input id="password" formControlName="password" type="password" />
      <div *ngIf="form.get('password')?.invalid && form.get('password')?.touched">
        <small *ngIf="form.get('password')?.errors?.required">Password is required.</small>
        <small *ngIf="form.get('password')?.errors?.minlength">Password must be at least 6 characters long.</small>
      </div>
    </div>

    <div>
```

```
<label for="confirmPassword">Confirm Password:</label>
<input id="confirmPassword" formControlName="confirmPassword" type="password" />
<div *ngIf="form.get('confirmPassword')?.errors?.passwordMismatch && form.get('confirmPassword')?.touched">
  <small>Passwords do not match.</small>
</div>
```

““

““

Step 4: Create the Template (Continued)

app.component.html:

html

““

```
  <div>
    <button type="submit" [disabled]="form.invalid">Submit</button>
  </div>
</form>

<div *ngIf="form.valid && form.touched">
  <h3>Form Values:</h3>
  <pre>{{ form.value | json }}</pre>
</div>
</div>
```

Explanation of the Template

- **Form Controls**: We have two input fields for the password and confirm password, each bound to the respective form controls using **formControlName**.
- **Validation Messages**: We display validation messages conditionally based on the state of the form controls. For example, if the password is required and not filled, we show a message indicating that.

- **Custom Validation Message**: If the passwords do not match, we display a specific message for that validation error.
- **Submit Button**: The submit button is disabled if the form is invalid.
- **Form Values Display**: If the form is valid and has been touched, we display the form values in JSON format.

Step 5: Run Your Application

Now that you have set up the custom validator and the form, you can run your Angular application.

1. Make sure your development server is running. If it's not, you can start it with:

bash

""

1ng serve

2. Open your browser and navigate to **http://localhost:4200**.

Step 6: Test the Custom Validation

- **Valid Input**: Enter a valid password (at least 6 characters) in the password field and the same password in the confirm password field. When you submit the form, you should see the form values displayed below.
- **Invalid Input**: Try entering different passwords in the password and confirm password fields. You should see the "Passwords do not match." message. If you leave the password field empty, you should see the "Password is required." message.

Conclusion

You have successfully created a custom validation in Angular 19! This tutorial demonstrated how to:

1. Set up a new Angular project.
2. Create a custom validator function.
3. Implement a reactive form that uses the custom validator.
4. Display validation messages in the template.

Additional Enhancements

You can further enhance your custom validation by:

- **Creating More Complex Validators**: You can create validators that check for specific patterns, such as strong passwords or unique usernames.
- **Using Async Validators**: If you need to validate against a backend service (e.g., checking if a username is already taken), you can create asynchronous validators.
- **Styling Validation Messages**: You can add CSS styles to make validation messages more visually appealing.

Example of an Asynchronous Validator

If you want to create an asynchronous validator, you can do so by returning an observable. Here’s a brief example:

typescript

““

```
import { AbstractControl, ValidationErrors, AsyncValidatorFn } from '@angular/forms';
import { Observable, of } from 'rxjs';
import { map, catchError } from 'rxjs/operators';

export function usernameTakenValidator(existingUser names: string[]): AsyncValidatorFn {
  return (control: AbstractControl): Observable<ValidationErrors | null> => {
    return of(control.value).pipe(
      map(value => {
        return existingUser names.includes(value) ? { usernameTaken: true } : null;
      }),
      catchError(() => of(null))
    );
  };
}
```

You would then use this validator in your form control setup.

Final Note

Custom validation is a powerful feature in Angular that allows you to enforce specific rules and provide a better user experience. By following this tutorial, you should now have a solid understanding of how to implement custom validation in your Angular applications. Happy coding!