Introduction
Lecture 01

CSCI 3230U – Web App Development

Today’s outline

1. Course overview
2. Assessment

Instructor

Dr. Rohollah Moosavi

Email: rohollah.moosavi@ontariotechu.ca

Office hours (by appointment):

UA 3035 & Google Meet (https://meet.google.com/zsy-zjso-awa)

Instructor

About me:

- Bachelor's degree (B.Sc.) and a Master's degree (M.Sc.) in Computer Software Engineering
- Ph.D. in Computer Science-Computer Vision
- Postdoctoral Fellowship in Computer Science-Computer Vision from McMaster University & University of Waterloo
- Former Assistant Professor at Azad University, Tehran, Iran (2005 – 2017)
- Teaching Faculty at Ontario Tech University
- Research interests: Medical Image Processing, Medial Data Analysis, Machine Learning, Deep Learning, Precision Medicine
- Former Senior Software Engineer at AGFA Healthcare
- …

Course Outline

A detailed outline of the course, dates, and assessment, and other policies can be found in the **course outline**.

You'll find it on the Canvas course page.

Material covered

Lecture notes: Online through Canvas

No required textbook, though supplemental readings may be provided at times.

Attendance

Attendance is *expected*, but not required.

Attendance is the biggest predictor of university grades.

Research has repeatedly shown that attendance is more important than standardized admissions test scores (NCEA, ATAR/OP/SAT), study habits, and study skills [1, 2].

Have your laptop ready

- Students learn best when they are involved in the lecture process. This is called ***active learning***.

- In my classes, I incorporate interacting programming elements. These help you to learn the concepts we just covered in the slides.

TA Virtual Office hour

- Teaching Assistants hold virtual office hours using Google Meets.
- Check the Course Outline for their schedule.

Today’s outline

1. Course overview
2. Assessment

Assessment

Item	Quantity	Weight	Graded
Labs	Lab 1 (no mark) Lab 2 to Lab 9 (each 4%)	32%	Percentage
Group project	1	34%	Percentage
Test	1	34%	Percentage

Labs

- Labs will consist of web development programming tasks.
- Face-to-face labs begin from the second week, and more info will be posted.
- Attendance is not mandatory.

Group Project

- This group project is designed for you to demonstrate the skills that you have learned in this course.
- The final project that you submit in the last week of classes will be a completed web application.
- You will be responsible for forming your own teams (max 5 people per team). You will then formally register your team membership.

Ethics in computing

- OntarioTech's Academic Integrity policies and processes provide the foundation for expected student behaviour in Computer Science.
- Computer science is a collaborative endeavour. For that reason, we both encourage and require collaboration with your peers.
- But when studying you should avoid collusion, plagiarism, contract cheating, and other forms of academic dishonesty.

Collaboration vs Collusion

- **Collaboration** involves two or more students working together on a piece of assessment. It is a permitted group activity.
- It involves the discussion of class problems with other students. However, each student must write-up and submit their own work.
- **Collusion** also involves two or more students working together on a piece of assessment. This activity is not permitted.
- It involves the sharing and submission of work as it if were your own work product.
- Generally, it's good to collaborate for learning, but not for assessment.

Object Oriented Development (Java) Lecture 02

CSCI 3230U – Web App Development

Today’s outline

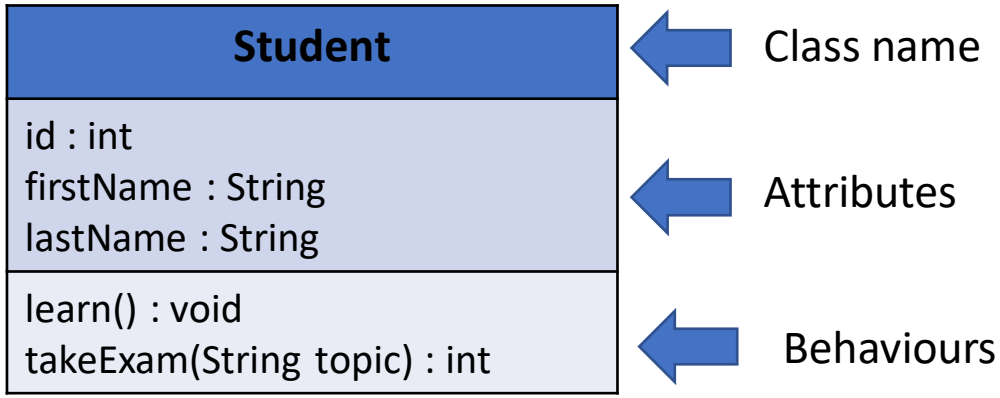
1. Classes
2. Encapsulation
3. Abstraction
4. Dependencies
5. Static Keyword
6. Inheritance-classes

Today’s outline

1. Classes
2. Encapsulation
3. Abstraction
4. Dependencies
5. Static Keyword
6. Inheritance-classes

Class

- A **blueprint** or **template** used to create objects.
- Many instances/objects can be created from a single class.
- Defines **attributes** and **behaviours** for the objects.
- A class diagram is a UML representation of a class:



Creating a class and objects

```
public class ClassName {
    // attributes (variables and constants)
    int attributeName;

    // behaviours (methods)
    public void methodName() {}
}
```

Create an object of a class:

```
Student student = new Student();
```

Variable scope within classes

- **Class level attributes** are visible throughout the class.
- **Method level fields** can only be accessed within the method.

```
public class ClassName {

    int classLevelAttribute;

    public void methodName(int methodLevelField) {
        String anotherMethodLevelField = “”;
    }

}
```

Today’s outline

1. Classes
2. Encapsulation
3. Abstraction
4. Dependencies
5. Static Keyword
6. Inheritance-classes

The 4 pillars of Object-Oriented Development

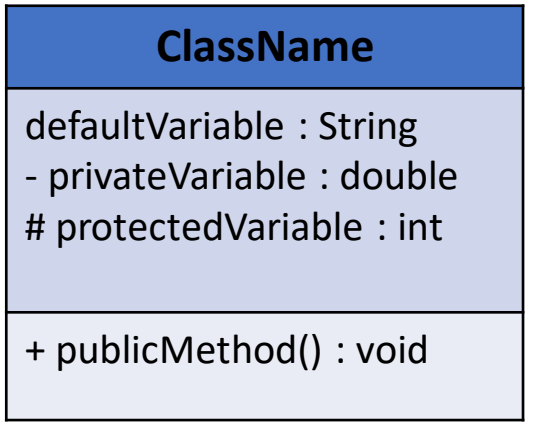
- Encapsulation
- Abstraction
- Inheritance
- Polymorphism

Encapsulation

- A class groups together related attributes.
- The class controls access to its attributes.
- The class controls the values which can be given to its attributes.

Encapsulation – Access modifiers

Determine which objects can access attributes and methods of a class:



- **Default** (no modifier) – any object of a class in the same package can access.

```
int variableName;

void methodName();
```

- **Public** – any object can access.

```
public void methodName();
```

- **Private** – can only be accessed by methods within its own class.

```
private int variableName
```

- **Protected** – can be accessed by objects of classes in the same package and child classes.

```
protected int variableName
```

Today’s outline

1. Classes
2. Encapsulation
3. Abstraction
4. Dependencies
5. Static Keyword
6. Inheritance-classes

Abstraction

- Complexity of behaviour is hidden and We don’t need to know anything about the code within a method.
- Knowing a method’s name, argument types and return type is **sufficient** to use it.

Today’s outline

1. Classes
2. Encapsulation
3. Abstraction
4. Dependencies
5. Static Keyword
6. Inheritance-classes

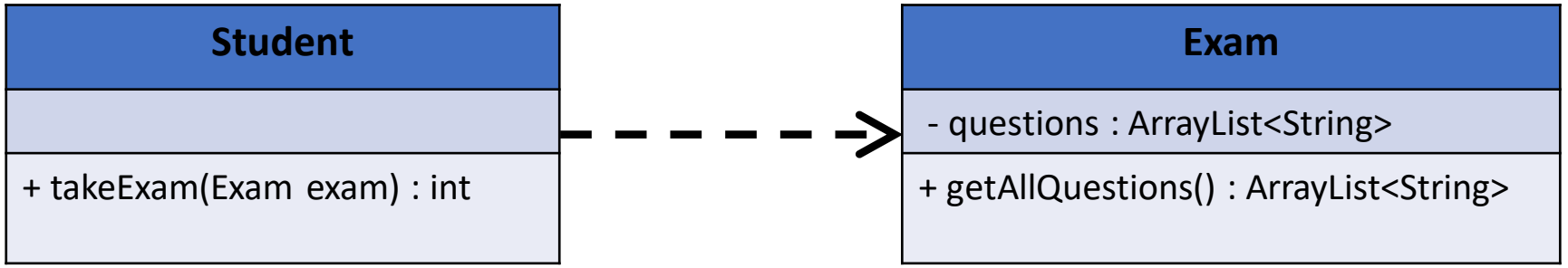
Dependency

- Dependency means an **object** of one class **relies** on an object of another class.

- There are 4 types of dependency:
  - Dependency
  - Association
  - Aggregation
  - Composition

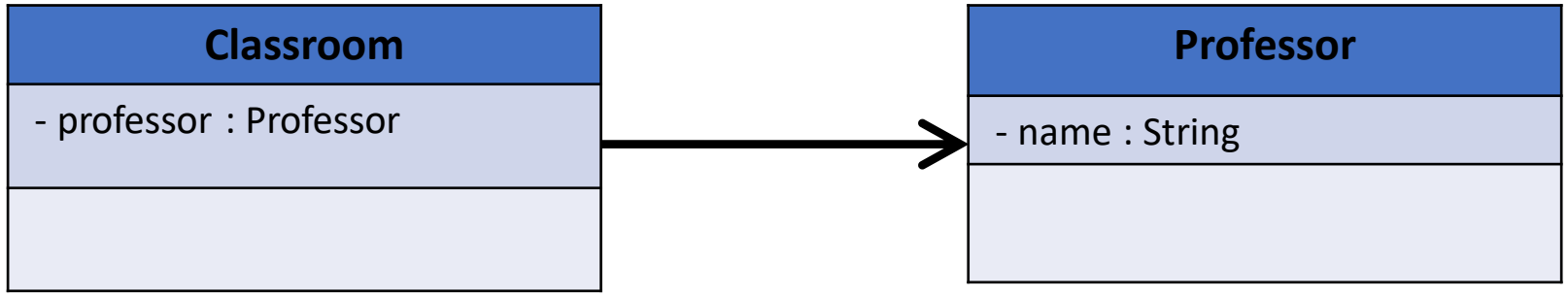
Dependency

- One class uses an object of another class as a method argument or return type.



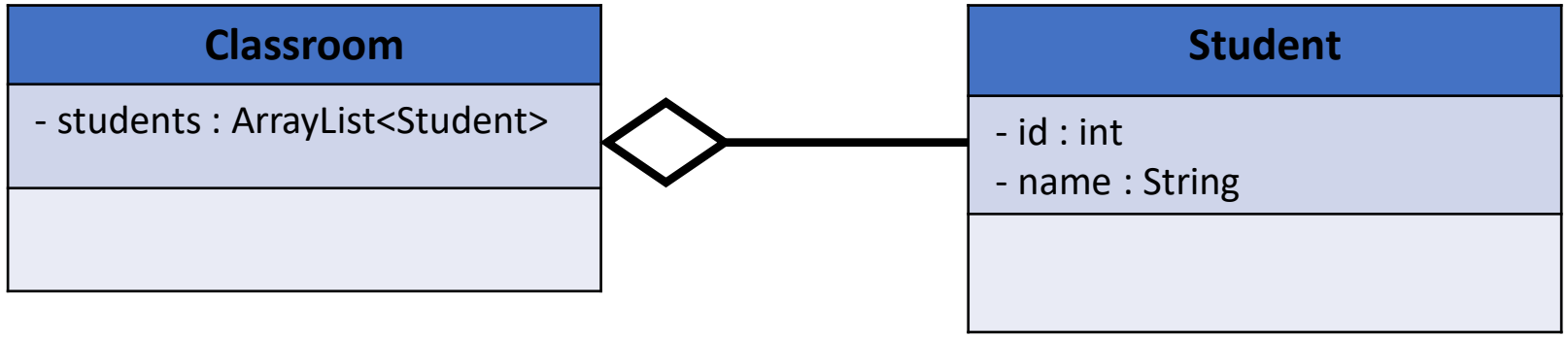
Association

- One class has an **object** of another class **as an attribute**.



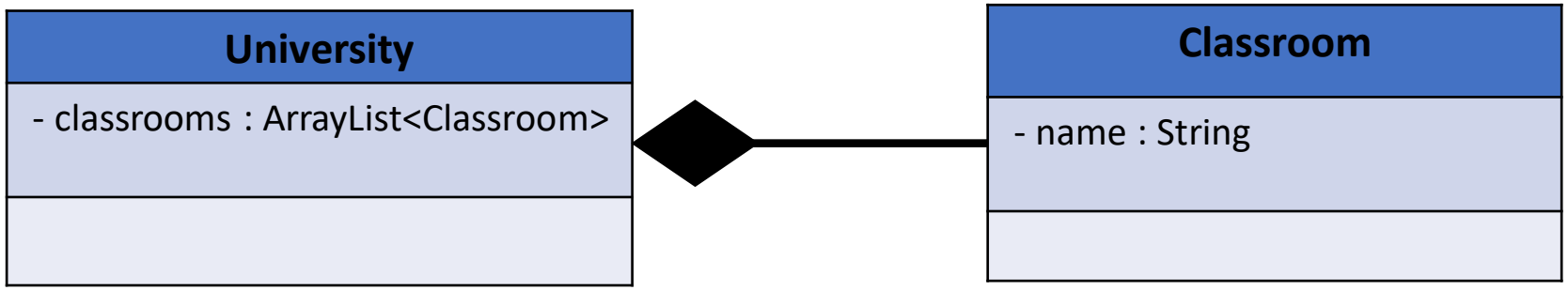
Aggregation

- One class has **multiple objects** of another class as **attributes**.



Composition

- An object of one class belongs to an object of another class and can’t exist without it.



Today’s outline

1. Classes
2. Encapsulation
3. Abstraction
4. Dependencies
5. Static Keyword
6. Inheritance-classes

Static Keyword

- The static keyword is used to denote that an attribute or method belongs to the class itself, rather than an instance of a class.
- Static keyword can be applied to methods, attributes, and even classes.

Static Keyword

- In **methods**, the static keyword must be used somewhere before the return type:

```
public static void methodName() {}

static public void methodName() {}
```

- In **variables**, the static keyword must be used somewhere before the data type:

```
private static int variableName;

static private int variableName;
```

- Static variables and methods are **accessed directly from the class**:

```
ClassName.methodName();

ClassName.getVariableName();
```

Static variable example

- A single variable value needs to be shared amongst all objects of the class.

```
public class ParkingSpace {

	private static int numberOfAvailableSpaces = 0;

	public ParkingSpace() {
		numberOfAvailableSpaces ++;
	}

	public static int getNumberOfSpaces() {
		return numberOfAvailableSpaces;
	}
}
```

ParkingSpace
- numberOfAvailableSpaces : int - spaceId : int
+ getNumberOfAvailableSpaces() : int + getSpaceId() : int

Static constant example

- Where a **constant** has a hardcoded value, **static** will save memory by only storing a single copy of the value.

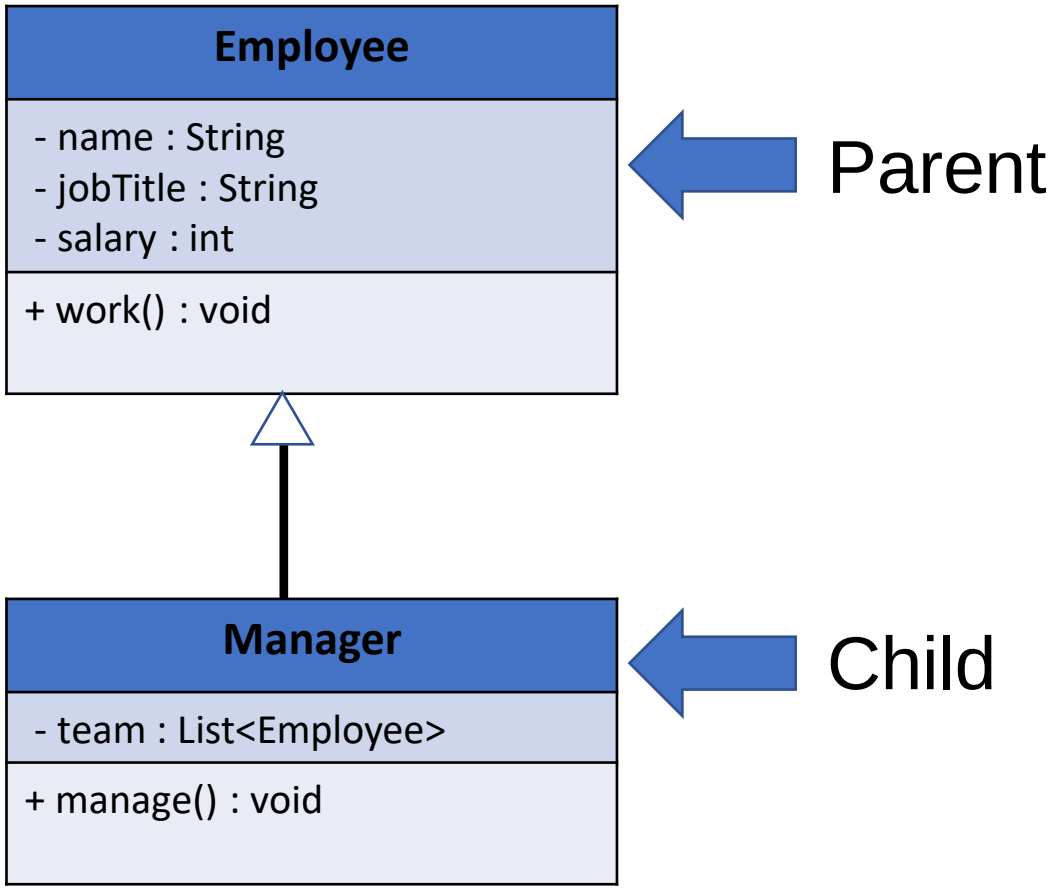
```
public class Circle {

	private final static double pi = 3.1415;

	public static double getPi() {
		return pi;
	}
}
```

Today’s outline

1. Classes
2. Encapsulation
3. Abstraction
4. Dependencies
5. Static Keyword
6. Inheritance-classes

Inheritance-Class

One class (the child) acquires the attributes and behaviours of another class (the parent)



Inheritance-Class

- The '**extends**' keyword defines the parent of a class

```
public class Manager extends Employee {}
```

- A class can **only have one parent** and a class can **have multiple children**
- A child class inherits all attributes and behaviours of its parent

Inheritance-Class

- **Advantage**:
  - Increase Reusability (less development and maintenance costs)
  - Enhance Reliability (the base class code is already tested and debugged)
  - Reduce code Redundancy

- **Disadvantage**:
  - Inheritance works slower than normal one.
  - Increase coupling between base class and derived class (a change in base class will affect all the child class), often data member in base class are left unused that lead to waste memory

Final classes

- The final keyword prevents a class from being extended.

```
public final class ChildlessClass{}
```

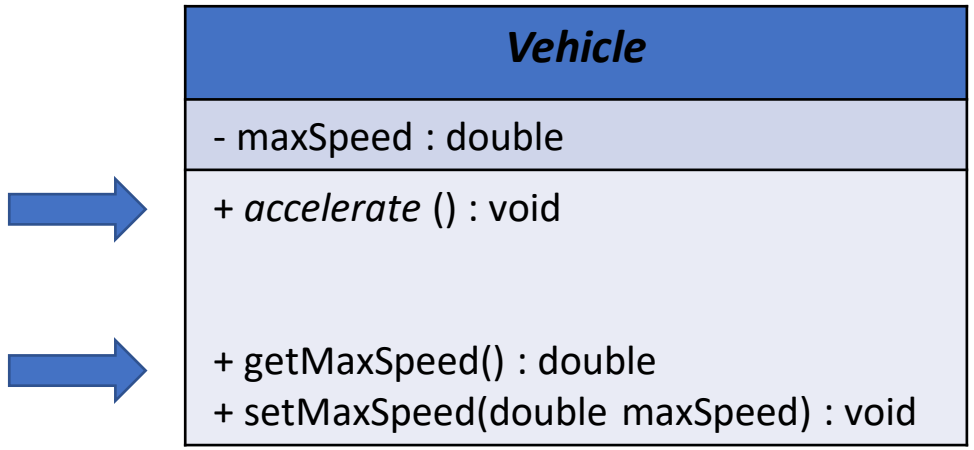
Abstract classes

- A class which **can't be instantiated**
- Exists for **inheritance purposes only**
- Defines the **top of a hierarchy** of classes
- Can contain **abstract** and **non-abstract methods**
- Cannot use the 'final' keyword in its header

```
public abstract class Vehicle {}
```

Abstract method

Non-abstract methods



Abstract methods

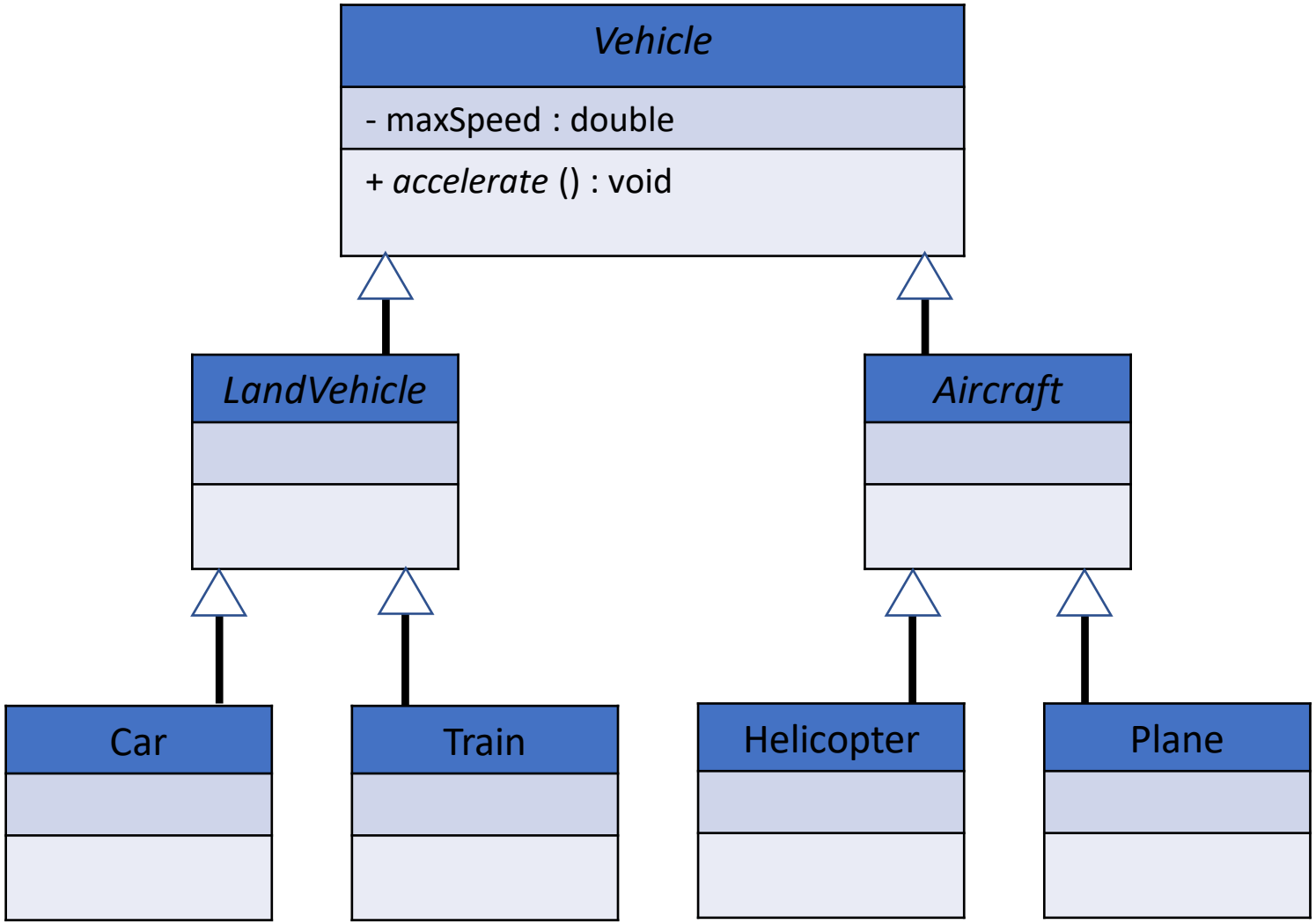
- Abstract methods have no body:

```
public abstract void accelerate();
```

- They must be implemented in a **concrete class** which extends the abstract class:

```
public class Car extends Vehicle {

    @Override
    public void accelerate(){
        // car specific code implementing the method
    }

}
```

Creating a hierarchy of classes



Object Oriented Development (Java) Lecture 03

CSCI 3230U – Web App Development

Today’s outline

1. Inheritance-Interfaces
2. Polymorphisim
3. Architecture

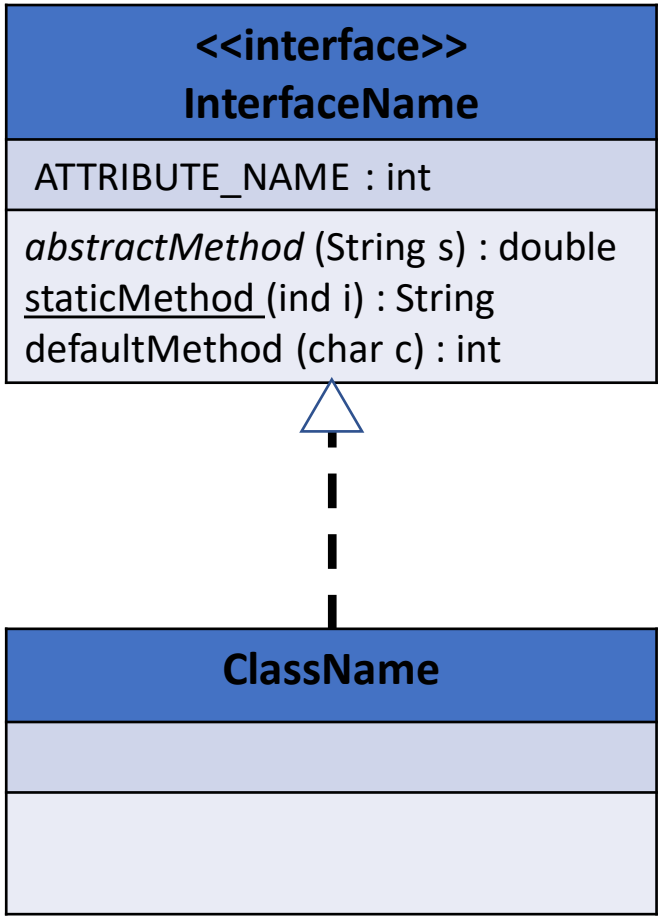
Today’s outline

1. Inheritance-Interfaces
2. Polymorphism
3. Architecture

Interface

- Interface defines common behaviors for different classes and it is another form type of inheritance to using abstract classes.

- In java, a class can inherit from multiple interfaces, but cannot inherit from multiple classes.



Interface

- In interface, we have 3 types of methods: **Abstract**, **default** & **static**
- Methods are **implicitly abstract** when no modifier is used
- All **methods** are **implicitly public**
- All **attributes** are **implicitly public**, **final** and **static**
- Don’t have constructors
- Can’t be instantiated

Creating an interface

```
public interface InterfaceName {

    // behaviours (methods)
    void abstractMethodName (int x);
    default void defaultMethodName(){
        //body of default method
    }
    static void staticMethodName() {
        //body of static method
    }
}
```

Implementing an interface

```
Public class ClassName implements InterfaceName {


    // behaviours (methods)
     void abstractMethodName (int x){
        //code implementing the abstract method
    }
}
```

Implementing an interface

```
interface FirstInterface {
  public void myMethod(); // interface method
}

interface SecondInterface {
  public void myOtherMethod(); // interface method
}

class DemoClass implements FirstInterface, SecondInterface {
  public void myMethod() {
    System.out.println("Some text..");
  }
  public void myOtherMethod() {
    System.out.println("Some other text...");
  }
}

class Main {
  public static void main(String[] args) {
    DemoClass myObj = new DemoClass();
    myObj.myMethod();
    myObj.myOtherMethod();
  }
}
```

https://www.w3schools.com/java/java_interface.asp

Today’s outline

1. Inheritance-Interfaces
2. Polymorphism
3. Architecture

Polymorphism

- A method can have many forms.

- There are two types of polymorphism:
    1. **Overriding**
        - Also known as late binding or runtime polymorphism
    2. **Overloading**
        - Also known as early binding or compile time polymorphism

Overriding

- A **sub class** has a **different implementation** of a method inherited from its super class.
- The sub class method must have the same name and input arguments as the super class method

```
public class Vehicle {
      public void accelerate(){
            // generic code to accelerate a vehicle
      }
}

public class Car extends Vehicle {
      @Override
      public void accelerate(){
            //car specific code
      }
}
```

Overriding

- We use super keyword to access a method in the super class, if needed.

```
public class Vehicle {
     public void accelerate(){
          // generic code to accelerate a vehicle
     }
}

public class Car extends Vehicle {
     public void accelerate(){
          super.accelerate();
     }
}
```

Overriding

- We can use final keyword to prevent a method from being overridden in any subclasses.

```
public class Car extends Vehicle {

    public final void accelerate(){
        // car specific code
    }
}
```

Overloading

- A class has multiple methods with the same name but with different arguments.
- Each method will have a different functionality.
- The specific method will be called depends on the arguments passed in.

```
class Shapes {
  public void area() {
    System.out.println("Find area ");
  }

  public void area(int r) {
    System.out.println("Circle area = "+3.14*r*r);
  }

  public void area(double b, double h) {
    System.out.println("Triangle area="+0.5*b*h);
  }

  public void area(int l, int b) {
    System.out.println("Rectangle area="+l*b);
  }
}
```

Today’s outline

1. Inheritance-Interfaces
2. Polymorphism
3. Architecture

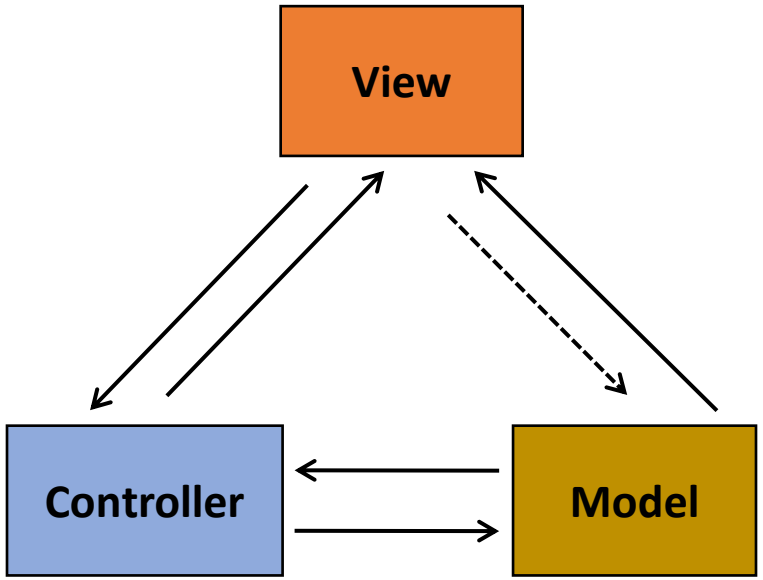
MVC

MVC is one of the most common application architecture design patterns.

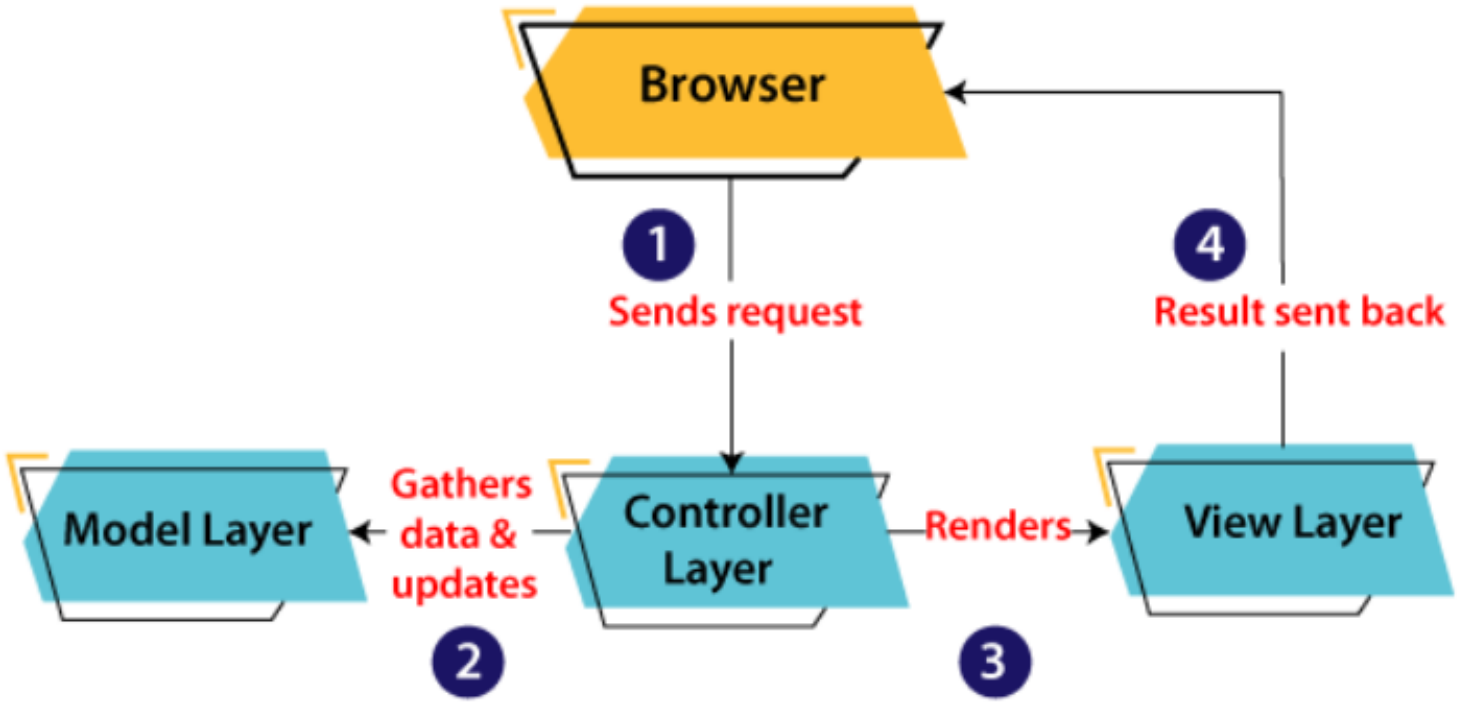
Model – the business logic

View – the user interface

Controller – manages Model and View interaction



MVC


Browser
1
Sends request
4
Result sent back
Model Layer
Gathers
data &
updates
Controller
Layer
Renders
View Layer
2
3

Object Relational Mapping (ORM)
Java Persistence API (JPA)
Lecture 04

CSCI 3230U – Web App Development

Today’s outline

1. Persistence
2. ORM
3. JPA

Today’s outline

1. Persistence
2. ORM
3. JPA

Persistence

- Persistence means, continuous saving data and maintain.
- Almost all applications need to persist data.
- There are several ways to persist data, like storing in File, database, etc.
- Using database is the best way to persist data (portability, vast storage, different machine, etc.)

Persist Data in Java

- Relational DB is one of the best way to persist data.
  - Java DataBase Connectivity **(JDBC)** is an API that allows Java application to interact with databases.
  - Java Persistence API (**JPA**) is a Java specification for accessing, persisting, and managing data between Java objects/classes and a relational database.

Today’s outline

1. Persistence
2. ORM
3. JPA

Object Relational Mapping (ORM)

- **ORM** stands for Object-Relational Mapping is a programming technique for converting data between relational databases and object oriented programming languages such as Java, C#, etc.

- **ORM** is a technique that maps object state to data in a relational database.

- There are several **persistent frameworks** and ORM options in Java, like Hibernate, Spring DAO, etc.
    - A **persistent framework** is an ORM service that stores and retrieves objects into a relational database.

Today’s outline

1. Persistence
2. ORM
3. JPA

Java Persistence API (JPA)

- The **Java Persistence API** is the ORM solution in Java EE

- **Hibernate** is an implementation of **JPA**

JPA-Entities

- We want to persist **Entity classes**.

- **Objects** can be an **entity**.

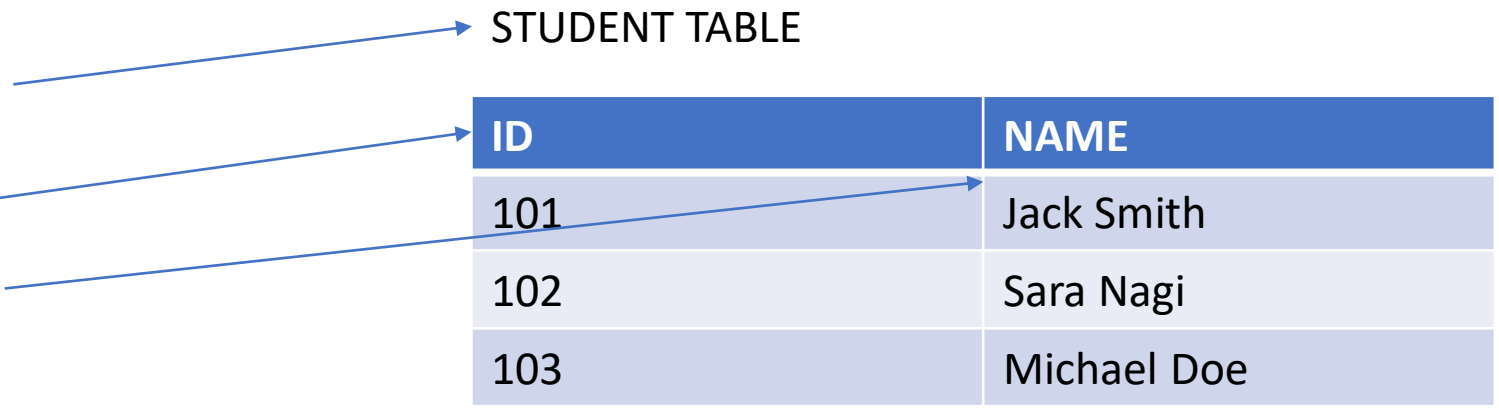
- **Metadata** (annotation) relates entities to records in database.
    - Like @Entity, @id, etc.

Model Example

```
public class Student{

    private int id;
    private String name;

    public Student(){

    }

    // setters and getters

}
```

Mapping Student to Table

```
@Entity
public class Student{
    @Id
    private int id;
    private String name;

    public Student(){

    }

    // setters and getters

}
```



STUDENT TABLE

ID	NAME
101	Jack Smith
102	Sara Nagi
103	Michael Doe

Mapping Student to Table

- We can override default mapping.

```
@Entity
@Table(name = “STD”)
public class Student{
    @Id
    @Column(name = “STD_ID”)
    private int id;

    @Column(name = “STD_NAME”)
    private String name;

    public Student(){ }

    // setters and getters
}
```

STD TABLE

STD_ID	STD_NAME
101	Jack Smith
102	Sara Nagi
103	Michael Doe

H2 Database setup

- Step 1:
    - https://www.h2database.com/html/download-archive.html
        - Preferably download version 1.4.197

JPA Implementation steps

- Step 2:
  - Configure Entity classes with annotation

```
@Entity
public class Student{
    @Id
    private int id;
    private String name;

    public Student(){

    }

    // setters and getters

}
```

JPA Implementation steps

- Step 3:
  - Provide configuration details in **persistence.xml** file
    - It’s an XML file that defines **persistence units**
      - **Persistence unit** is used to configure Entity classes and has database connection details, other properties.
    - Should be placed in a directory called META-INF

JPA Implementation steps

Step 4:

Create an EntityManagerFactory

```
EntityManagerFactory emFactory =
        Persistence.createEntityManagerFactory("persistenceUnitName");
```

Step 5:

Create an EntityManager instance from EntityManagerFactory

```
EntityManager em = emFactory.createEntityManager();
```

JPA Implementation Demo

JPA Example

Transient

```
@Entity
@Table(name = “STD”)
public class Student{
    @Id
    @Column(name = “STD_ID”)
    private int id;

    @Column(name = “STD_NAME”)
    private String name;

    @Transient
    private int age;

    public Student(){ }

    // setters and getters
}
```

STD TABLE

STD_ID	STD_NAME	DOB
101	Jack Smith	11 - DEC - 1992
102	Sara Nagi	05 - DEC - 1990
103	Michael Doe	16 - DEC - 1995

Object Relational Mapping (ORM)
Java Persistence API (JPA)
Lecture 05

CSCI 3230U – Web App Development

Today’s outline

1. Managing Entities
2. Query
3. Java Persistence Query Language (JPQL)
4. Primary Key Generation

Today’s outline

1. Managing Entities
2. Query
3. Java Persistence Query Language (JPQL)
4. Primary Key Generation

Managing Entities - Managed/Detached

When an EntityManager gets a reference to an Entity, that instance is said to be managed, so EntityManager will automatically sync Entity state with the DB

Otherwise, the instance is said to be detached

Managing Entities - Transaction

- We use Transactions:

```
em.getTransaction().begin();

// Make changes to database

em.getTransaction().commit();
```

Managing Entities - Persist

```
Student student = new Student(101, “John”);

em.getTransaction().begin();

em.persist(student)

em.getTransaction().commit();
```

Managing Entities - Find

```
Student student = em.find(Student.class, 101)
```

Managing Entities - Delete

```
// Obtain managed Entity
Student student = em.find(Student.class, 101);

em.getTransaction().begin();

em.remove(student);

em.getTransaction().commit();
```

Managing Entities - Update

```
// Obtain managed
Student student = em.find(Student.class, 101);

em.getTransaction().begin();

// Modify Entity state
student.setName(“Sarah”);

em.getTransaction().commit();
```

Managing Entities - Merge

```
// Obtain managed
Student student = new Student(101);

em.getTransaction().begin();

Student managedStudent = em.merge(student);

em.getTransaction().commit();
```

Today’s outline

1. Managing Entities
2. Query
3. Java Persistence Query Language (JPQL)
4. Primary Key Generation

Query

```
Query query = em.createQuery("SELECT s FROM Student s", Student.class);

List<Student> allStudent = query.getResultList();
```

Today’s outline

1. Managing Entities
2. Query
3. Java Persistence Query Language (JPQL)
4. Primary Key Generation

Java Persistence Query Language (JPQL)

- Methods defined for querying using JPQL:
  - createQuery()
  - createNamedQuery()
  - createNativeQuery()

```
Query query = em.createQuery(“Select s from Student s”, Student.class);
List<Student> students = query.getResultList();
```

Java Persistence Query Language (JPQL)

- **Named Parameters:**

```
Query query = em.createQuery("Select s from Student s WHERE s.name LIKE
:stdName", Student.class);
query.setParameter("stdName", "John");
```

- **Positional Parameters:**

```
Query query = em.createQuery("Select s from Student s WHERE s.name LIKE
?1", Student.class);
query.setParameter(1, "John");
```

Java Persistence Query Language (JPQL)

- Dynamic Queries:
  - **Dynamic queries** are defined directly within an application's business logic.
  - we use createQuery method to create dynamic queries

```
public List<Student> findStudentByName(String sName){
   Query query = em.createQuery("SELECT s FROM Student s WHERE s.name LIKE
   :stdName", Student.class);
   query.setParameter("stdName", sName);

   return query.getResultList();
}
```

Java Persistence Query Language (JPQL)

- Static Queries:
  - **Static queries** are defined in the metadata by using **@NamedQuery annotation**.
  - we use **createNamedQuery** method to create static queries

```
@NamedQuery(
  name = “findStudentByName”,
  query = SELECT s FROM Student s WHERE s.name LIKE :stdName”)
@Entity
Public class Student {
  …
}
```

```
List<Student> students = em.createNamedQuery(“findStudentByName”)
                                        .setParameter(“stdName” , “John”)
                                        .getResultList();
```

Today’s outline

1. Managing Entities
2. Query
3. Java Persistence Query Language (JPQL)
4. Primary Key Generation

Primary Key Generation

- Assign primary field with primary key value generator

```
@Entity
Public class Student {
    @id
    @GeneratedValue(strategy = GenerationType.Auto)
    private int id;

    private String name;
  …
}
```

Primary Key Generation

- We can use another strategy, SequenceGenerator:

```
@Entity
Public class Student {
     @id
    @SequenceGenerator(name = “STD_SEQ_GNR”, sequenceName = “STD_SEQ”)
    @GeneratedValue(strategy = GenerationType.SEQUENCE, generator =  “STD_SEQ_GNR”)
    private int id;

    private String name;
  …
}
```

Hypertext Mark-up Language (HTML)
Cascading Stylesheet (CSS)

Week 05

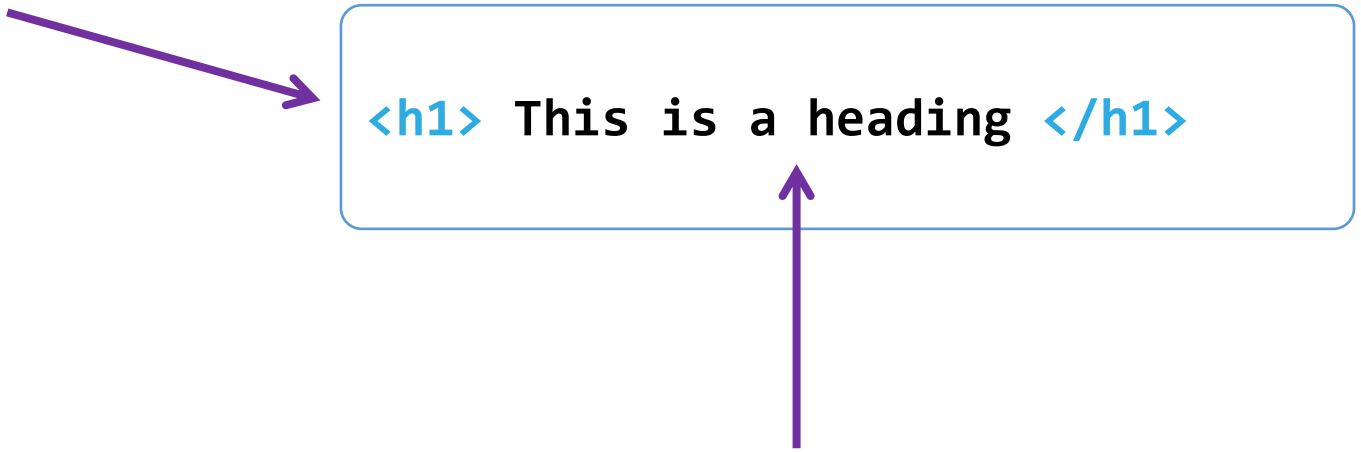
CSCI 3230U – Web App Development

HTML and CSS

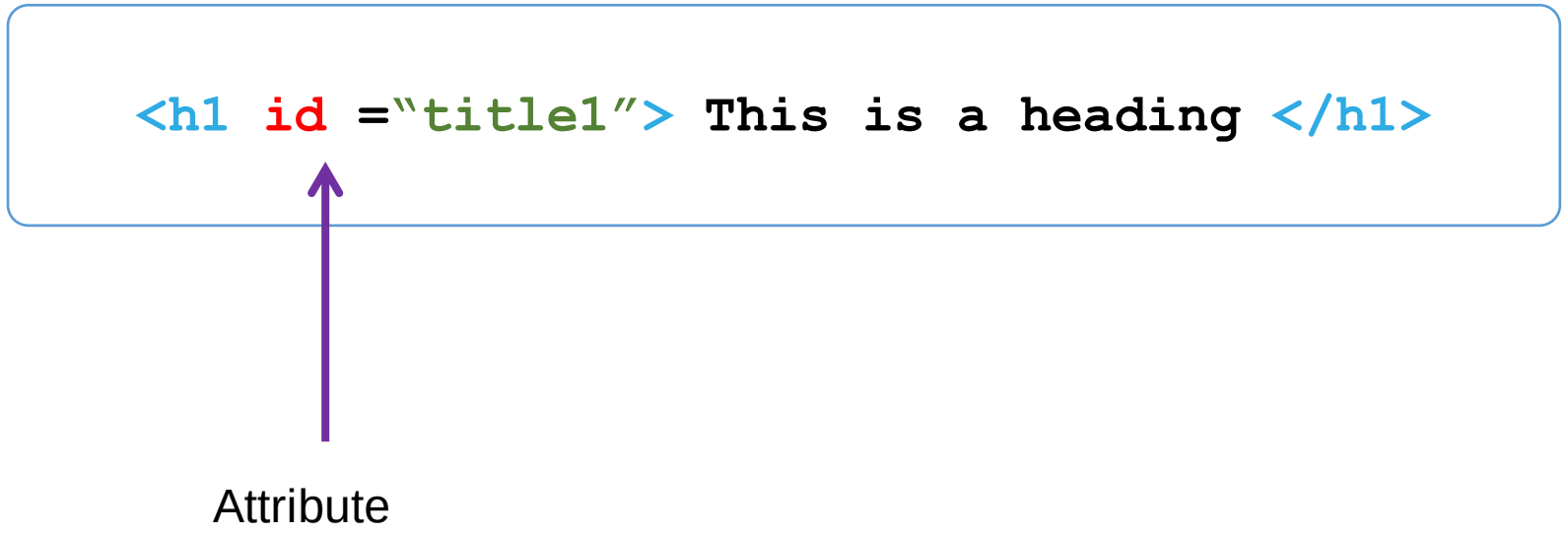
Feature	HTML	CSS
Purpose	Defines the **structure** and **content** of web pages	**Styles** the presentation and layout of web pages
Syntax	Uses **tags** (e.g., <html>, <head>, <body>)	Uses **selectors** and declarations (e.g., .class, color: blue;)
Usage	Provides the basic building blocks for web pages	Enhances the appearance and design of web pages
Elements	Includes **elements** such as , <p>, <a>, <img>	Does not have elements, but defines **styling rules for elements**
Structure	**Hierarchical structure** of nested elements	**Flat structure** of selectors and declarations
Functionality	Determines the content and **semantic meaning** of elements	Determines the **visual presentation** and layout of elements
Responsiveness	Does not inherently provide responsiveness	Allows for responsive design through media queries
Browser Support	Supported universally by web browsers	Supported by modern web browsers with some variations
File Extension	Typically .html	Typically .css
Examples	<html><head><title>Example</title></head><body> Hello, World! </body></html>	.header { font-size: 20px; color: blue; }

HTML

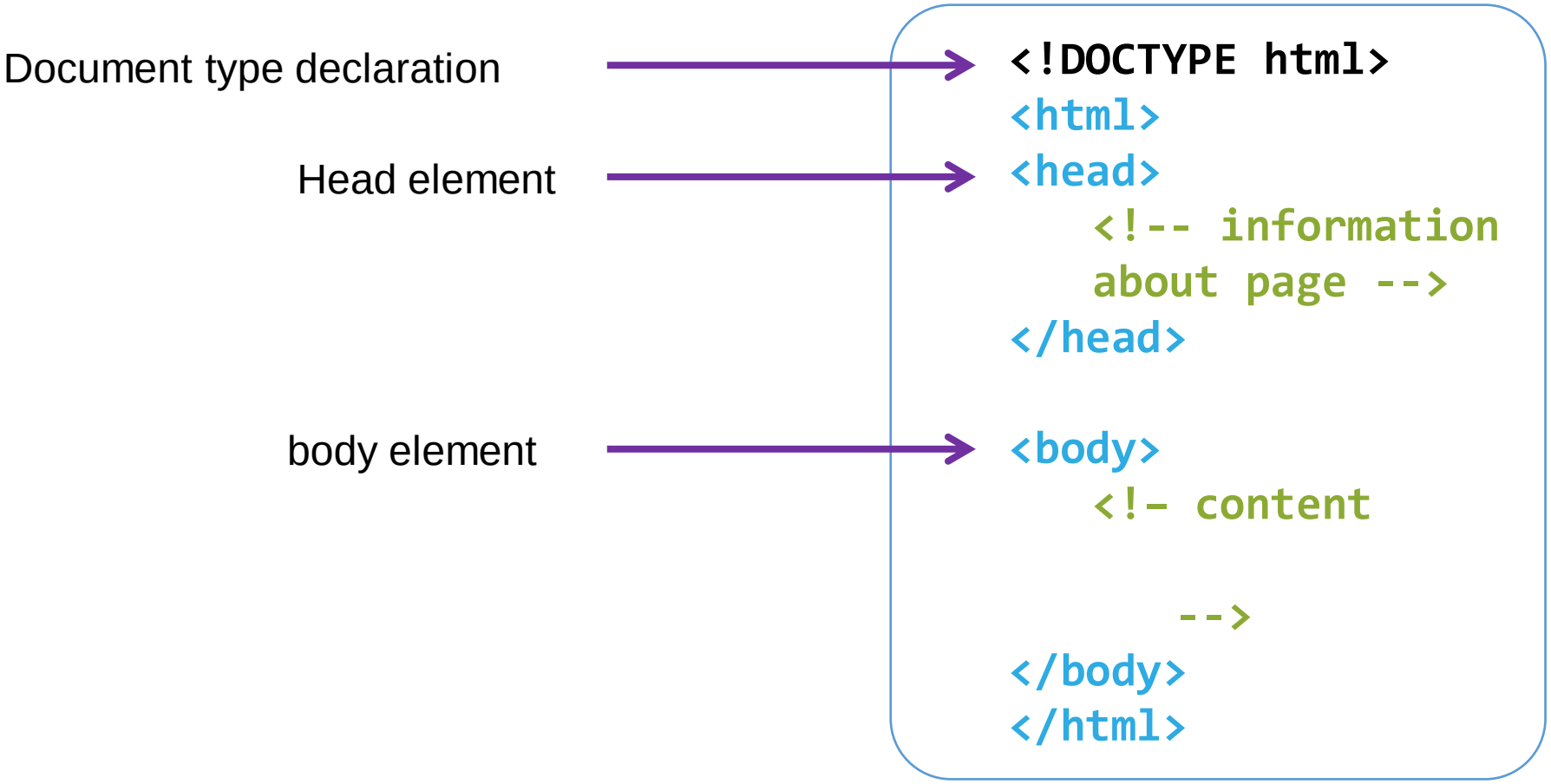
An element example



HTML



HTML 5 Structure



CSS

```
h1 {
    background: green;
    color: yellow;
    text-align: center;
}
```

CSS

```
h1, h3 {
    margin: 20px;
}

.demo {
    background: blue;
}

#title1 {
    background: yellow;
    color: black;
}
```

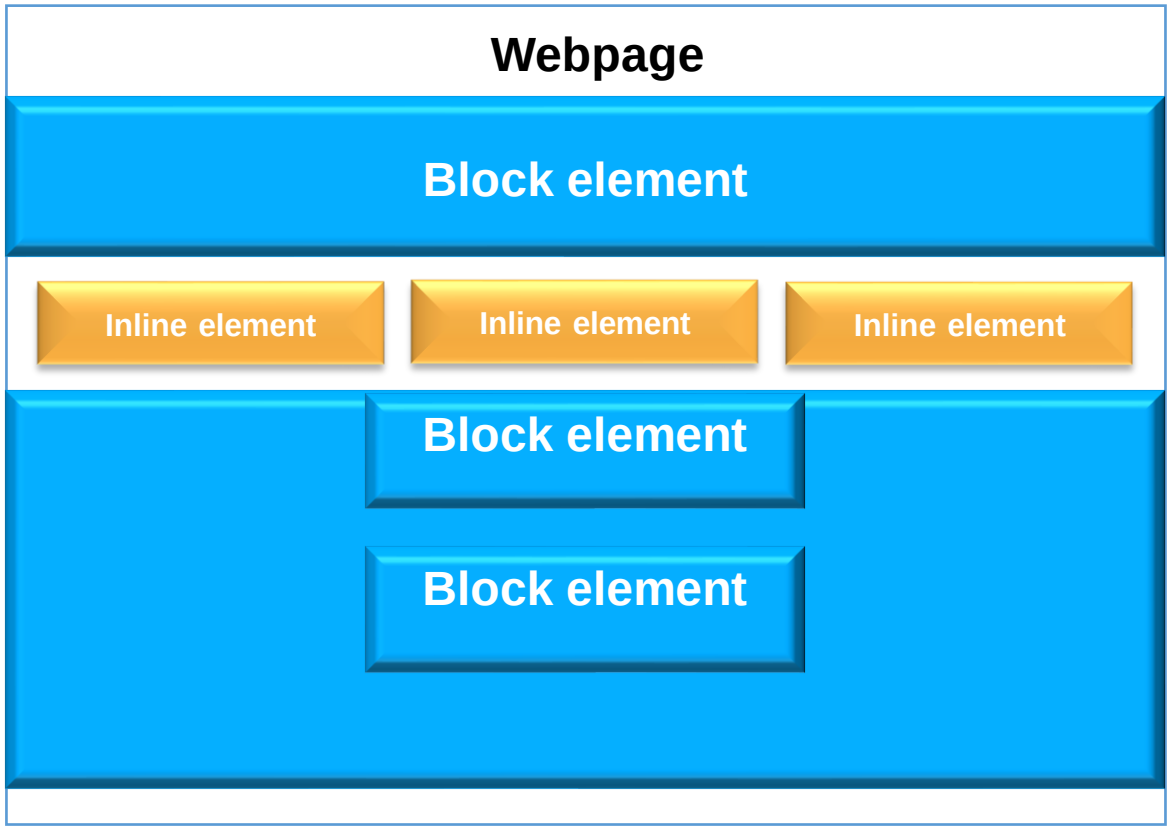
HTML 5 and CSS

```
<!DOCTYPE html>
<html>
<head>
  <title>Basic class</title>
  <link rel="stylesheet"
       href="style1.css">
</head>
<body>
  <p class="demo">
   This is a demo class
  </p>
  <p>
   This is a default
 </p>
</body>
</html>
```

```
.demo {
    background: blue;
    color: green;
    border: 15px solid red;
}

body {
    margin: 25px;
}
```

HTML 5 and CSS

Block and Inline Elements



HTML 5 and CSS

Semantic and Non-Semantic

Feature	Inline Elements	Block Elements	Semantic Elements	Non-Semantic Elements
Definition	Elements that do not force a new line and only take up as much width as necessary for their content	Elements that force a new line and take up the full width available	Elements with specific meanings, conveying information about their content	Elements used for generic styling or layout purposes without conveying specific meaning
Examples	, <a>, ,	, <p>, <h1>-<h6>, ,	<header>, <footer>, <article>, <nav>	,
Usage	Used for styling small portions of text or elements within a block element	Used for structuring larger sections of content or creating layout	Used to give meaning to the content and enhance accessibility	Used for layout purposes or grouping elements for styling purposes without conveying specific meaning

HTML 5 and CSS

HTML & CSS Example

Hypertext Mark-up Language (HTML) Cascading Stylesheet (CSS) (Part 2)

Week 05

CSCI 3230U – Web App Development

Source of Style

Internal

External

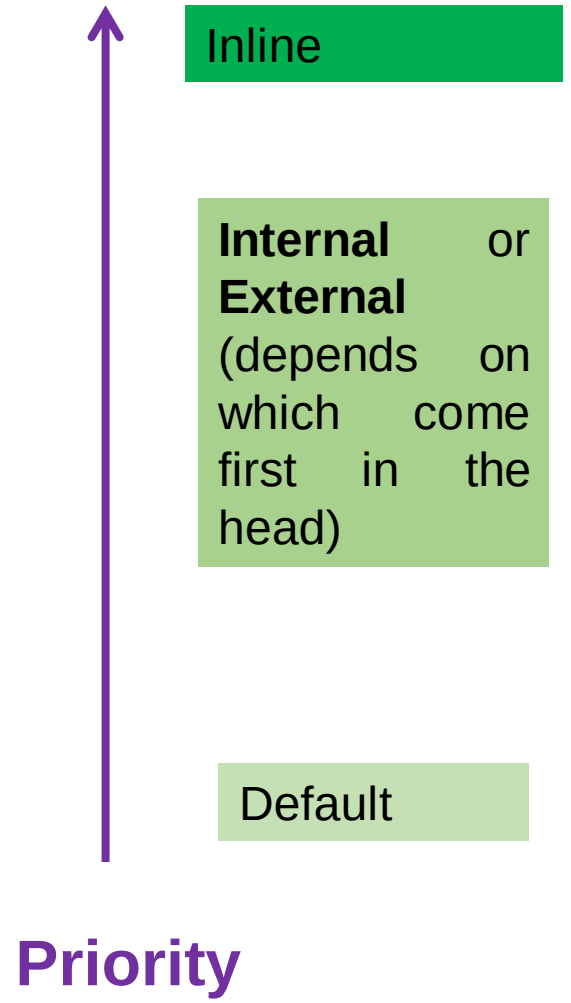
Inline

Default

```
<!DOCTYPE html>
<html>
    <head>
        <style>
            .title {
                color: green;
            }
        </style>

        <link rel="stylesheet" href="styles1.css" />
    </head>
    <body>
        <h1 style="color:red;">Greeting fellows
            </h1>
        <p>this is a paragraph</p>
    </body>
</html>
```

CSS Priority

- When styling an element, different CSS sources can be used, leading to various rules.
- Therefore, the browser needs to figure out which rules are more important than others.
- The CSS source with highest priority, decides which styles get used.
- If a style isn't specified in the highest priority, will be considered with a lowest priority.



CSS Priority

```
<!DOCTYPE html>
<html>
<head>
  <title>Basic class</title>
  <link rel="stylesheet"
      href="style1.css">
</head>
<body>
  <p class="demo" id="demo">
   This is a demo class
  </p>
  <p>
   This is a default
 </p>
</body>
</html>
```

```
.demo {
    background: blue;
    color: green;
    border: 15px solid red;
}

#demo {
    background: green;
    color: red;
    border: 10px solid yellow;
}
```

CSS Specificity

- Once two or more CSS rules apply to the same element, we need to decide which one choose.

- We can use **specificity value**
  - Once more that one rules select the same element, choose the most **specific rule**, the selector with the highest specificity value.

CSS Specificity

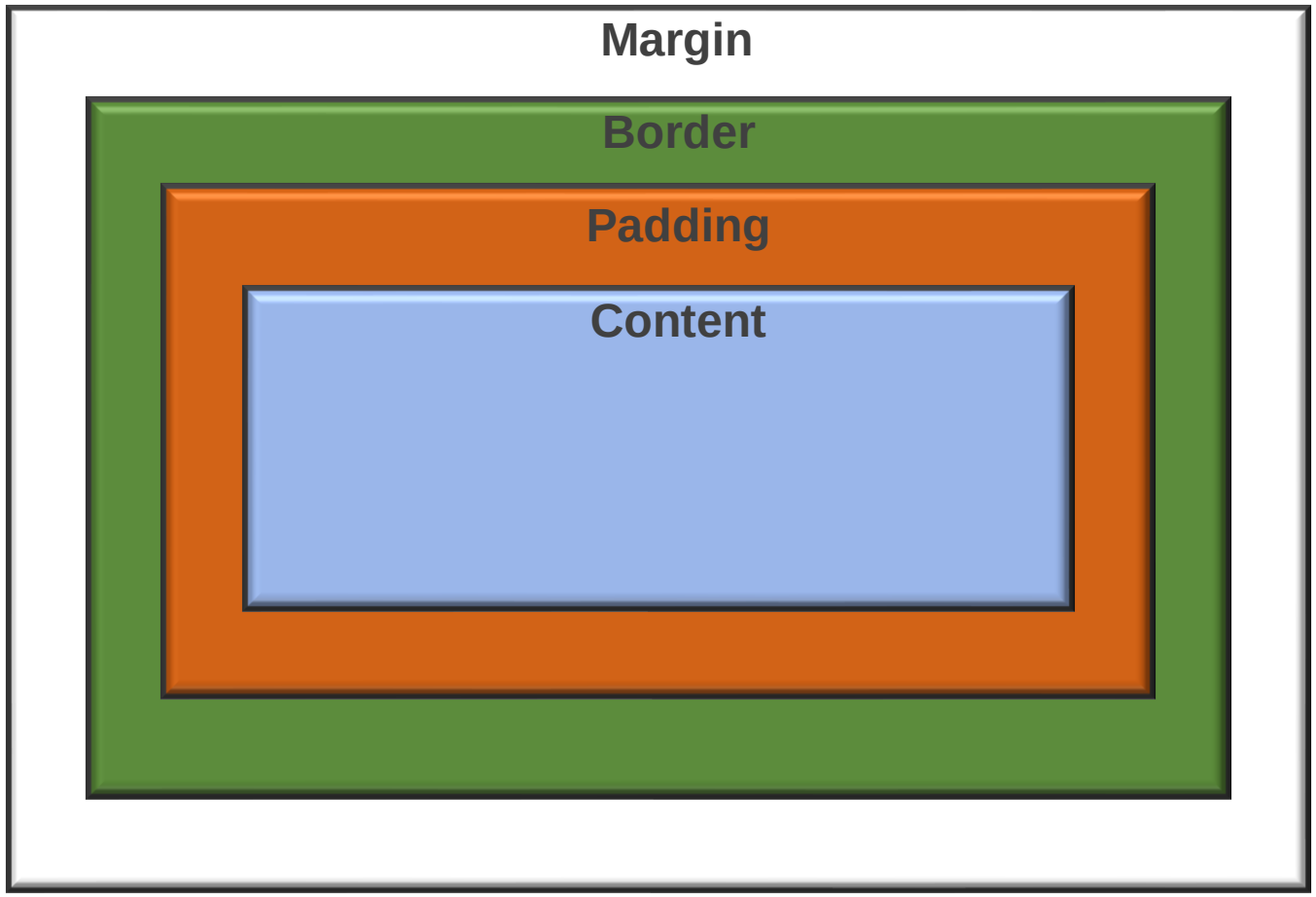
- We can consider 4-digit as follows, such that the higher the number the more specific selector.

CSS Specificity

- Find the order in the following styles:

1. **h1 { … }**
   0001
1. **#title1 { … }**
   0100
2. **h1 .myclass1 { … }**
   0011
3. **<h1 style="color:yellow;">Greeting fellows</h1>**
   1000
4. **header .buttons .externalbuttons .highlight h1 { … }**
   0032

Box Model



Box Model

```
.class-name {
    border: 1px solid blue;
    padding: 5px;
    margin: 5px;
}

.class-name {
    box-sizing: border-box;
}

.class-name {
    box-sizing: content-box;
}
```

Width of the outside of the border

Width of the outside of the content

Grid and Float

Leave it to you

Hypertext Mark-up Language (HTML) Cascading Stylesheet (CSS) (Part 2)

Week 06

CSCI 3230U – Web App Development

Chrome Debugging

- Chrome browser has provided a tool that allows the developer to debug the HTML and CSS.
  - Right-click on the web page and select Inspect option
  - Or select **More Tools > Developer Tools** from Chrome's Main Menu

Chrome Debugging

Practice with Chrome Debugging tool for your own website

Web Design And Development (Part 1)

Week 06

CSCI 3230U – Web App Development

Interaction Design

- **Interaction design** focuses on creating meaningful and efficient interactions between users and digital products or services.
- It includes understanding user needs, designing intuitive interfaces, and optimizing user experiences.

Interaction Design Considerations

- User-Centered Design
- Clear and Consistent Interface
- Responsiveness
- Accessibility
- Error Prevention and Recovery
- Scalability and Adaptability

Information Architecture

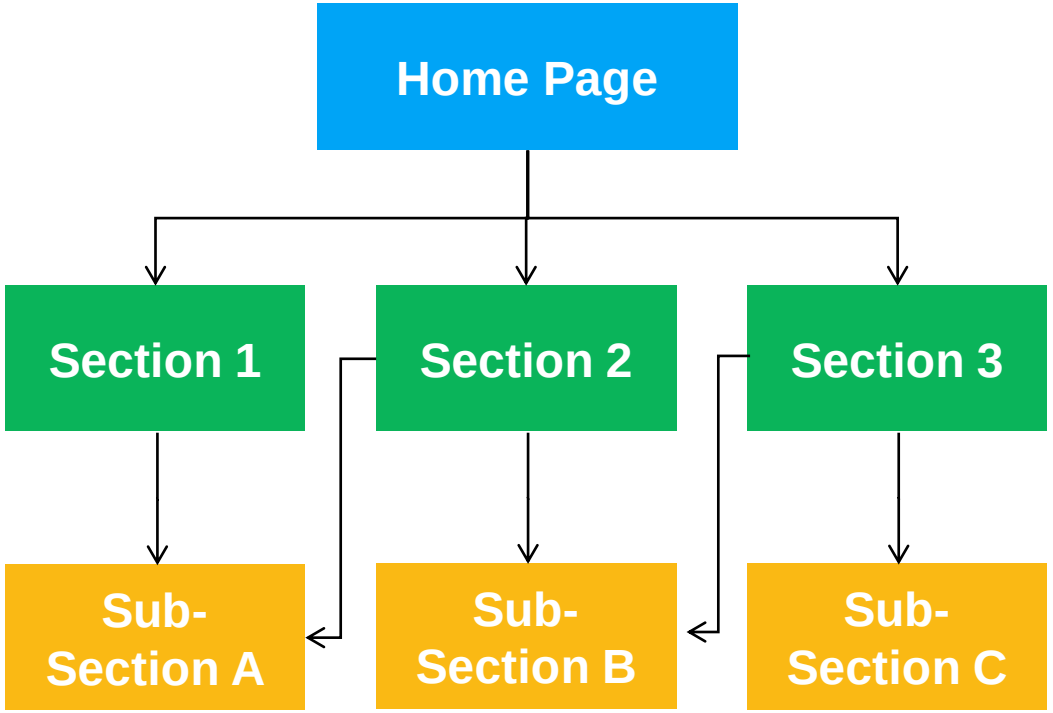
- Searchable
- Well organized
- Easily Navigable
- Clearly labelled

Site Maps

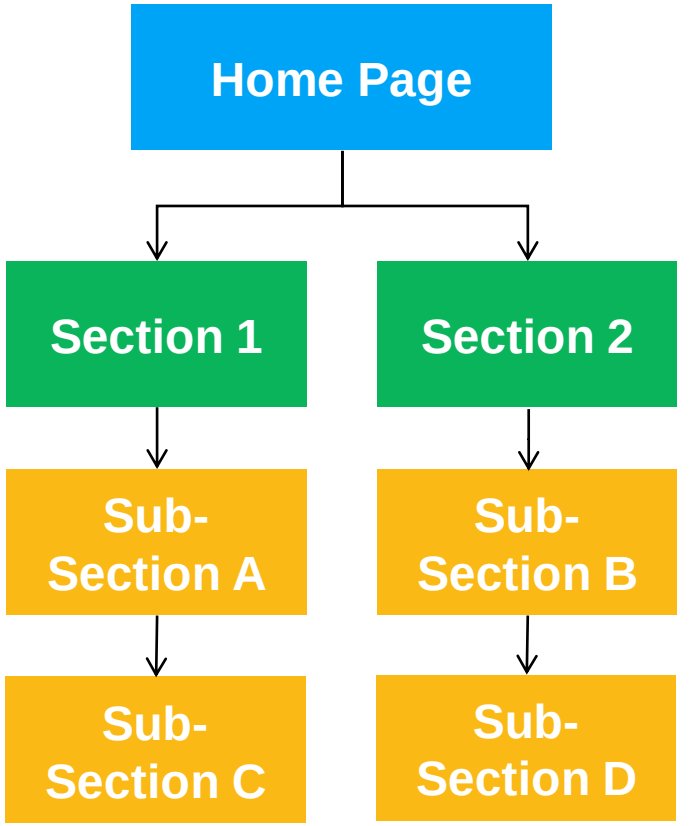
- Site maps are visual representations of a **website's structure**, illustrating the hierarchy of pages and their relationships.
- They serve as **navigational blueprints**, guiding users through digital landscapes and helping them find relevant content efficiently.
- Types of Site Maps:
  - Visual Site Maps
  - XML Sitemaps

Site Maps

- Can be flat or deep in structure.

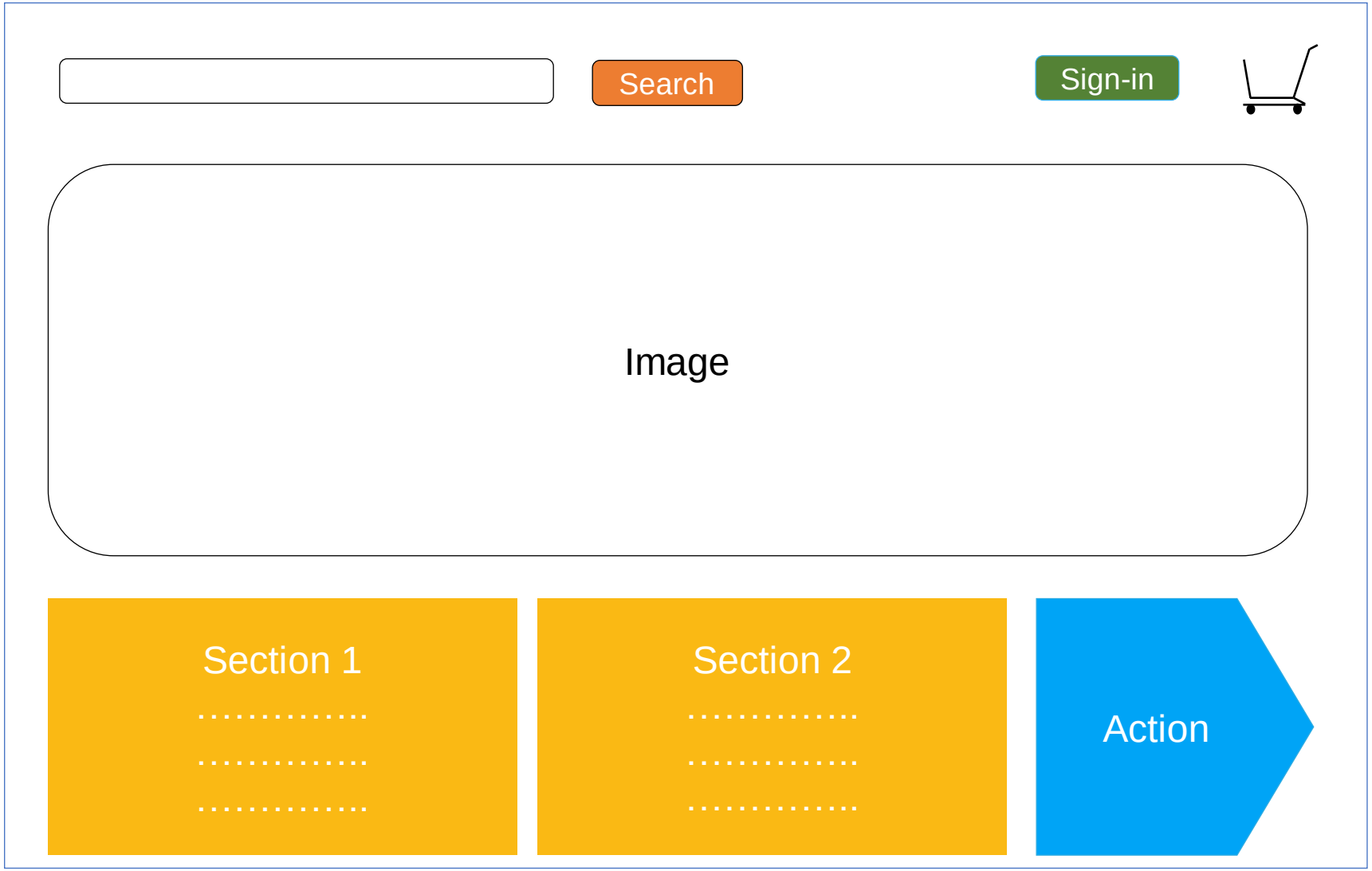


Flat example

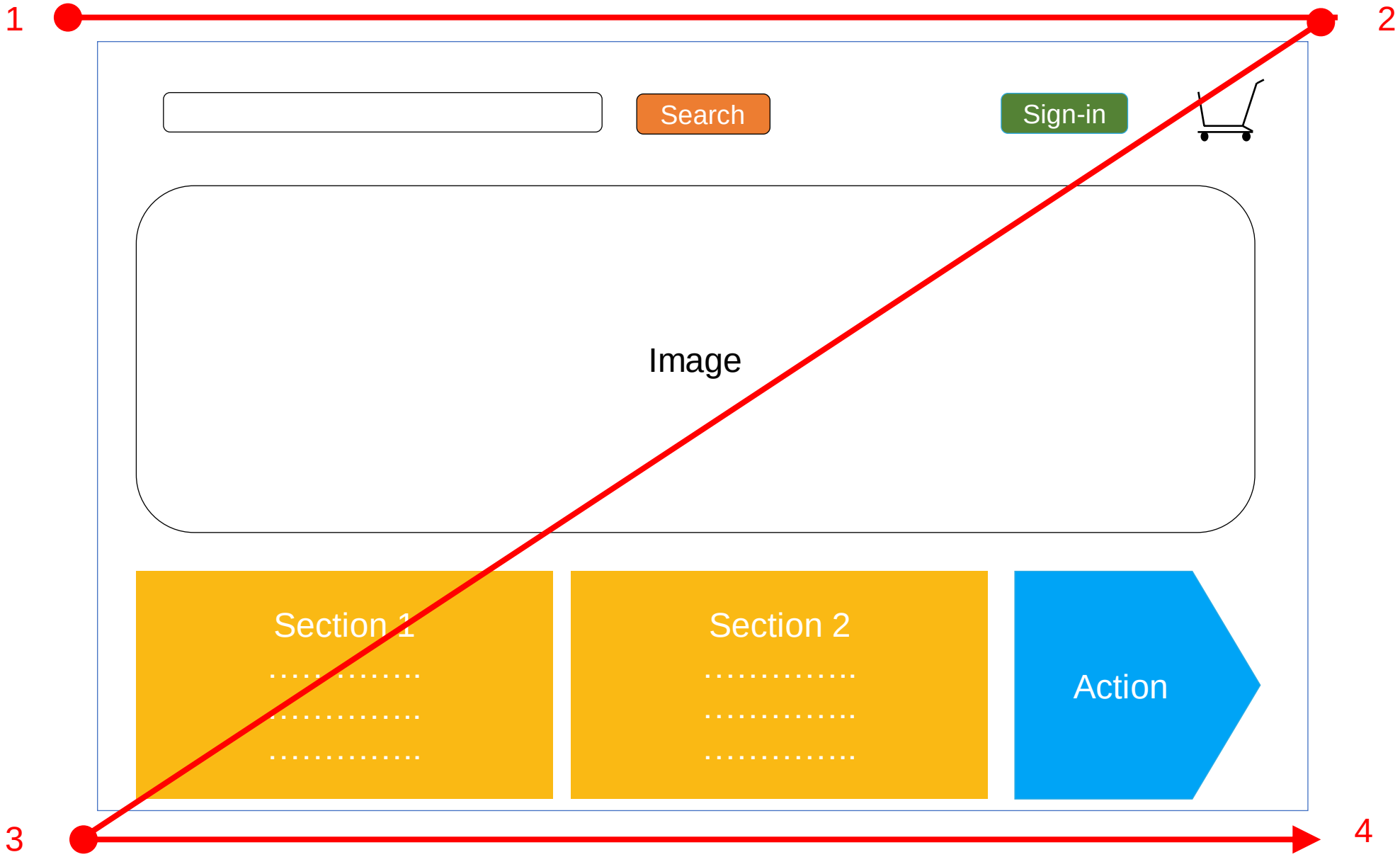


Deep example

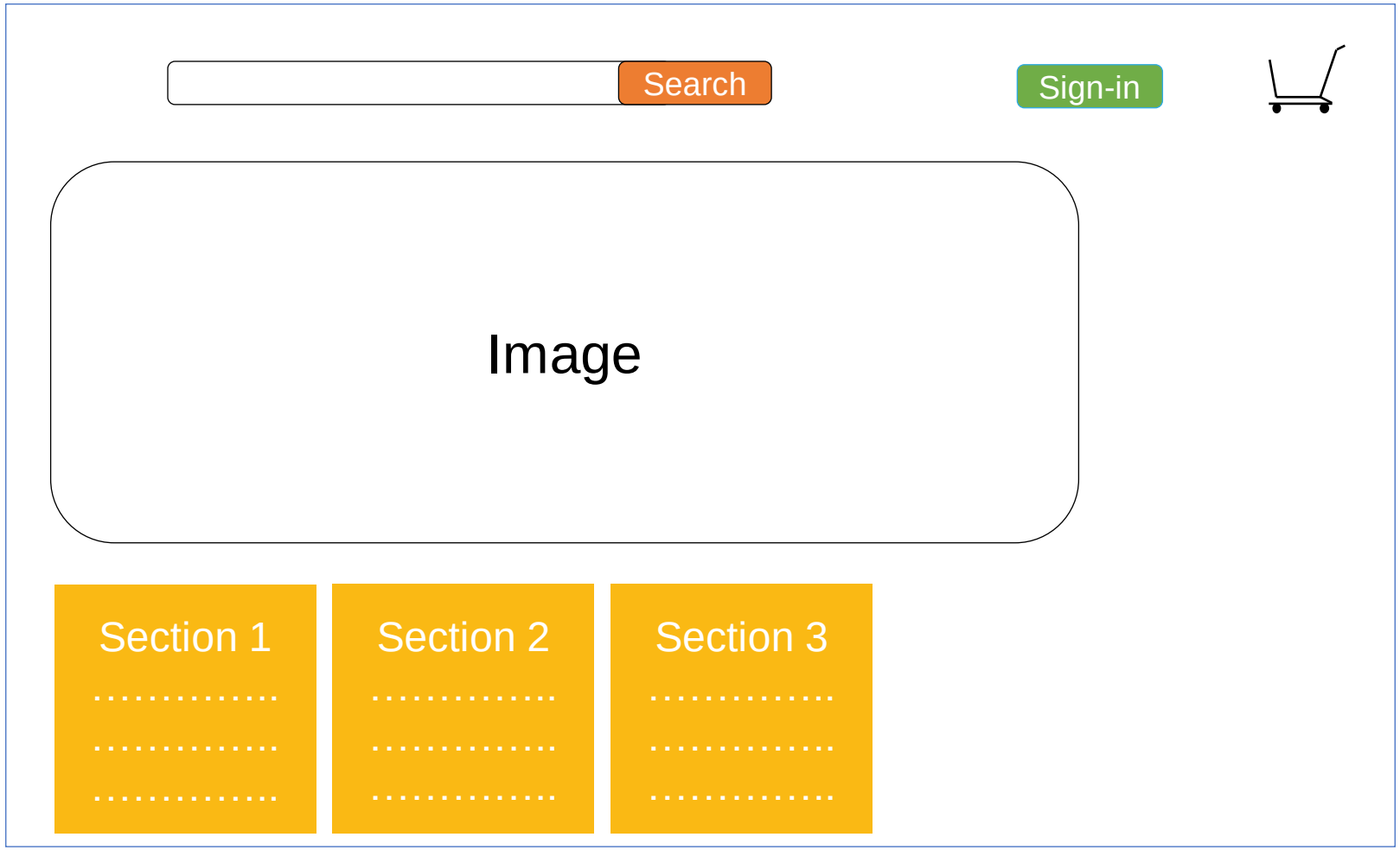
Visual Design Layouts



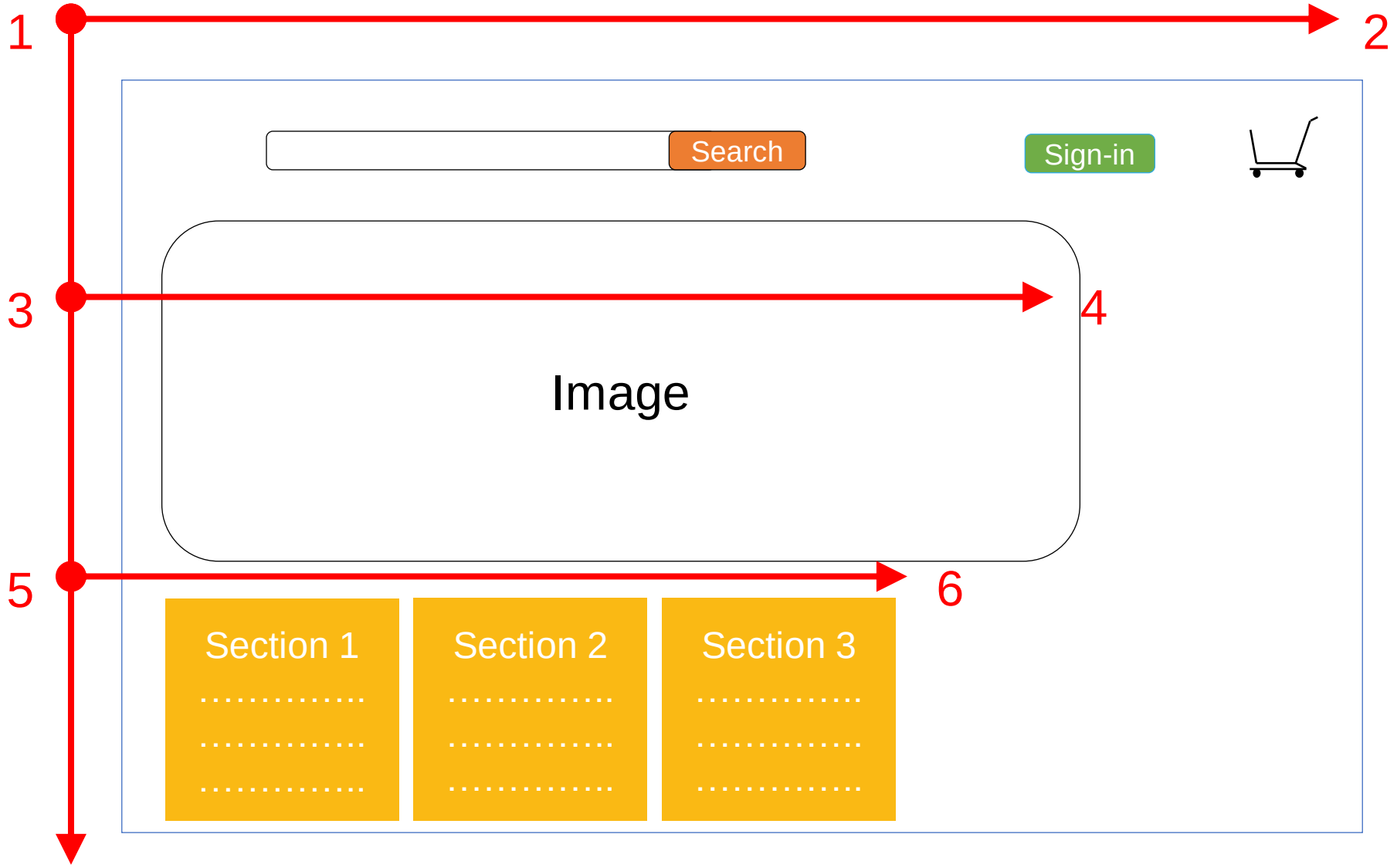
Visual Design Layouts



Visual Design Layouts



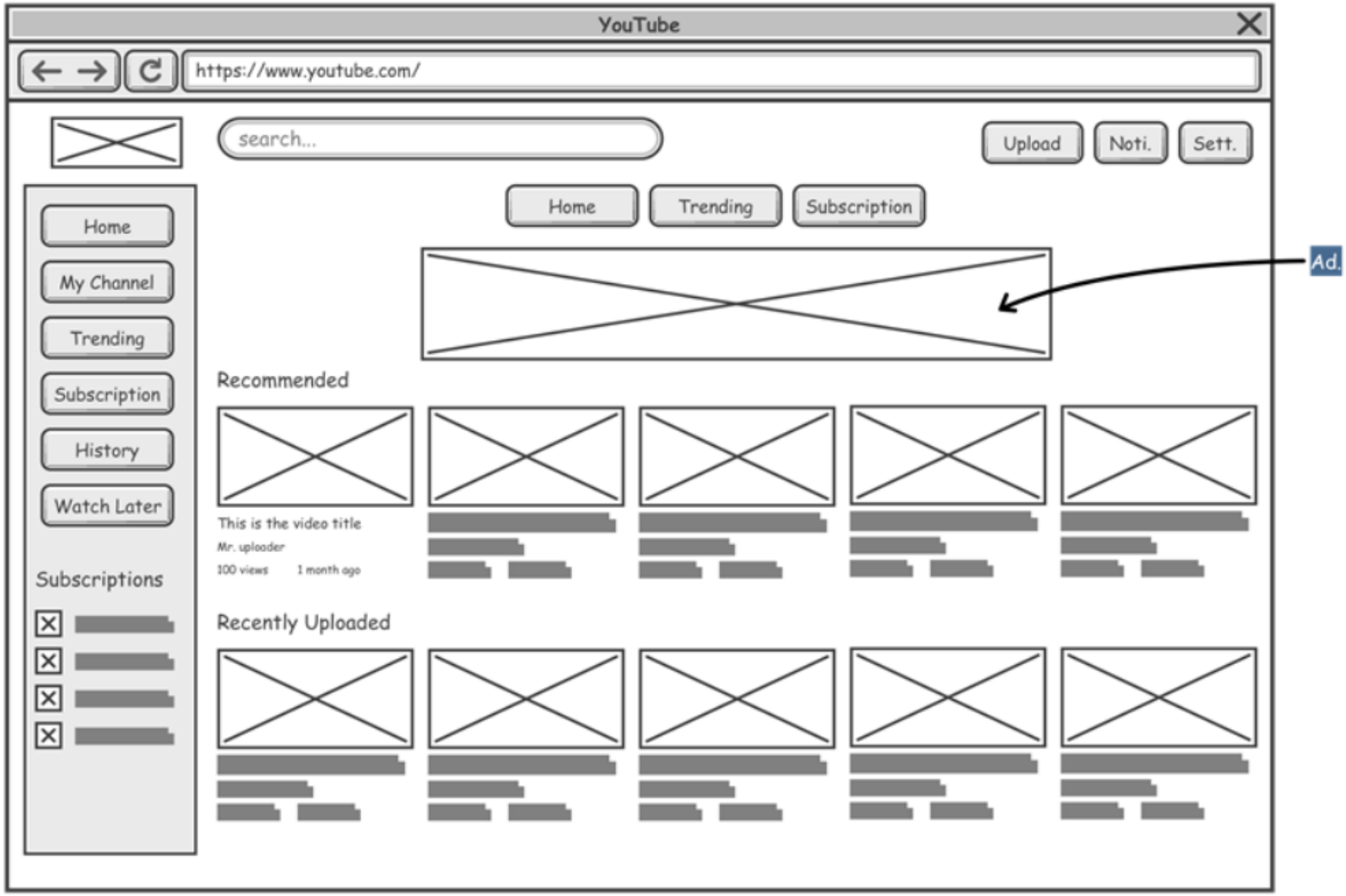
Visual Design Layouts



Website Wireframes

- Wireframes are **skeletal blueprints** of webpages or digital interfaces, illustrating the structure and layout without focusing on visual design elements.
- They serve as a visual guide for organizing content, navigation, and functionality before moving into detailed design and development phases.

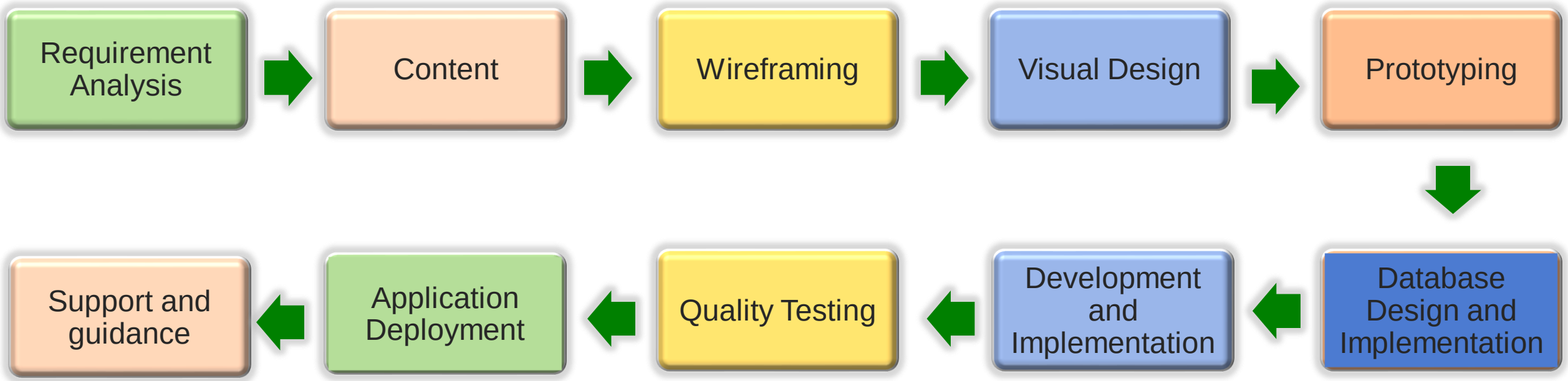
Website Wireframes



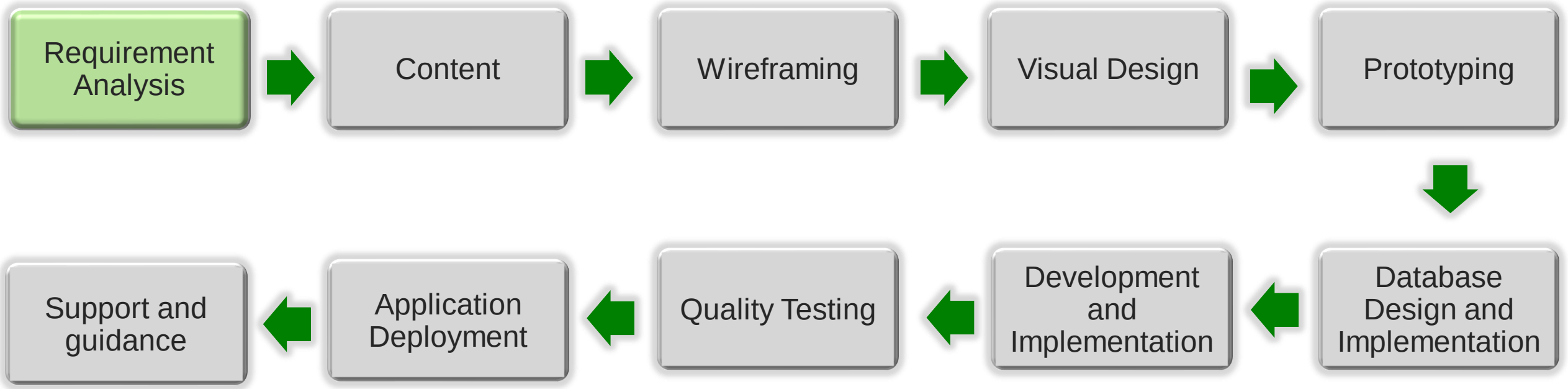
Website Wireframes

Design a wireframe for your website

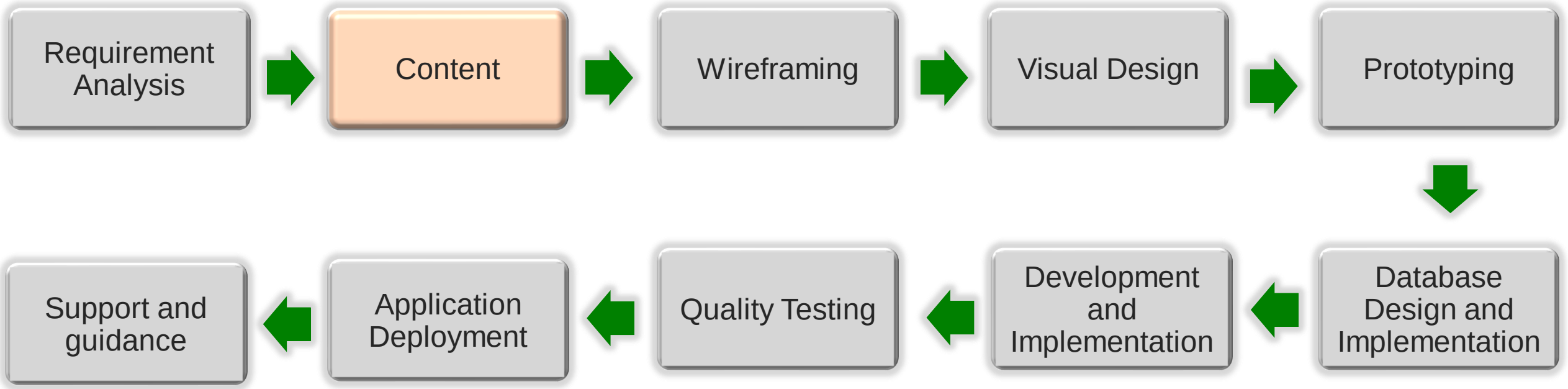
Web Development Workflow



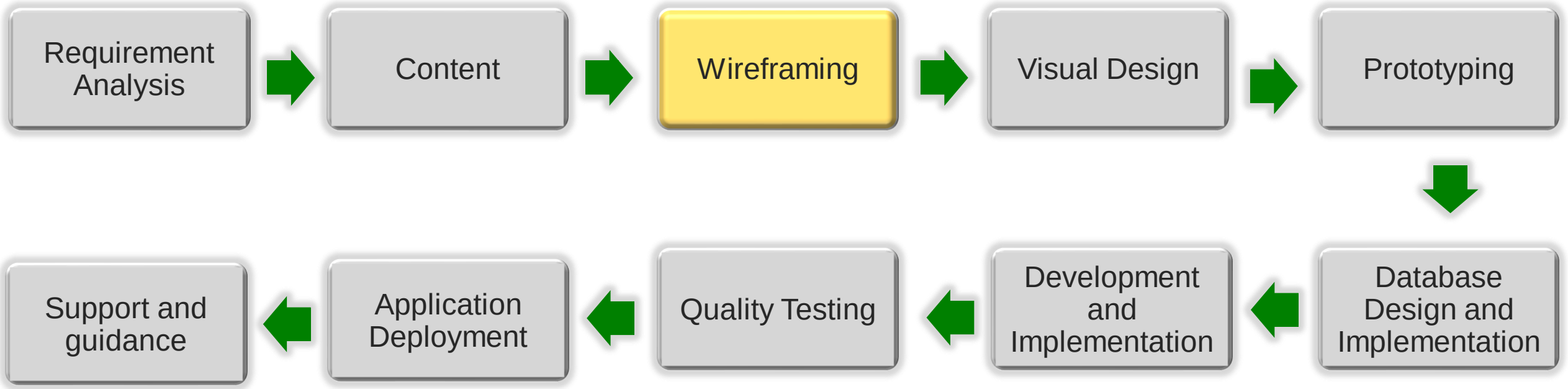
Web Development Workflow



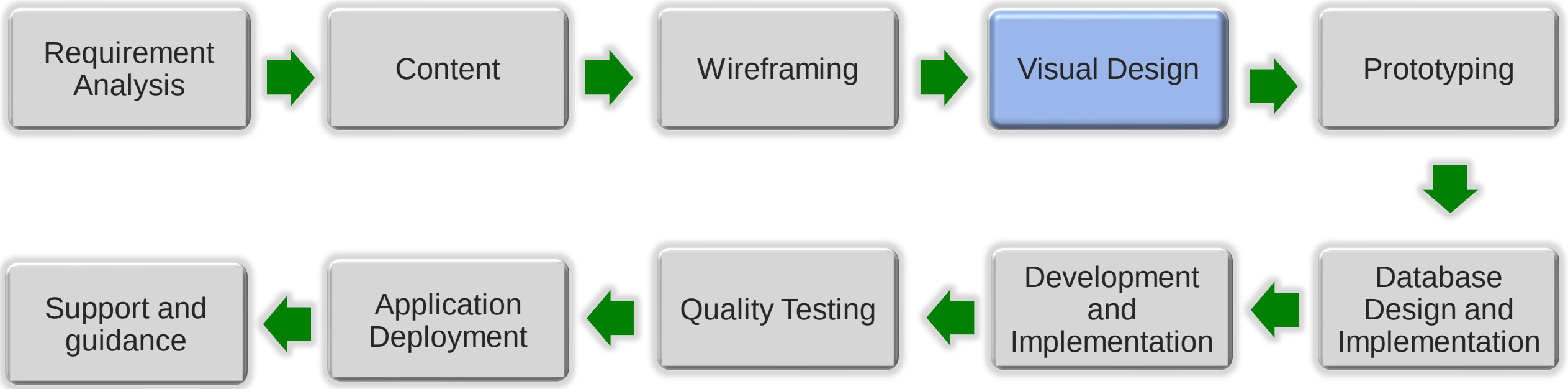
Web Development Workflow



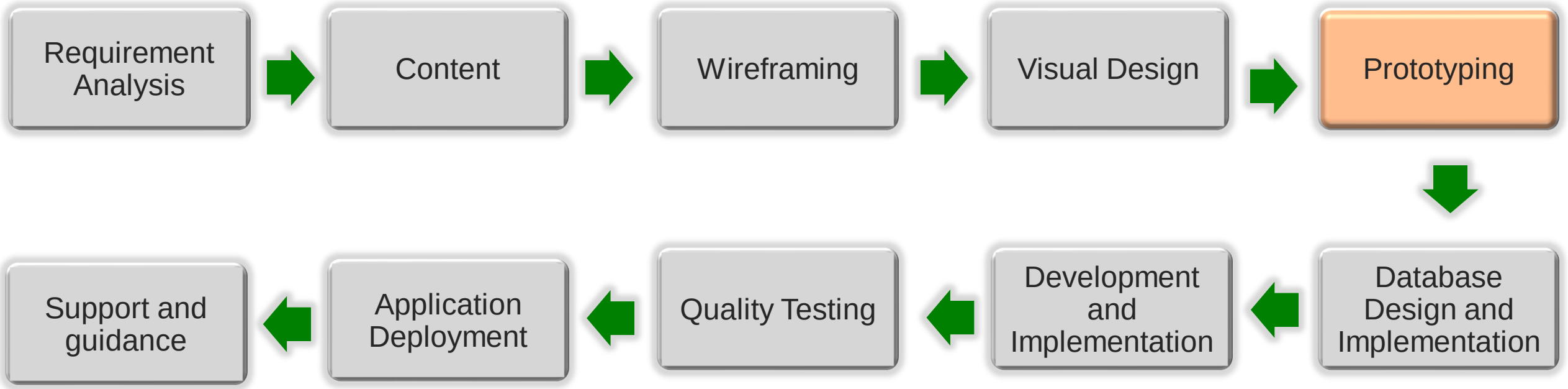
Web Development Workflow



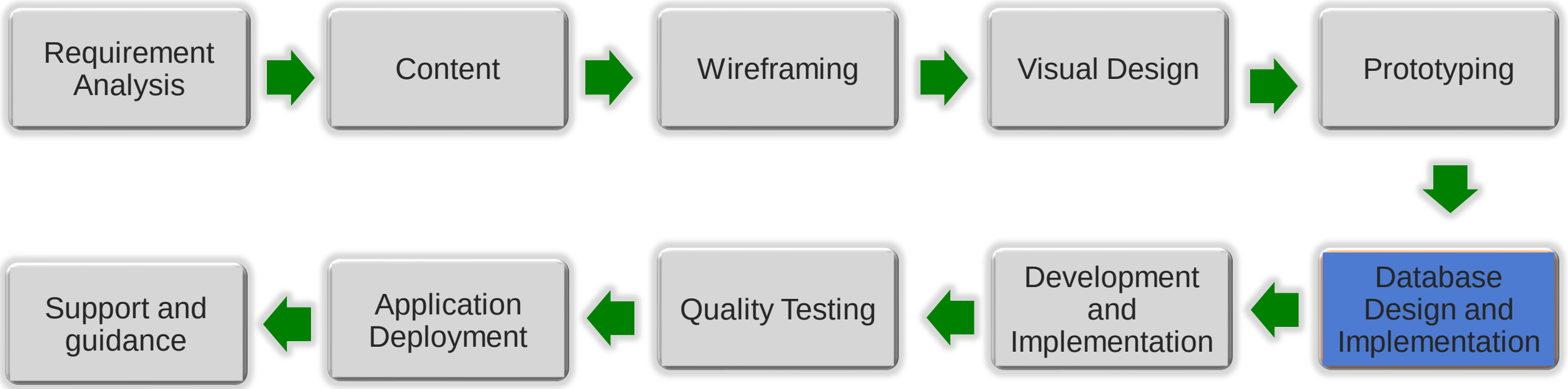
Web Development Workflow



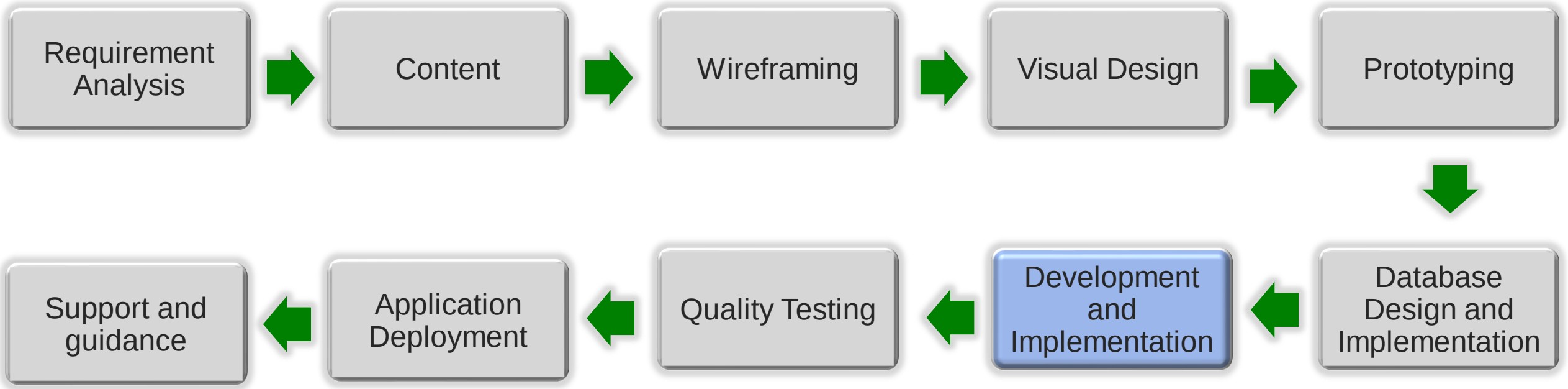
Web Development Workflow



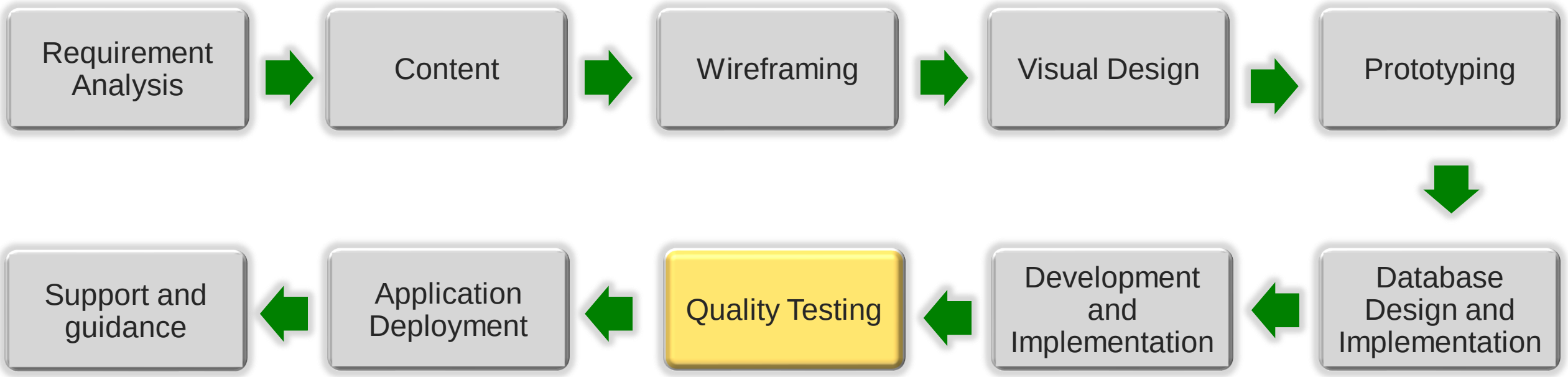
Web Development Workflow



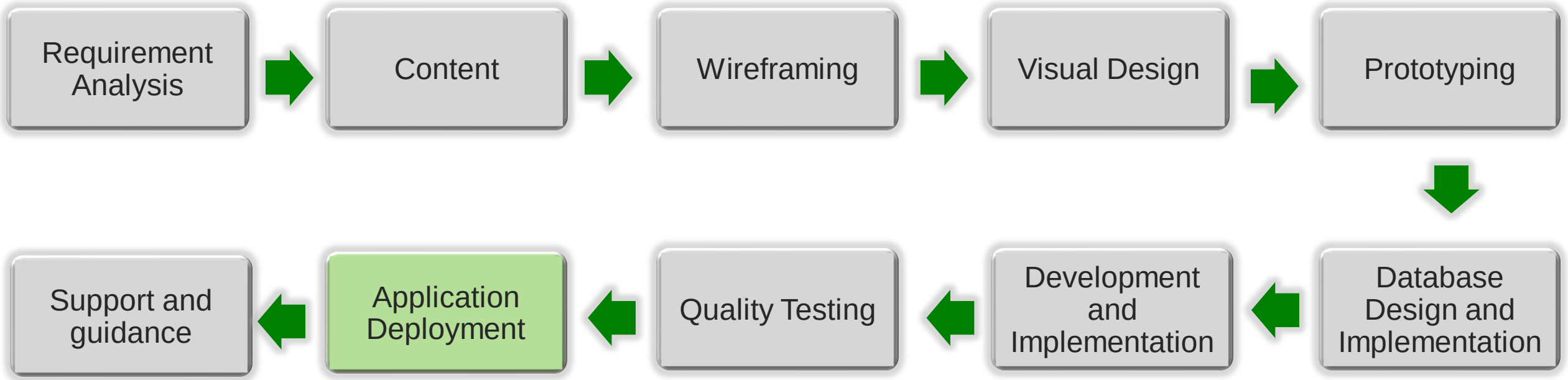
Web Development Workflow



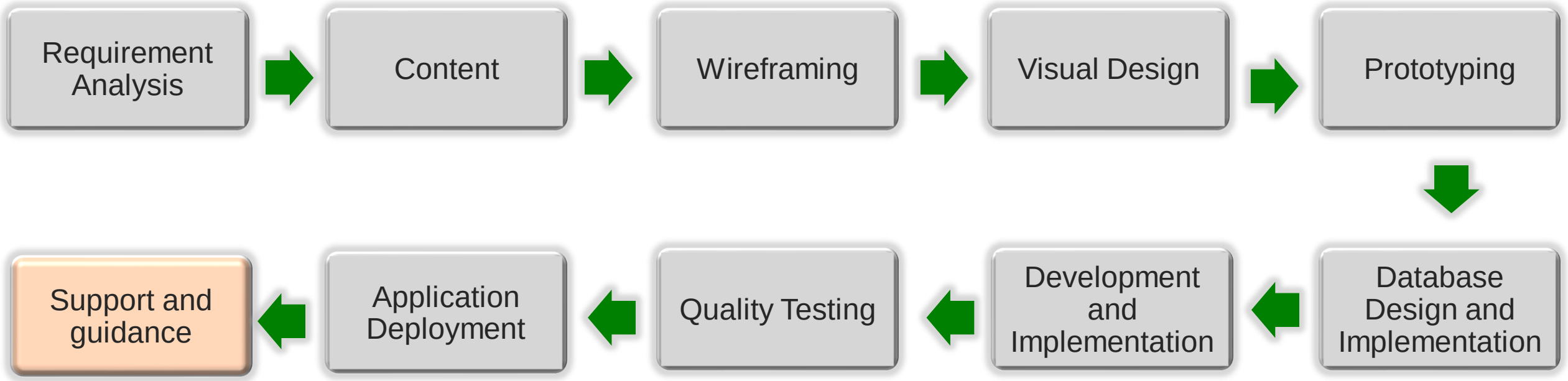
Web Development Workflow



Web Development Workflow



Web Development Workflow



Roles and responsibilities

Here is the roles and responsibilities involved in designing a web site:

Role	Responsibilities
Project Manager	Define project scope, timeline, and budget Coordinate tasks and resources Ensure project stays on track and meets objectives
Web Designer	Create visual elements of the website Design layout and user interface Ensure website is visually appealing and user-friendly
UX/UI Designer	Conduct user research and analysis Design user flows and wireframes Ensure seamless user experience Create intuitive interface designs
Front-end Developer	Write HTML, CSS, and JavaScript code Implement design elements into web pages Ensure cross-browser compatibility and responsiveness

Roles and responsibilities

Role	Responsibilities
Back-end Developer	Develop server-side logic and databases Integrate front-end components with back-end functionality Ensure website functionality and performance
Content Strategist	Develop content strategy and messaging Create or curate content for the website Ensure content aligns with branding and target audience
Quality Assurance	Test website functionality, usability, and compatibility Identify and report bugs or issues Ensure website meets quality standards and specifications
Security Specialist	Implement security measures to protect website data
Marketing Specialist	Develop marketing strategy for website launch Analyze website traffic and user engagement

Web Design And Development (Part 2)

Week 06

CSCI 3230U – Web App Development

Interaction Design

- **Interaction design** focuses on creating meaningful and efficient interactions between users and digital products or services.
- It includes understanding user needs, designing intuitive interfaces, and optimizing user experiences.

Interaction Design Considerations

- User-Centered Design
- Clear and Consistent Interface
- Responsiveness
- Accessibility
- Error Prevention and Recovery
- Scalability and Adaptability

Information Architecture

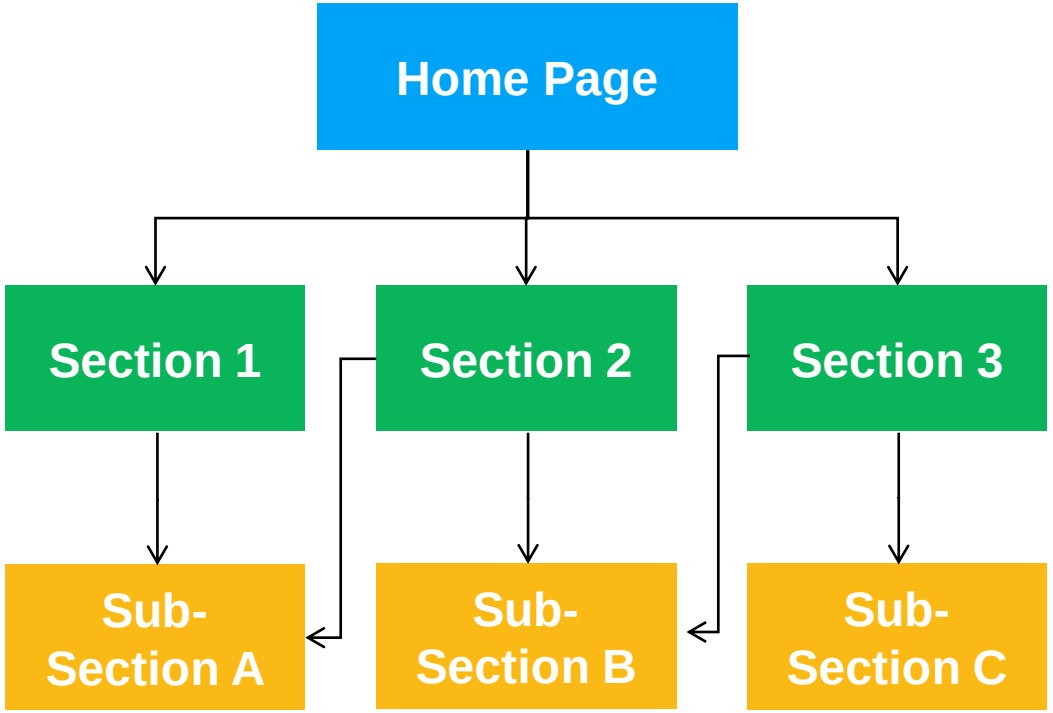
- Searchable
- Well organized
- Easily Navigable
- Clearly labelled

Site Maps

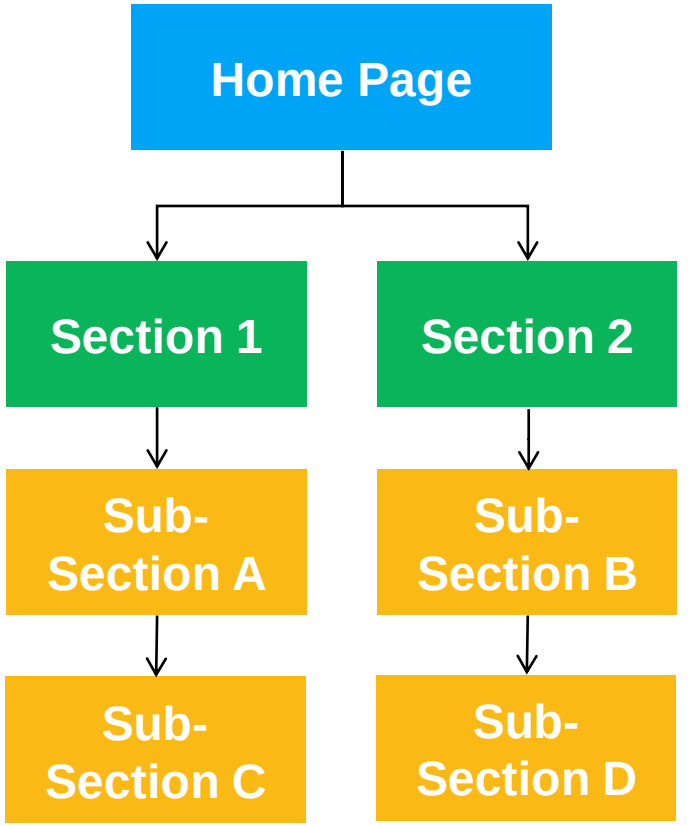
- Site maps are visual representations of a **website's structure**, illustrating the hierarchy of pages and their relationships.
- They serve as **navigational blueprints**, guiding users through digital landscapes and helping them find relevant content efficiently.
- Types of Site Maps:
  - Visual Site Maps
  - XML Sitemaps

Site Maps

- Can be flat or deep in structure.

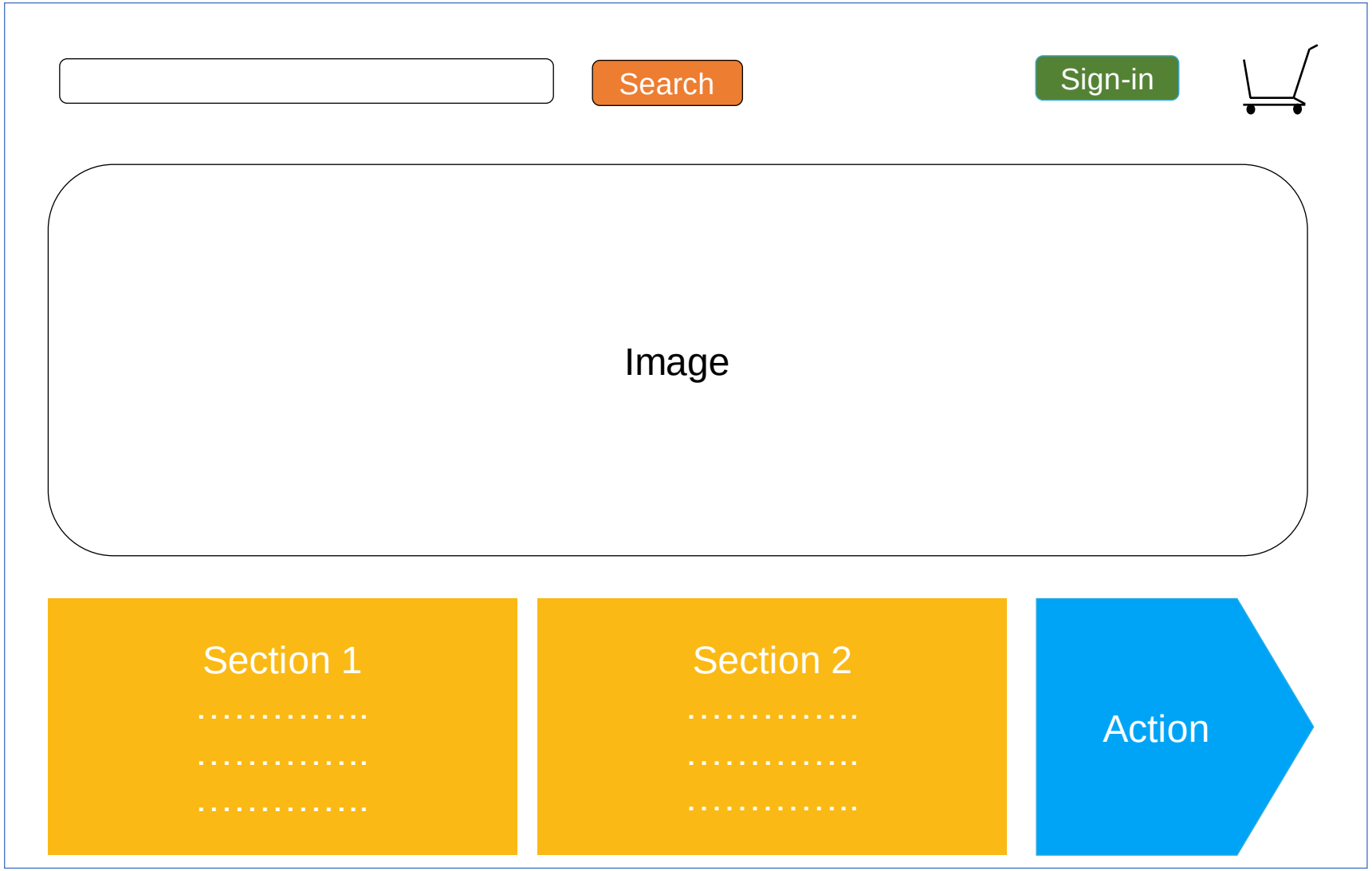


Flat example

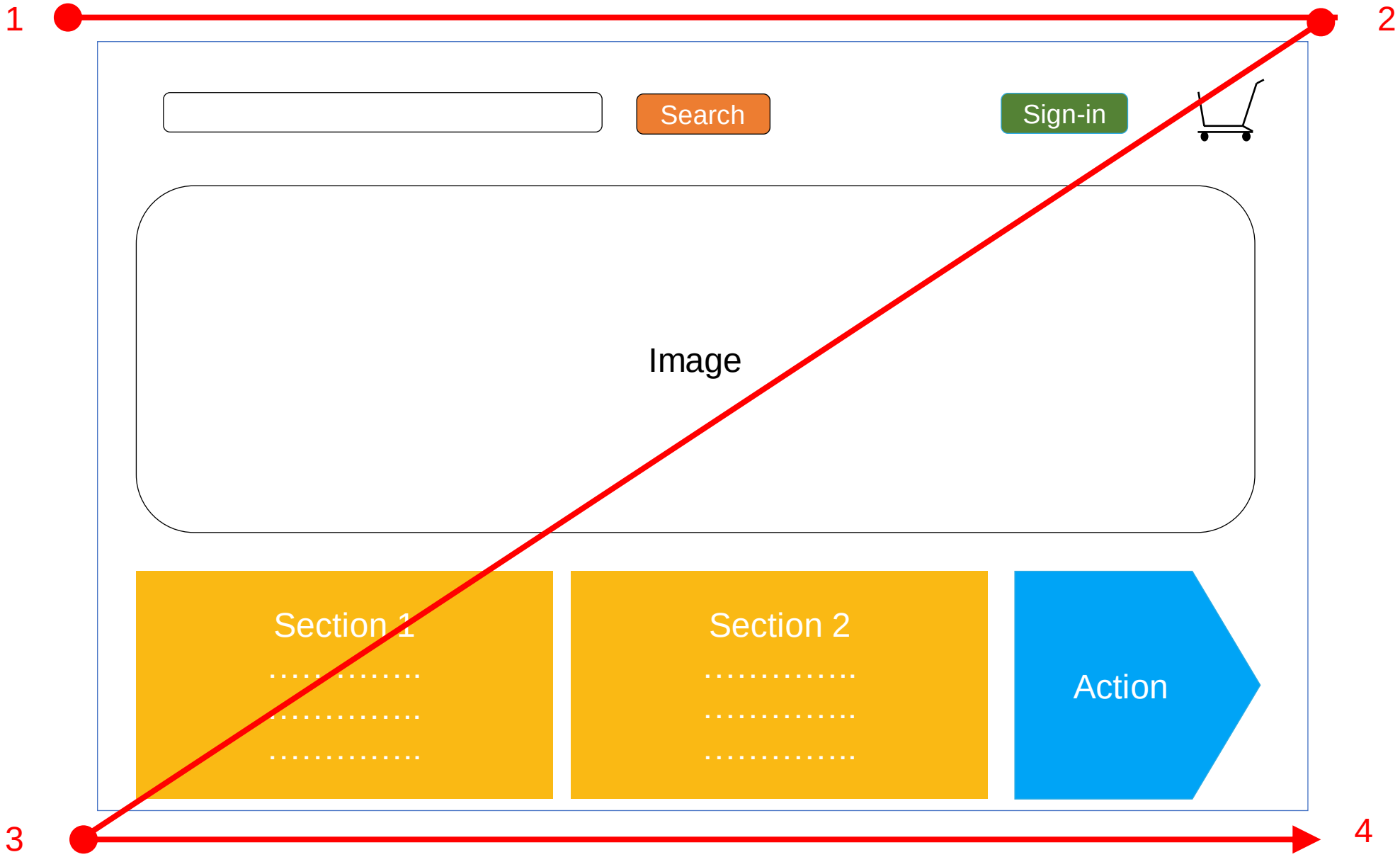


Deep example

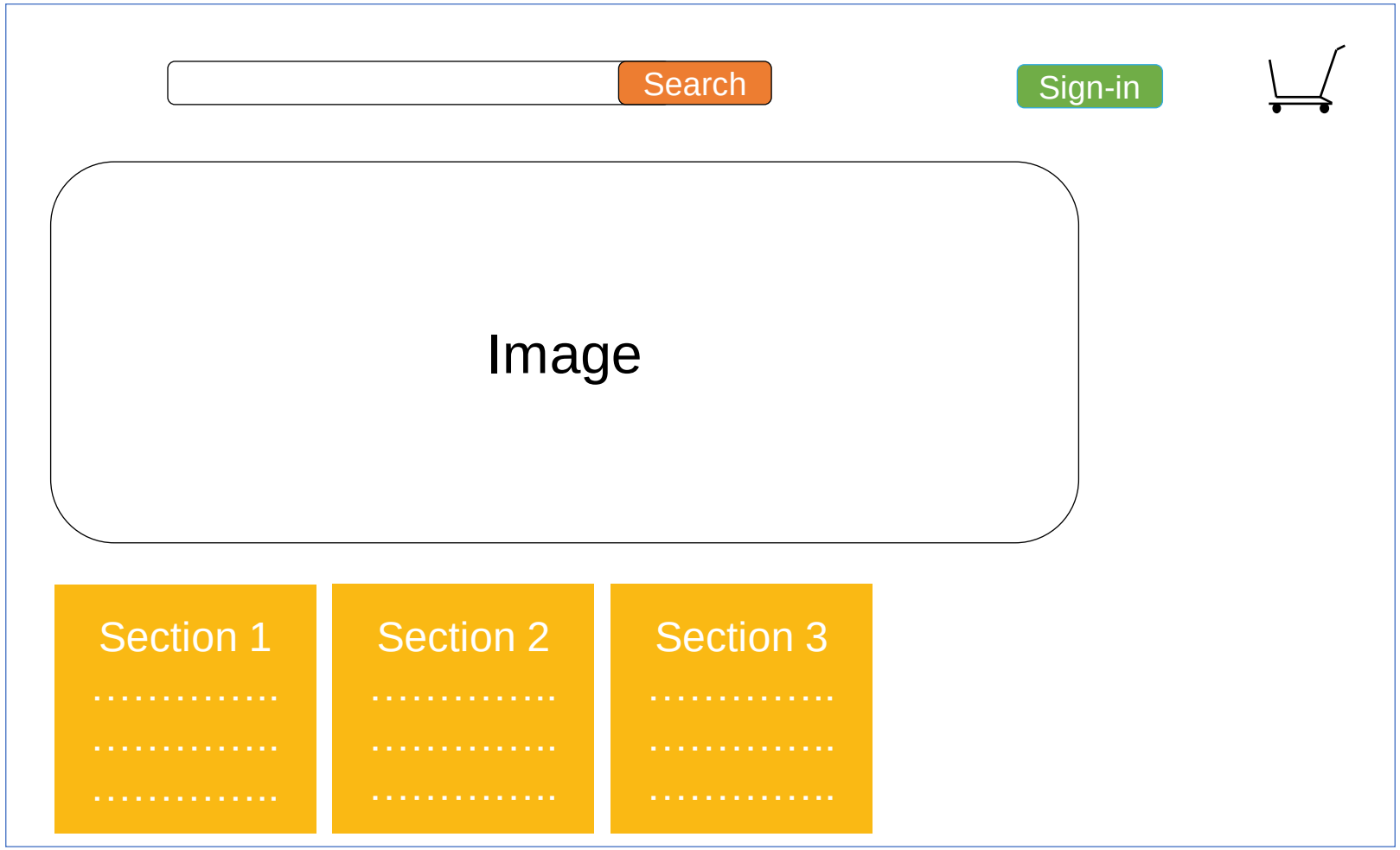
Visual Design Layouts



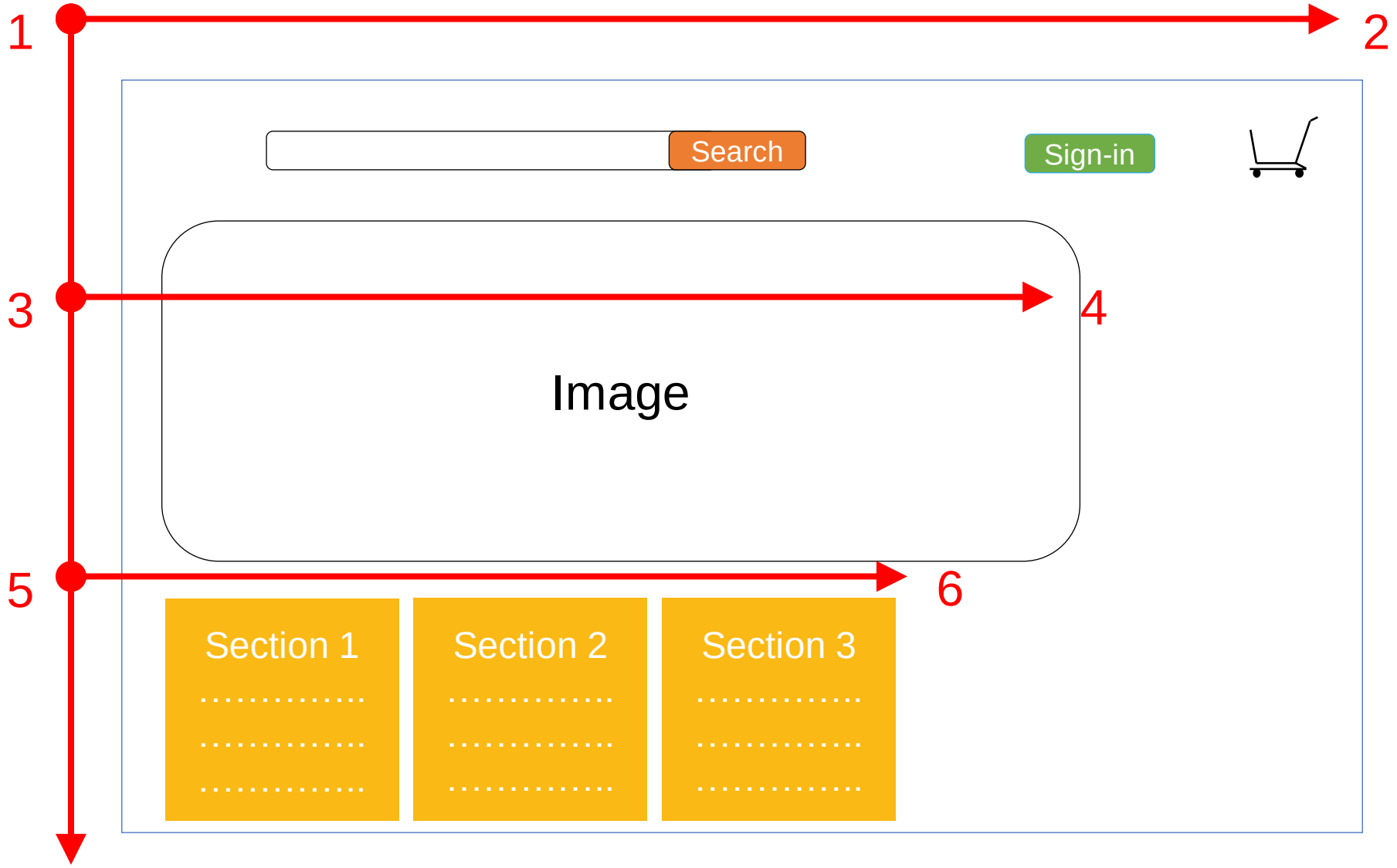
Visual Design Layouts



Visual Design Layouts



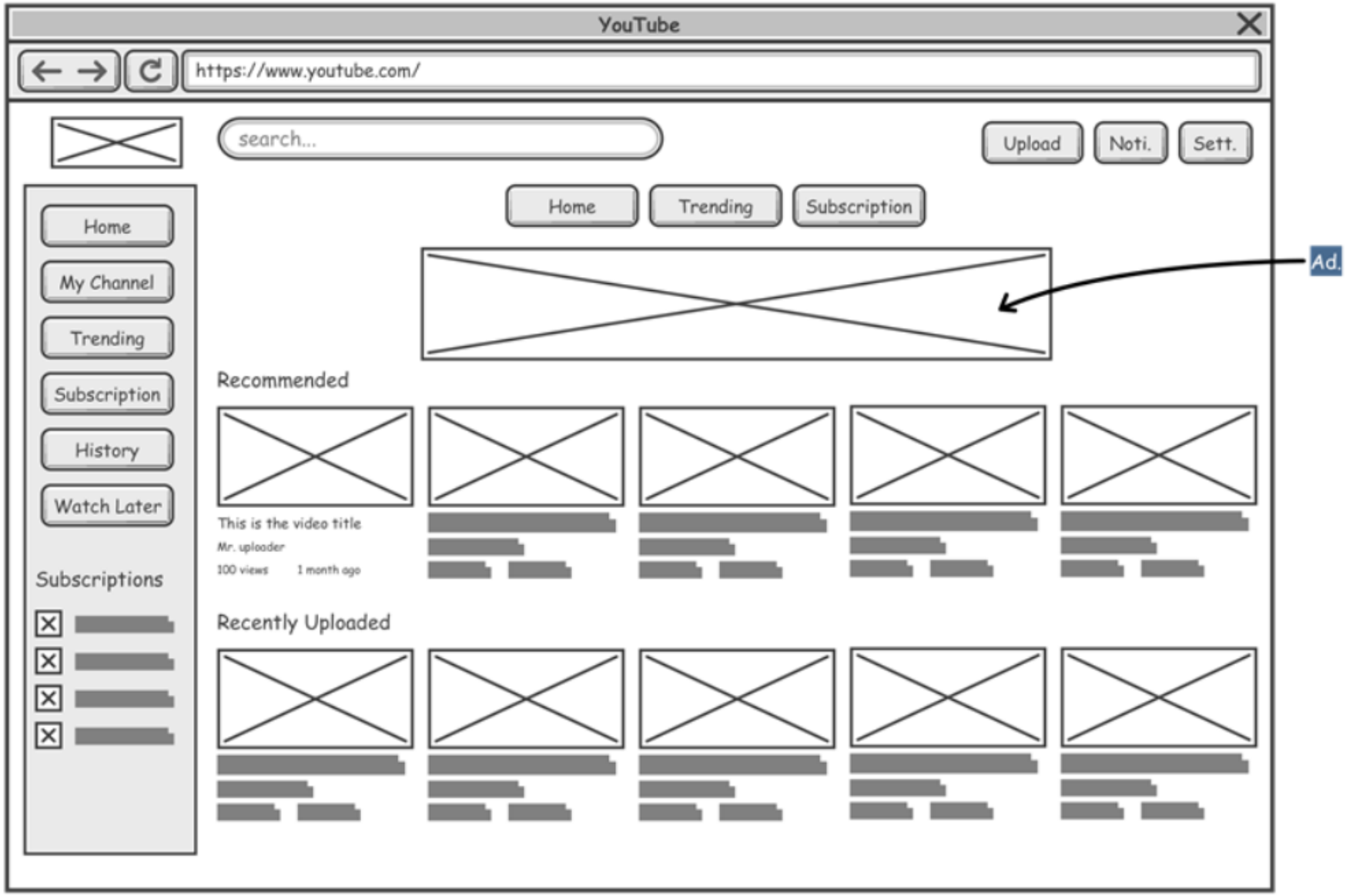
Visual Design Layouts



Website Wireframes

- Wireframes are **skeletal blueprints** of webpages or digital interfaces, illustrating the structure and layout without focusing on visual design elements.
- They serve as a visual guide for organizing content, navigation, and functionality before moving into detailed design and development phases.

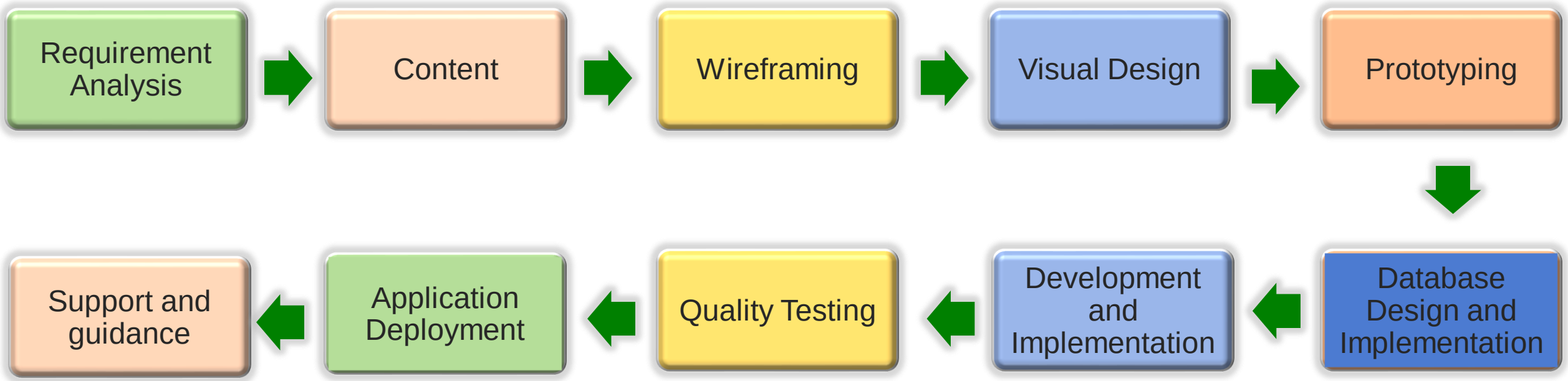
Website Wireframes



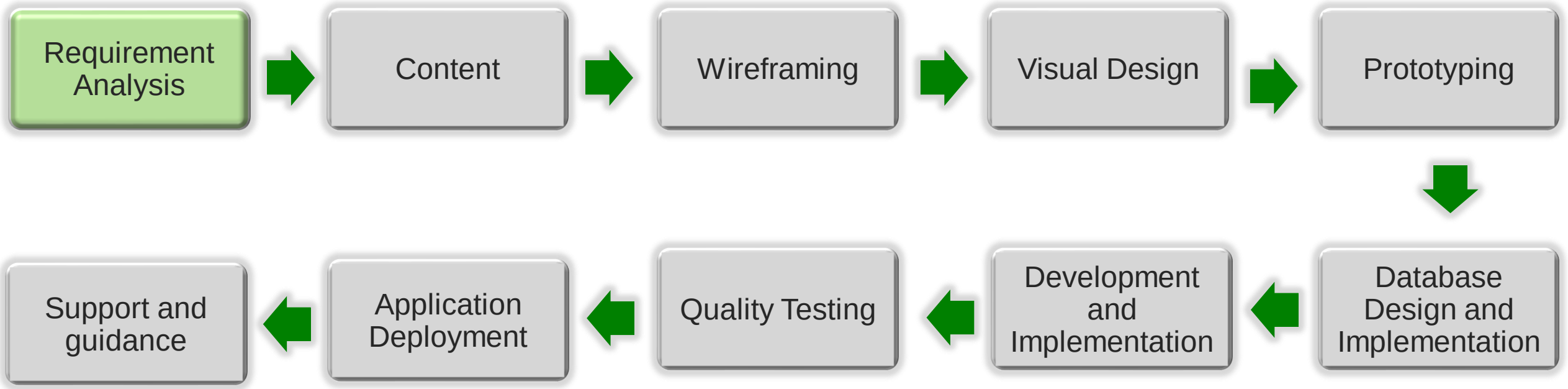
Website Wireframes

Design a wireframe for your website

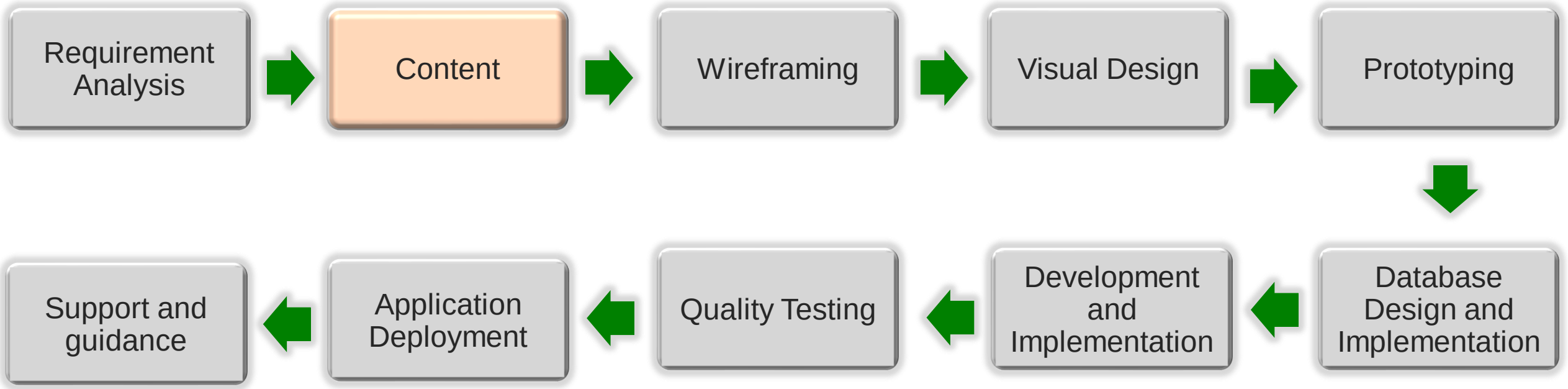
Web Development Workflow



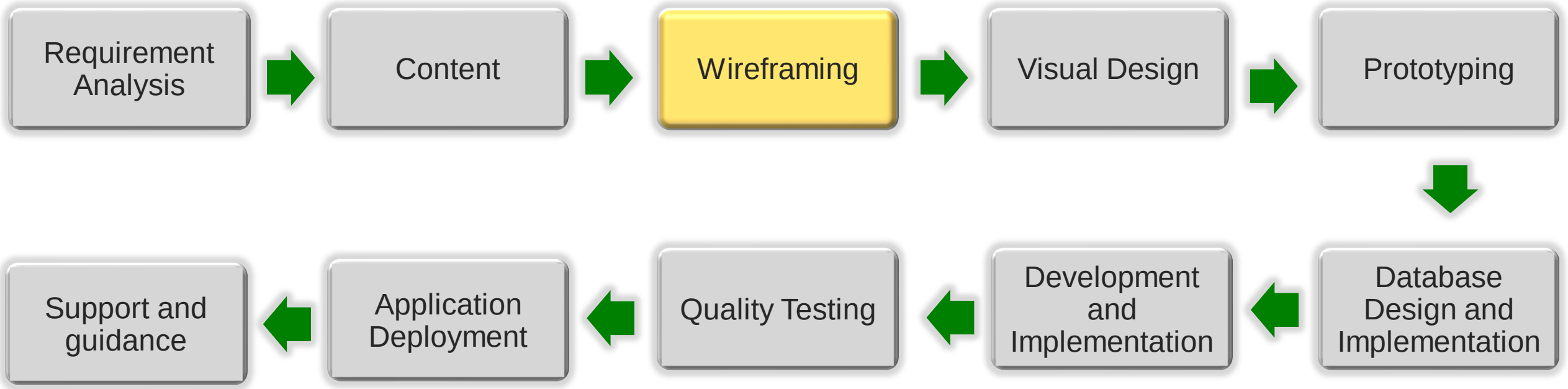
Web Development Workflow



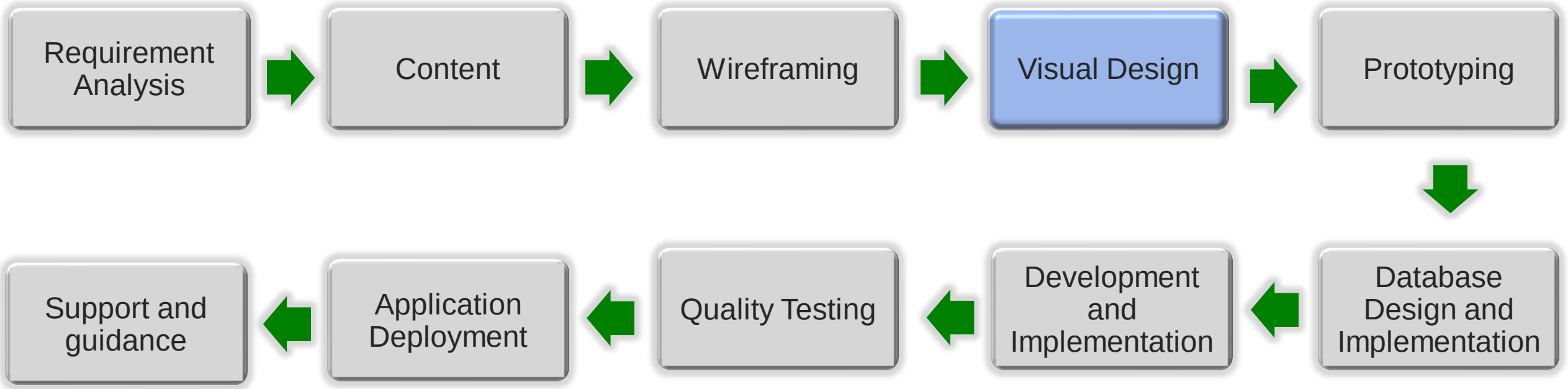
Web Development Workflow



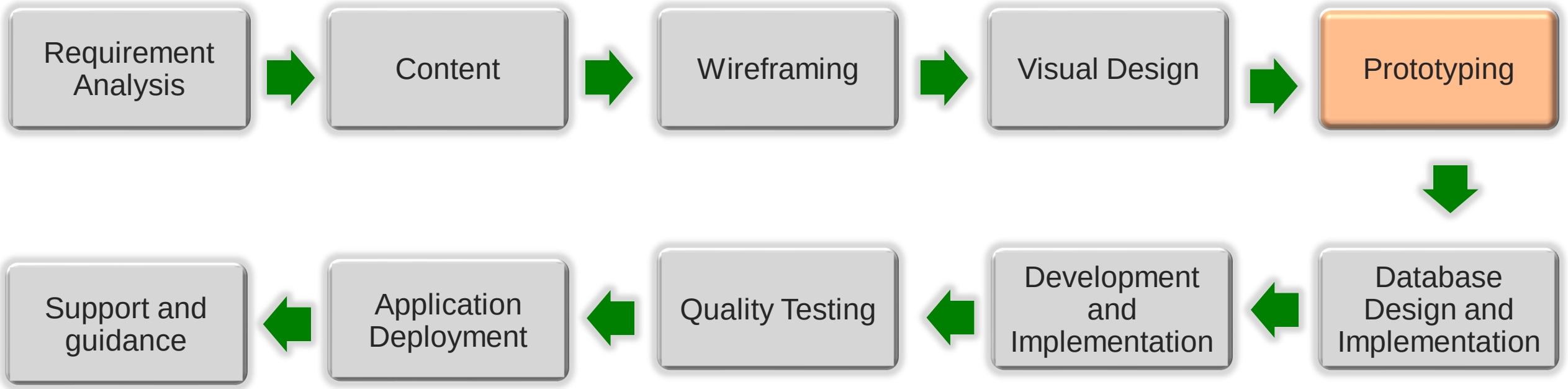
Web Development Workflow



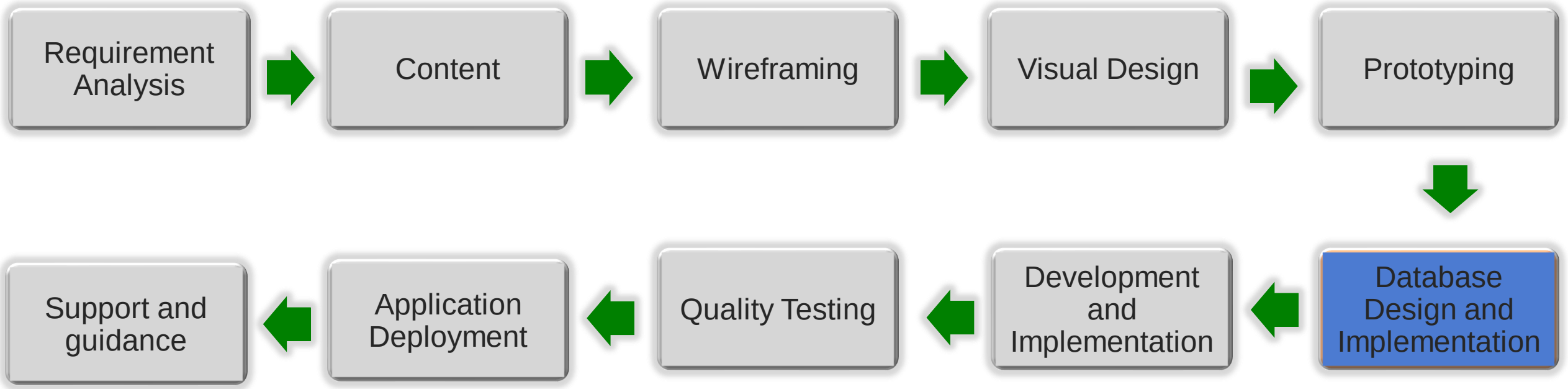
Web Development Workflow



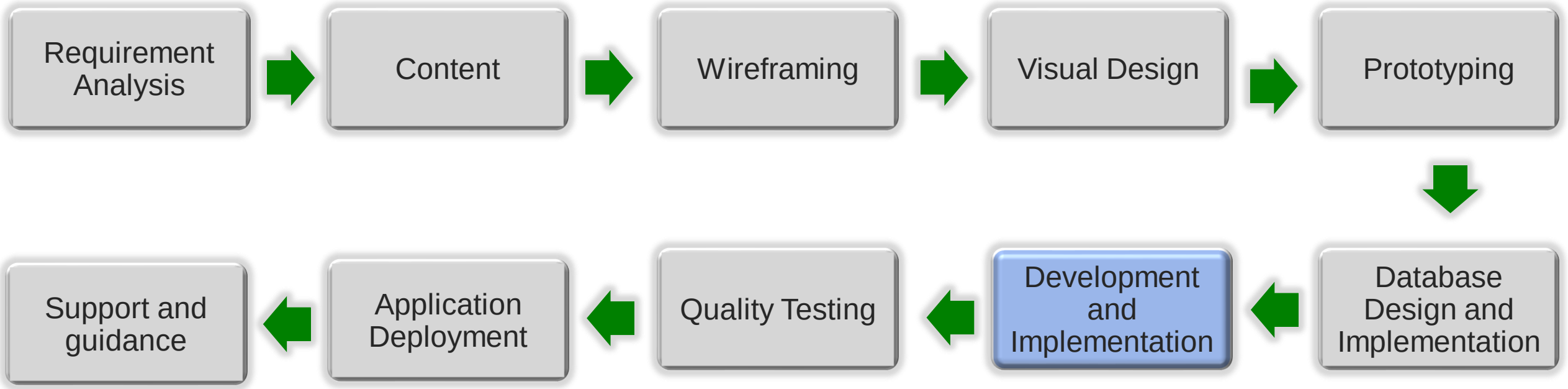
Web Development Workflow



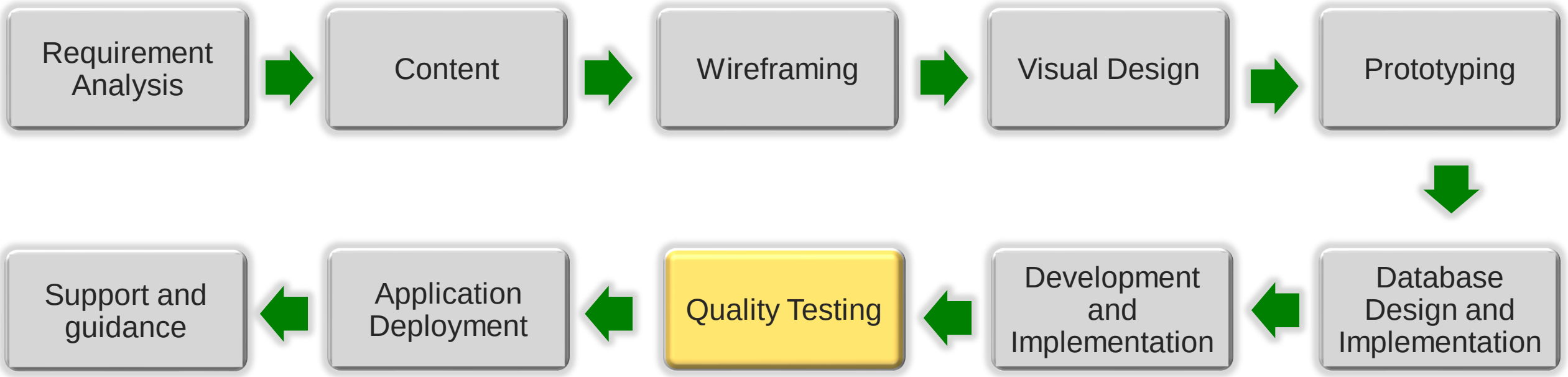
Web Development Workflow



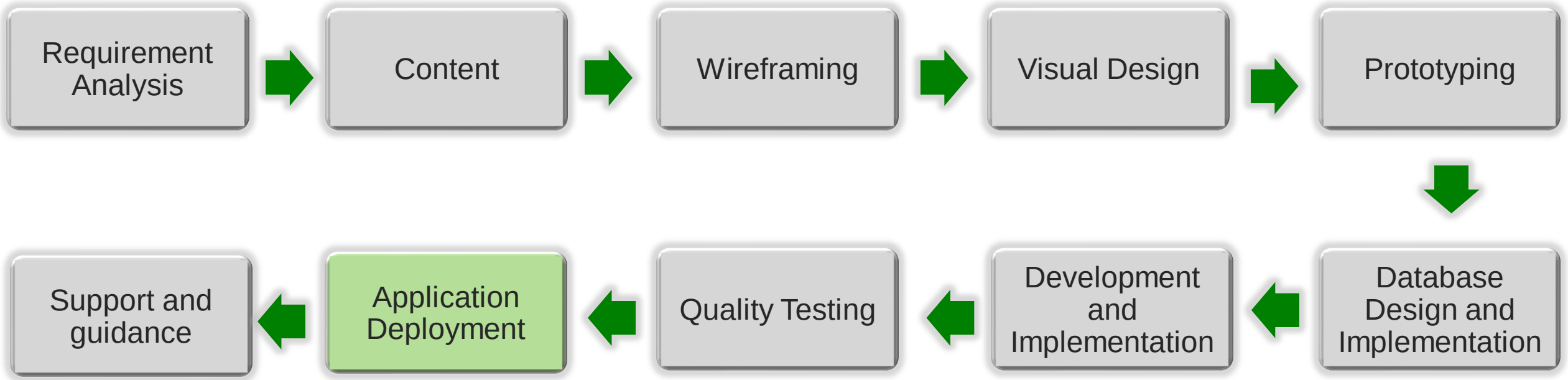
Web Development Workflow



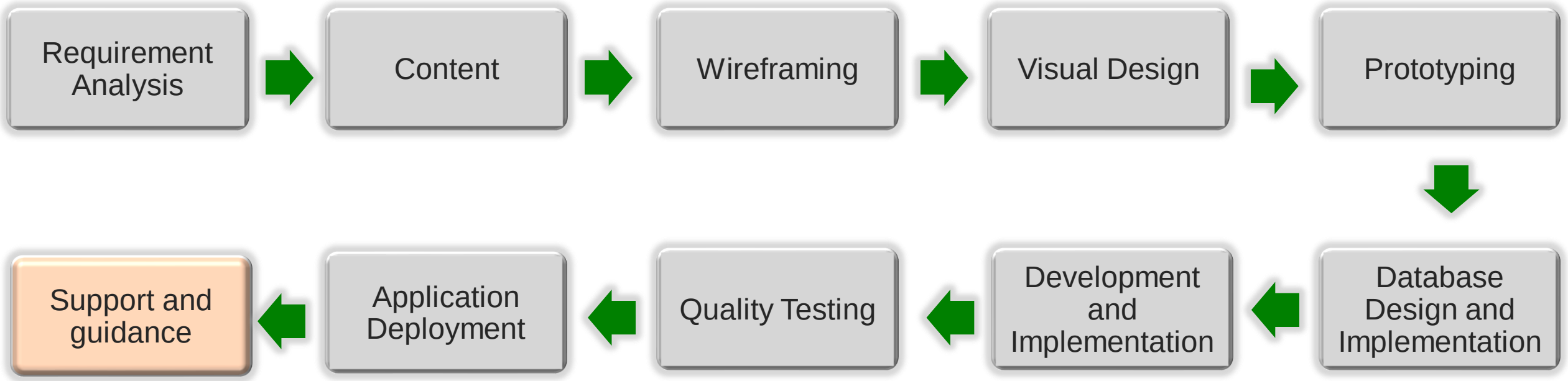
Web Development Workflow



Web Development Workflow



Web Development Workflow



Roles and responsibilities

Here is the roles and responsibilities involved in designing a web site:

Role	Responsibilities
Project Manager	Define project scope, timeline, and budget Coordinate tasks and resources Ensure project stays on track and meets objectives
Web Designer	Create visual elements of the website Design layout and user interface Ensure website is visually appealing and user-friendly
UX/UI Designer	Conduct user research and analysis Design user flows and wireframes Ensure seamless user experience Create intuitive interface designs
Front-end Developer	Write HTML, CSS, and JavaScript code Implement design elements into web pages Ensure cross-browser compatibility and responsiveness

Roles and responsibilities

Role	Responsibilities
Back-end Developer	Develop server-side logic and databases Integrate front-end components with back-end functionality Ensure website functionality and performance
Content Strategist	Develop content strategy and messaging Create or curate content for the website Ensure content aligns with branding and target audience
Quality Assurance	Test website functionality, usability, and compatibility Identify and report bugs or issues Ensure website meets quality standards and specifications
Security Specialist	Implement security measures to protect website data
Marketing Specialist	Develop marketing strategy for website launch Analyze website traffic and user engagement

Web Client

- A web client, also known as a web browser, is a software application **used to access information on the World Wide Web**.
- It **interprets and renders web pages**, allowing users to interact with web content.
- Web clients send requests to web servers to retrieve web pages, files, or data.

Web Server

- A web server is a software application or hardware device that **serves content to web clients over the internet**.
- It **stores**, **processes**, and **delivers** web pages, files, or data in response to requests from web clients.
- Web servers listen for incoming requests from web clients and respond with the requested content.

Type of Resources

- **Static resources**, can be served the same way every time, like images, static web pages, audios, etc.

- **Dynamic resources**, need to be customized for each request, like Google search results, Facebook pages, etc.

Front-end Development

- Front-end development, also known as client-side development, involves creating the **user interface** and **user experience** of a website or web application.
- It focuses on what users see and interact with directly in their web browsers.
- **Technologies** commonly used in front-end development include HTML (Hypertext Markup Language), CSS (Cascading Style Sheets), and JavaScript.
- **Front-end developers** work on aspects such as layout, design, responsiveness, and interactivity to create engaging user experiences.

Back-end Development

- Back-end development, also known as **server-side development**, involves building and maintaining the **server-side logic** and **databases**
- It handles tasks such as **data storage**, **retrieval**, **authentication**, and **server-side processing**.
- **Technologies** commonly used in back-end development include **programming languages** like Java, Python, Ruby, and **frameworks** like Spring Boot, Ruby on Rails.
- Back-end developers focus on creating robust, scalable, and secure server-side systems that support the functionality of web applications.

HTTP

- HTTP stands for Hypertext Transfer Protocol, a protocol used for **transmitting data over the World Wide Web**.
- It is an **application layer protocol** that facilitates communication between clients and servers.
- Each request from a client to a server is independent and unrelated to any previous requests, making HTTP **stateless**. This simplifies communication but requires additional mechanisms like cookies or sessions for maintaining state.
- HTTP messages are **text-based**, consisting of **headers** and an optional **body**. This simplicity allows for easy readability and debugging.

HTTP Methods

- HTTP defines several **methods** or verbs that indicate the desired action to be performed on a resource. Common HTTP methods include:
  - **GET**: Retrieve data from the server.
  - **POST**: Send data to the server to create or update a resource.
  - **PUT**: Update a resource on the server.
  - **DELETE**: Delete a resource on the server.
  - **HEAD**: Retrieve only the headers of a resource without the body.
  - **OPTIONS**: Determine the communication options available for a resource.

Spring Framework (Core)

(Part 1)

Week 07

CSCI 3230U – Web App Development

Dr. Rohollah Moosavi

What is the Spring Framework

- Spring is a **powerful** and **lightweight open-source framework** for building enterprise Java applications.
- It provides **comprehensive infrastructure** support for developing robust and maintainable applications.

Key Feature of Spring Framework

- **Inversion of Control (IoC)**
  - Promotes loose coupling by managing object dependencies.
- **Dependency Injection (DI)**
  - Allows components to be **wired** together effortlessly, enhancing modularity.
- **Container**
  - Contains and manages the lifecycle of objects.
- **Lightweight**

Modules of Spring Framework

- **Core Container**
  - Provides essential functionalities such as IoC and DI.
- **Data Access/Integration**
  - Offers support for database operations and integration with other data sources.
- **Web**
  - Facilitates the development of web applications with features like MVC, REST, and WebSocket.
- **Test**
  - Includes testing utilities for unit and integration testing of Spring components.

Spring Core Container

- Responsible for **creating** and **configuring** the objects (called **Beans/Spring Beans**) that application needs.

- The core of the Spring Framework, responsible for managing the **lifecycle** of Spring beans and their dependencies from creation to destruction of these objects.

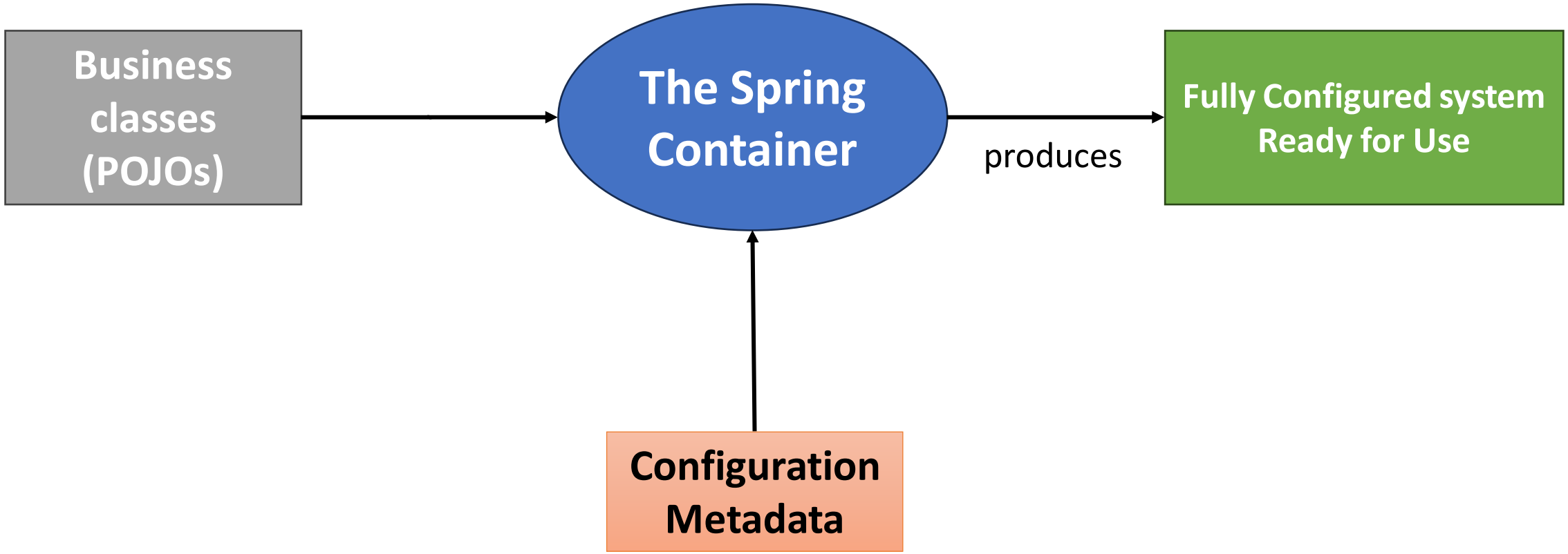
- Utilizes **Dependency Injection** (DI) to wire the objects together.

Spring Core Container

- It can be configured in several different ways:
  - **XML Configuration**: Traditionally configured through XML files defining beans and their dependencies.
  - **Java-based Configuration**: Enables configuration using Java classes with annotations like @Configuration and @Bean.

- They contain bean information and describes how these beans are connected to each other using Dependency Injection

Architecture of Spring Container



Spring Inversion of Control (IOC)

- Spring uses a software engineering principle called **Inversion of Control (IOC)**.
  - **IoC** is a design principle where the control of object creation and lifecycle management is inverted or moved outside of the application code.
- In **traditional programming**, our custom **controls** the flow of the application; Our code calls a library to complete tasks and controls and the Library always returns control to our code.
- However in **Inversion of Control**, we give control to a framework; We plug our custom code into the framework and the framework will call our code when required.

Spring Inversion of Control (IOC)

- A core aspect of IoC is **Dependency Injection (DI)**, where dependencies are injected into objects by an external entity.

- **Dependency Injection** lets Spring take control of the weaving process of the program by automatically injecting objects (Beans) where needed.

- We can interact with the **Spring IoC container** in our code through the **ApplicationContext** interface.

Configuration

- To use Spring Container, need to add **Spring-Context** dependency in pom.xml file.
  - https://mvnrepository.com/artifact/org.springframework/spring-context

- We use Java class for configuration, including Beans, Injections and Advanced Configuration.

- To declare a **Java Class Spring Configuration** file, we add @**Configuration** above the class in our application.

Application Context

- We can interact with the Spring Container through one of the **ApplicationContext implementations**:
  - **AnnotationConfigApplicationContext** *(we use this one)*
  - AnnotationConfigWebApplicationContext
  - ClassPathXmlApplicationContext
  - FileSystemXmlApplicationContext
  - XmlWebApplicationContext
- Use the following line to instantiate all the beans required to run the application.

```
AnnotationConfigApplicationContext context =
        new AnnotationConfigApplicationContext(Config.class);
```

- And to close all auto closable beans in the context, use the following one:

```
context.close();
```

Bean Creation

- To define a bean via the configuration class, we can add the @Bean annotation above a method that returns the type we wish to create as a bean:

```
@Bean
public EntityManagerFactory entityManagerFactory() {
     return Persistence.createEntityManagerFactory("myPersistence.xml");
}

@Bean
public EntityManager entityManger(EntityManagerFactory emf) {
     return emf.createEntityManager();
}
```

- The objects (beans) that we want Spring to manage them, need to be defined in the configuration.

Components and Autowiring

- It is not a best practice to specify Bean methods, like the previous slides, for all the dependences.

- Another approach is using **Stereotype Annotation** like **@Component** on our classes to specify for Spring which classes will be Bean to manage them.

- In this approach, Spring instantiates Beans and can populates fields, arguments, dependencies (***wiring the application together***) that have **@Autowired** annotation there.

- So instead of using @Bean method in a config class, we need to add **@Component** annotation to the classes that we want a Bean. Also annotate the attributes or methods with **@Autowired** to know the beans should be injected into those methods, constructors or attributes.

Components and Autowiring

- We can **mark a class** as a component by using @Component, which will be picked up by Spring as a class out of which to create a Bean and **wire** as a dependency where needed.

```
@Component
public class ExampleService {
    private ExampleRepository repository;

    @Autowired
    public ExampleService(ExampleRepository repository) {
        this.repository = repository;
    }

    // Other methods
}
```

- The above code uses the constructor to **wire** the correct dependency during instantiation.
- This means the `ExampleService` class **depends** on an instance of `ExampleRepository`.
- By annotating the constructor, Spring can recognize this dependency and automatically wire the correct bean dependency here (called **constructor injection**)

Spring Core Framework

Spring Core Demo 1

Spring Framework (Core)

(Part 2)

Week 07

CSCI 3230U – Web App Development

Dr. Rohollah Moosavi

Lifecycle of a bean

- As mentioned, **Spring Container** keep track and maintain beans of our application.
- The **Spring Framework** abstracts away much of the complex communication between the Spring Container and the Spring Beans. Thus, our primary focus remains on accurately defining our code, while Spring handles the underlying complexities seamlessly.
- There are five main stages of the lifecycle: **creation**, **injection**, **validation**, **registration**, **destruction**.

Beans and Dependencies

- There are three different types of **injection**:

  1. Through a constructor (constructor injection)

     - We used it earlier by using @Autowired annotation and it is good for mandatory dependencies.

```
@Component
public class OrderService {
    private final UserRepository userRepository;

    @Autowired
    public OrderService(UserRepository userRepository) {
        this.userRepository = userRepository;
    }
    // Class implementation
}
```

Beans and Dependencies

- There are three different types of **injection**:

  2. Through setters (setter injection)

     - Good for optional dependencies

```
@Component
public class ProductService {
    private ProductRepository productRepository;

    @Autowired
    public void setProductRepository(ProductRepository productRepository) {
        this.productRepository = productRepository;
    }
    // Class implementation
}
```

Beans and Dependencies

- There are three different types of **injection**:

  3. Through reflection (field injection)

     - Has many drawbacks and should be avoided.

```
@Component
public class ShoppingCartService {
    @Autowired
    private ProductRepository productRepository;

    // Class implementation
}
```

Repository Annotation

- **@Component** is a very general type of annotation that can be used on any type of class.
- We can use **@Repository**, to mark a class as a **repository**.
  - Used to annotate DAO (Data Access Object) components.
  - Encapsulates data access logic, such as database operations.
  - Enables Spring to handle exceptions and transactions.

```
@Repository
public class EmployeeRepository {

    private Map<Integer, Employee> repository;

    public void store(Employee emp) {
        repository.put(emp.getEmployeeId(), emp);
    }

}
```

Controller Annotation

- We can use **@Controller**, to mark a class as a **controller**.
  - Used to annotate controller components in MVC architecture.
  - Handles HTTP requests and delegates to appropriate service methods.
  - Facilitates separation of concerns in web applications.

```
@Controller
public class UserController {
    @Autowired
    private UserService userService;

    // Request mapping methods
}
```

Service Annotation

- We can use **@Service**, to mark a class as **service**.
  - Used to annotate service layer components.
  - Typically contains business logic.
  - Automatically detected during component scanning.

```
@Service
public class EmailService {

    public boolean validateEmail(String emailAddress) {
        boolean isValid = false;
        isValid = emailAddressValidator.validateAddress(emailAddress);
        //additional validation code
        return isValid;
    }
}
```

Bean Scopes

- There are two possible Bean Scopes for Spring Beans:
  - **Singleton:** Only one instance of the Bean Class will be created and used throughout the life of the application (it's the default Bean Scope)

```
@Bean
@Scope("singleton")
public Person personSingleton() {
    return new Person();
}
```

  - **Prototype:** A new instance of the Bean Class will be created and used every time one is requested.

```
@Bean
@Scope("prototype")
public Person personPrototype() {
    return new Person();
}
```

Spring Core Framework

Spring Core Demo 2

Spring Framework (Spring Boot)

Week 08

CSCI 3230U – Web App Development

Dr. Rohollah Moosavi

Spring Problems

- Configuring Spring applications can be time-consuming and takes time from writing application logic.
- Managing project dependencies can be frustrating due to Spring dependencies are very version sensitive.
- Redundancy in configuration and dependency management across projects adds to the complexity, with similar tasks often needing to be repeated from one project to another.

Spring Boot

- In Spring Boot, we have:
  - Automatic common configuration.
  - Starter dependencies.
  - Actuator-additional feature to help monitor and manage a spring boot application
  - …
- Spring Boot is ideal for building **microservices**, **web applications**, and **RESTful APIs** with minimal setup and configuration overhead.

Spring Boot Setup

- The **Spring Initializr** is a tool that can generate Spring Boot project structures.
- Spring Initializr can be used:
  - Through Spring Tool Suite
  - Through a web-based interface
  - Using the Spring Boot CLI
  - …

Spring Boot Setup

Through Spring Tool Suite

Create a Project to manage Students list using start.spring.io with the following dependencies: web, Thymeleaf, JPA, H2 and DevTools

Spring Boot Setup

Through a web-based interface

Create a Project to manage Students list using start.spring.io with the following dependencies: web, Thymeleaf, JPA, H2 and DevTools

Spring Boot Setup

Generated Java Classes

- Once the project is created, 3 components are generated automatically:
  - `SpringBootStudentDemoApplication.java`
    - The application's bootstrap class and primary Spring configuration class.
  - `SpringBootStudentDemoApplicationTests.java`
    - A basic integration test class.
  - `application.properties`
    - A place to configure application and Spring Boot properties.

Spring Boot Setup

Application entry point

- If you browse the files, you will see a pre-generated Java class with a main method:

```
@SpringBootApplication
public class SpringBootStudentDemoApplication {

    public static void main(String[] args) {
        SpringApplication.run(SpringBootStudentDemoApplication.class, args);
    }

}
```

- This class will be used to run our application, even web based.

Spring Boot Setup

Application entry point

- This class has two purposes:
  - **Configuration**: It functions as the primary Spring configuration class.
  - **Bootstrapping**: It activates the auto-configuration feature in a Spring Boot application
- The Spring Boot auto-configuration process will handle a significant portion of the necessary configuration.
- However, you'll still require some minimal Spring configuration to facilitate auto-configuration.

Spring Boot Setup

Application entry point

- The **@SpringBootApplication** enables Spring **component-scanning** and Spring Boot **auto-configuration**.
- This annotation combines three essential annotations:
  - **@Configuration**: Sets a class as a java configuration class.
  - **@ComponentScan**: Enables component-scanning, allows the spring framework to automatically read annotations in other classes such as controllers.
  - **@EnableAutoConfiguration**: Enables Spring Boot auto-configuration avoiding to manually write required configuration.

Spring Boot Setup

Application configuration

- Navigate to `src\main\resources` and open the application.properties file, which is currently empty. This is where we can configure the application and override any default values that are not suitable.

- You can enter some basic configuration for our web server and database like:

```
server.port=8081
spring.jpa.hibernate.ddl-auto=create

spring.h2.console.enabled=true
spring.datasource.url=jdbc:h2:mem:testdb
spring.datasource.driverClassName=org.h2.Driver
spring.datasource.username=sa
spring.datasource.password=password
spring.jpa.database-platform=org.hibernate.dialect.H2Dialect
```

Spring Boot Setup

DevTools dependency

- Incorporating **spring-boot-devtools** as a dependency offers advantageous features during development.
- Particularly, when constructing a web application with embedded **Tomcat**, fast restarts are provided each time a file on your class path is modified and saved.
- DevTools employs an efficient strategy by reloading only the compiled '.class' files that have undergone changes, leading to significantly faster restarts.

Spring Boot Setup

DevTools dependency

- Another optimization for even speedier restarts is to **activate the lazy creation** of our Spring beans using the following property:

```
spring.main.lazy-initialization=true
```

- However, one drawback with this is that your application does not fail fast at start-up.
- Instead, it may only fail when attempting to request a misconfigured bean.
- While this may be acceptable during development, it would be dangerous in a live application so should be turned off in production.

Java EE Web Architecture, Containers and Servlets

Week 09

CSCI 3230U – Web App Development

Dr. Rohollah Moosavi

HTTP

- Generally, HTTP is an **application Layer Protocol.**
- An **application layer protocol** defines how applications communicate with each other in a client/server architecture.
- Here is example of **application layer protocols**:
  - Web Servers/Browsers use HTTP
  - File Transfer Utilities use FTP
  - Electronic Mail applications use SMTP
  - Naming Servers use DNS

HTTP

- HTTP is a lightweight protocol for the web involving a single request & response for communication
- There are 4 Main HTTP Methods:
  - **Get**: Used to request data from server
  - **Post**: Used to post data to the server
  - **Put**: Allows you to "put" (upload) a resource (file) on to a webserver so that it be found under a specified URI.
  - **Delete**: Allows you to delete a resource (file).

HTTP

- **GET** and **POST** allow information to be sent back to the web server from a browser. For example, when you click on the “submit” button of a form, the data in the form is send back to the server, as "name=value" pairs.
- Using an **HTTP GET Request** will append all of the data to the URL and it will show up in the URL bar of your browser. The amount of information you can send back using a GET is restricted as URLs can only be 1024 characters.

HTTP

- A **POST** sends the information through a socket back to the webserver and it won't show up in the URL bar. This allows a lot more information to be sent to the server

- Data sent via **POST** could be secured using HTTPS.

HTTP

- **HTTP** is a **stateless protocol**. So, Request/Response occurs across a single network connection and at the end of the exchange the connection is closed.
- Websites maintain persistent authentication so user does not have to authenticate repeatedly
- Websites also create **sessions** to carry user data across multiple HTTP Request (for example in Amazon once you login).

HTTP - Headers

- HTTP Requests and Responses have a section called **Headers**.

- Generally, **Metadata** about the request/response should be set as a header.

- For Example, a header signifying that the response contains data in JSON format is: Content-Type="application/json"

HTTP - Body

- HTTP Requests and Responses have another section called **body**.
- The **body** of a request and response are used for the following reasons:
  - Submitting a POST/PUT request with form data.
  - Sending data from the server to the client in the response.

HTTP – Forwarding a request

- A **forward** has the below steps:
  - The client sends a request to a web application.
  - The web application receives the request, does some initial processing of the request, then gives it to another request handler in the same application for additional processing.
  - A response is sent from the web application to the client.
- Note that a request CANNOT be forwarded to another server. For example, we CANNOT forward a request from our web application to Google.

HTTP – Forwarding a request

- A **redirect** has the below steps:

    1. An initial request is sent to OUR web application from a client.
    2. A redirect response is sent from OUR web application back to the client with a location header telling the client where to send a second request to.
    3. The client sends a second request to the value of the location header of the response sent to the client in step 2.
    4. The server which handles the request sent to the value of the location header sent to the client in step 2 sends a response back to the client.

- A **redirect** can cause a request to be sent to a server we do **NOT** control!

Web Server

- A **web server** is a program that is running on the server. It **listens and processes requests** as they are sent by a client (mostly a browser).

- It is responsible for **receiving** and **interpreting** HTTP requests, **processing** and **sending back** HTTP responses.

- A web container relies on a web server to provide HTTP message handling.

Web Server

- **Tomcat** is a stand alone web server and a servlet container.

- It is open source and free for usage.

- It is written in Java. But you do not have to be a Java programmer to use it.

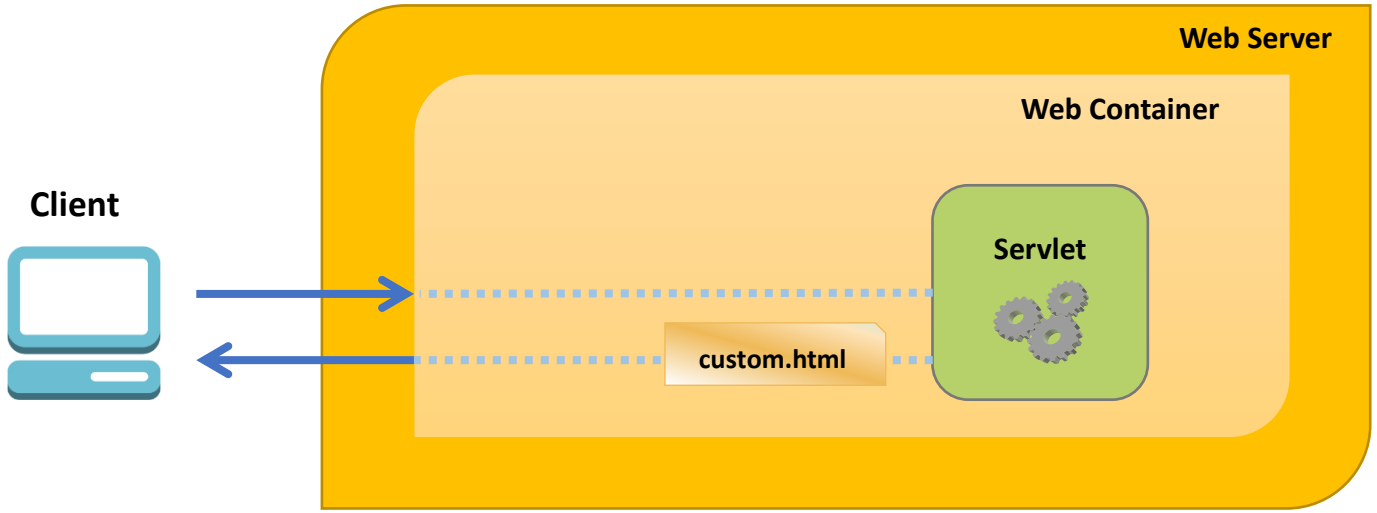
Web Container

- A simple web server application can only serve static resources, means serves the same way every time (images, audio files, simple HTML…), for example, Apache HTTP Server.

- To generate pages dynamically, the help of another application is needed, like web container (or application server), for example, Apache Tomcat

Servlet

- Web **Servlets** are server-side Java components (simple Java classes).
- It process HTTP requests and generate HTTP responses.
  - Running servlet code to generate responses
  - Security (restricting access to server resources)



Web Container

- The **web container** handles incoming HTTP requests by:
  - Running servlet code to generate responses
  - Security (restricting access to server resources)
  - Concurrency (each request is handled in a new thread)
  - Lifecycles of all components (Servlets, JSPs, Filters, Listeners, etc.)
  - Application configuration and deployment

Servlets

- The methods that handle HTTP requests within a Servlet look like this:

```
void doPost (HttpServletRequest req, HttpServletResponse resp) {
    PrintWriter out = resp.getWriter();
    out.println("<html>");
    out.println("<head><title>view Stock</title></head>");
    out.println("<body><p>" + currStock + "</p></body>");
    out.println("</html>");
}
```

Sending information to the Server

- HTML's `<form>` tag lets us send a GET or POST HTTP request:

```
<form method="POST" action="location">
    <input type="text" name="parameterName" />
    <input type="submit" value="Submit" />
</form>
```

- Hitting the 'Submit' button sends an HTTP request to the server
  - The container forwards it to a Servlet mapped to /location
- The `<input>` elements become the request parameters
  - parameterName is the **key**
  - What the user types into the text box is the **value**

Receiving client information from the server side

- Inside a Servlet, user input can be retrieved using parameterName as a lookup key:

```
protected void doPost(HttpServletRequest req, HttpServletResponse resp)
        throws ServletException, IOException {

    PrintWriter out = response.getWriter();
    out.print("<html><body>");
    out.print("You entered:" + req.getParameter("parameterName"));
    out.print(" </body></head>");
}
```

HttpServletRequest and HttpServletResponse

- A **pair of objects** is created to service a single request:
  - The `HttpServletRequest` object contains all data sent by the client.
  - The `HttpServletResponse` object can be used to construct the response.
- And passed to every component that processes the request:

```
protected void doGet (HttpServletRequest req,
                      HttpServletResponse res) {
    // Request processing
}
```

- Both are request scoped and therefore thread-safe

Spring Web - MVC

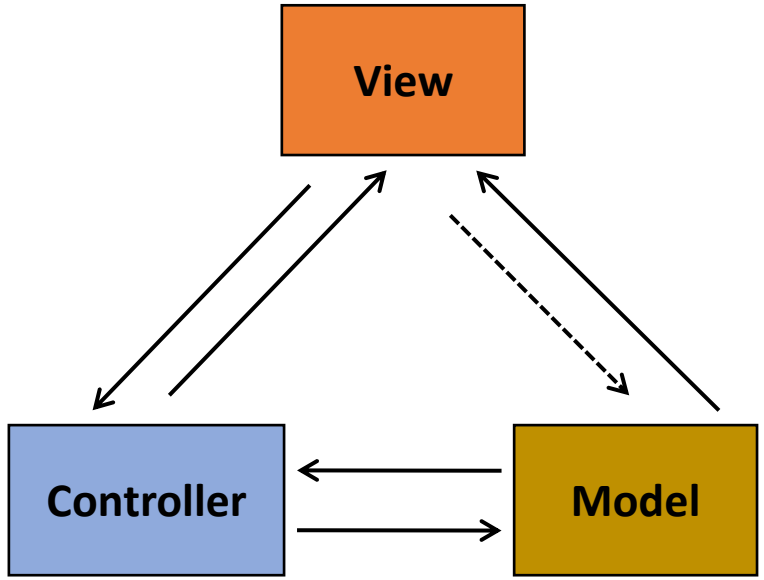
Week 09

CSCI 3230U – Web App Development

Dr. Rohollah Moosavi

MVC

- **MVC** is one of the most common application architecture design patterns.
  - **Model** – the business logic
  - **View** – the user interface
  - **Controller** – manages Model and View interaction

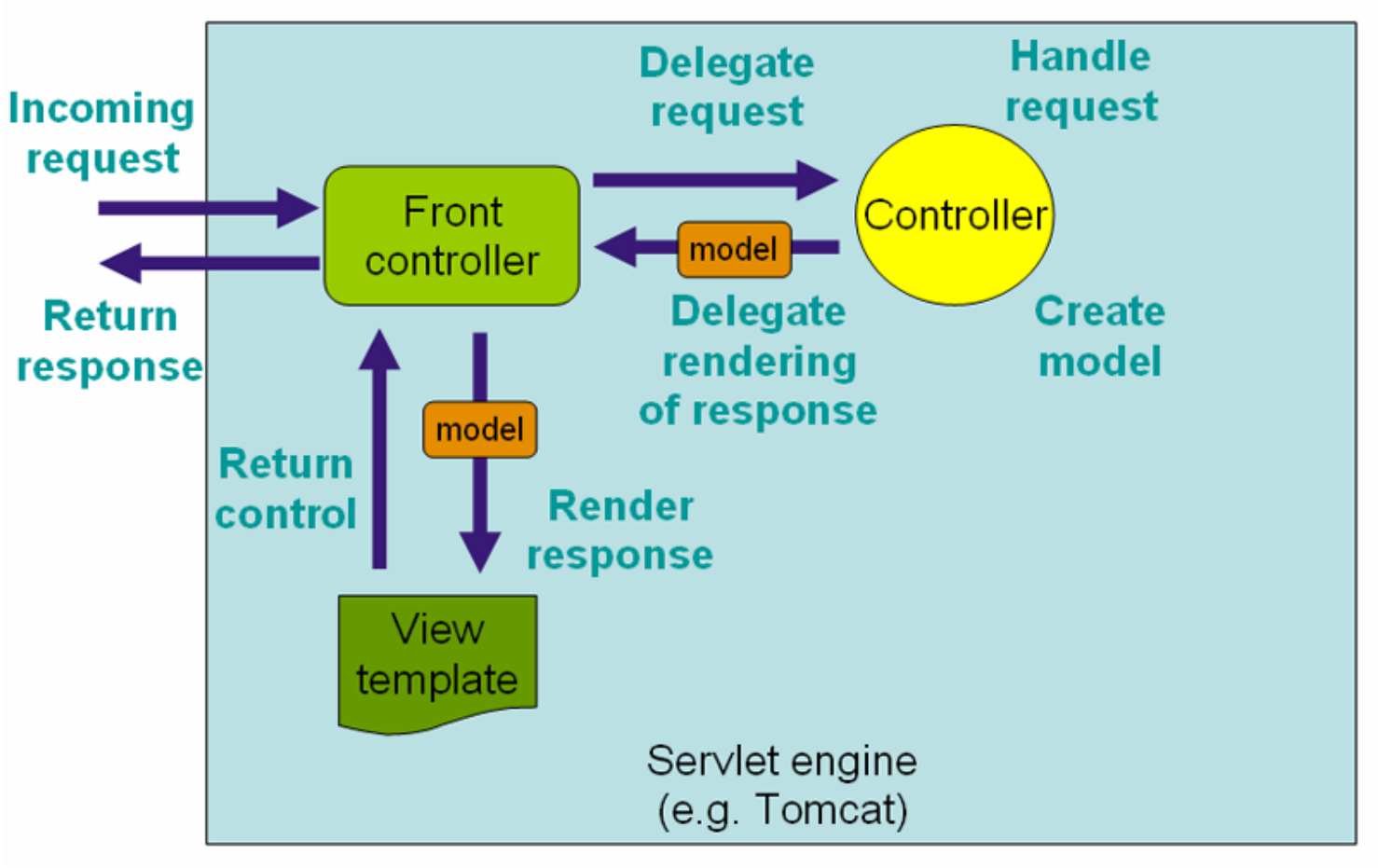


Spring MVC

- Implements the **Front Controller** pattern.
  - All requests are addressed to a **central servlet** (called **DispatcherServlet**)
  - The **servlet** delegates requests to **controller** classes
  - **Controllers** do the request handling work
- Spring facilitates working with Servlets and JSP (No need to create a Servlet for each command).

Spring MVC Architecture

Client



View Resolver

- Spring provides **ViewResolver** that is a class to locate the views.
- By using Spring Boot, the **ViewResolver** is automatically configured to **find HTML templates** in the `src/main/resources/templates` path (Without using Spring Boot, we will have to configure the ViewResolver manually).
- It attaches a specified prefix and suffix to a logical view name.

Controllers

- **DispatcherServlet** delegates request handling to regular Java classes:
  - **@Controller** marks these classes
  - **@RequestMapping** maps a URL to a method
    - **value**: specifies mapped **URL**
    - **method**: specifies **HTTP method**
  - Methods return a **String**, that is a **logical name** of the view to respond with

Controllers

- **@RequestMapping's "method" attribute** allows you to specify which HTTP Method the annotated method will handle.
- If no method is specified, then the method will be invoked by a request with any HTTP Method

```
@Controller
public class HomeController {

    @RequestMapping(value = "/register", method = RequestMethod.POST)
    public String goToRegisterPage() {
        return "registration";
    }

}
```

Controllers

- Also, by clicking on a hyperlink like `<a href="register">Register</a>`, the following method will be executed:

```
@Controller
public class HomeController {

    @RequestMapping("/register")
    public String goToRegisterPage() {
        return "registration";
    }
}
```

- Because it returns “registration”, we will be directed to:

`src/main/resources/templates/registration.html`

Controllers – Forward

- **Forwarding** a request can be performed by putting `forward:` before the destination url.
- For example, here we are forwarding the request to another request handler method with a value of `/additionalProcessing`

```
@Controller
public class HomeController {

    @RequestMapping(value = "/preProcess", method = RequestMethod.GET)
    public String preProcessRequest() {
        return "forward:/additionalProcessing";
    }
}
```

Controllers – Redirect

- **Redirecting** a request can be performed by putting "redirect:" before the url to redirect to.
- We also can carry attributes through a redirect by using the `RedirectAttributes` object

```
@Controller
public class HomeController {

    @RequestMapping(value = "/toGoogle", method = RequestMethod.GET)
    public String goToGoogle() {
        return "redirect:https://www.google.com";
    }
}
```

Controllers

- Here is the list of `@RequestMapping` variants:

1. `@PostMapping("/hello")`
    - `@RequestMapping(value="/hello", method=RequestMethod.POST)`
    - To handle the **HTTP POST requests**
2. `@GetMapping("/hello")`
    - `@RequestMapping(value="/hello", method=RequestMethod.GET)`
    - To handle the **HTTP GET requests**

Controllers

3. `@PutMapping("/hello")`

   - `@RequestMapping(value="/hello", method=RequestMethod.PUT)`
   - To handle the **HTTP PUT requests**

4. `@DeleteMapping("/hello")`

   - `@RequestMapping(value="/hello", method=RequestMethod.DELETE)`
   - To handle the **HTTP DELETE requests**

Request Parameters

- **Request Parameters** are String-to-String key-value pairs located in the request (For a GET request, they are contained in the URL of the request).

- In the following url, key is `bookName` and value is `OperatingSystem`

```
https://library.com/findBook?bookName=OperatingSystem
```

- For a **POST request**, they are contained in the body of the request

Request Parameters

- Methods annotated with `@RequestMapping` can take any number or type of parameters.
- It can be a `HttpServletRequest` to access request data, Strings representing specific request parameters, or a model object such as a User, Various others.

```
@Controller
public class HomeController {

    @GetMapping("/register")
    public String goToRegisterPage(HttpServletRequest req) {
        String type = req.getParameter("userType");
        log("Redirecting to registration page for " + type);
        return type + "registration";
    }
}
```

@RequestParam

- As shown on the previous slide, we can get form parameters from the `HttpServletRequest` object by using `req.getParameter("paramName");`
- Spring provides an alternate way using `@RequestParam`.
  - In the following example, Spring will get the form parameter "username", and pass its value into the method parameter "username".

```
@Controller
public class HomeController {

    @GetMapping("/register")
    public String goToRegisterPage(@RequestParam String username) {
        return "register/" + username;
    }
}
```

@RequestParam

- We can use `@RequestParam`, even on a method parameter with a different name, by using the `value` attribute.
- In the following example, Spring will get the form parameter "username", and pass its value into the method parameter "doesntMatch".

```
@Controller
public class HomeController {

@GetMapping("/register")
public String goToRegisterPage(@RequestParam(value = "username")
                               String doesntMatch) {
        return "register/" + doesntMatch;
    }
}
```

@PathVariable

- We can **bind a variable in the URL** (a placeholder in the request mapping) to a method parameter.

- In this way, user-submitted data (such as a search term) can be displayed on the URL.

```
@Controller
public class LibraryController {

    @GetMapping("/find/{title}")
    public String getBook(@PathVariable String title) {
        .
        .
        .
    }
}
```

Attributes

- **Attributes** are String-to-Object key-value pairs which are set by the developer (not the client).
  - In Spring MVC, we use the **Model object** to add and retrieve attributes.
  - **Model objects** are bound to a single request/response and a new object is created for each request/response.

```
@Controller
public class LibraryController {

    @GetMapping("/register")
    public String goToRegisterPage(Model model) {
        model.addAttribute("attributeName", "attributeValue");
        return "register";
    }
}
```

Sessions

- As mentioned before, **HTTP** is **stateless**; each request is independent of other requests.

- So, we need to use **sessions** to carry attributes/data between multiple requests from the same user.

- Generally, sessions can be used for authentication, authorization, and to carry user-specific information between requests.

Sessions

- Sessions can be managed in Spring using the `HttpSession` object
  - A session created and managed in a Spring application will be identified by the value of the JSESSIONID cookie
- There are many useful methods we can use with the `HttpSession` object
  - `setAttribute(String name, Object value)`, that sets an attribute with a "name=value" on the **session scope**.
  - `getAttribute(String name)`, that gets the value of an attribute with the given name
  - `invalidate()`, to invalidates the current session and unbinds any objects bound to it

Sessions

- When we need to use the **`HttpSession`** object in handling a request, we can simply make our controller method have a parameter of type HttpSession and Spring will automatically add the HttpSession object of this request when it calls this controller method.

```
@PostMapping("/login")
public String processLogin(LoginDto loginDto, HttpSession httpSession) {
    boolean successfulLogin = this.userService.login(loginDto);
    if (successfulLogin) {
        String username = loginDto.getUsername();
        httpSession.setAttribute("username", username);
        return "redirect:/home";
    }
    return "redirect:/failedLogin";
}
```

Spring Web - Thymeleaf

Week 10

CSCI 3230U – Web App Development

Dr. Rohollah Moosavi

Thymeleaf

- **Thymeleaf** is a **modern server-side Java template engine** used to process HTML, XML, JavaScript, and CSS.

- It serves as a library compatible with Spring, facilitating the generation of HTML templates.

- https://www.thymeleaf.org/doc/tutorials/3.0/usingthymeleaf.html#what-is-thymeleaf

Thymeleaf

- Thymeleaf-generated HTML files function as static web pages, ideal for rapid prototyping purposes.

- Thymeleaf seamlessly integrates with Spring and comes pre-equipped with support for Spring Expression Language.

- Thymeleaf attributes allow the developer to easily create dynamic HTML pages by binding model attributes to HTML.

Thymeleaf Dependency

- We need to add the following dependency for our Spring Boot application:

```
<dependency>
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-starter-thymeleaf</artifactId>
</dependency>
```

Thymeleaf Expressions

- To use Thymeleaf in a HTML file, we must set the Thymeleaf namespace.

```
<html xmlns:th="http://www.thymeleaf.org">
```

- The "`th`" after "`xmlns`" specifies how we will use the **attributes Thymeleaf** provides.

- Same like other Spring Boot application, all HTML templates must be located in `src/main/resources/templates`

Thymeleaf Variable Expressions

- We can access value of a model attribute by using:

```
${modelAttributeName}
```

- To add an attribute to the model (for example "username" here):

```
model.addAttribute("username", "myUsername");
```

- And to display the value of the attribute on the view side, we need to use `th:text`:

```
<span th:text="${username}"></span>
```

Thymeleaf Variable Expressions

- We can access session attributes by using:

```
${session.attributeName}
```

- To add an attribute to the httpSession object:

```
httpSession.setAttribute("username", "myUsername");
```

- And to access the session attribute on the view side:

```
<span th:text="${session.username}"></span>
```

Thymeleaf Selection Expressions

- We can access values of fields of model attributes with:

```
*{attributeName}
```

- For example, here we use `th:object` to get the **model attribute**:

```
<div th:object="${user}">
```

- And we use `th:text` to display the value of a **field of the model attribute**:

```
<span th:text="*{firstName}"></span>
```

Thymeleaf Link Expressions

- We can create URLs on the view with `@{url}` with `th:href` on **non-form tags** and `th:action` on **form tags**.

- If we deploy a web app named myWebApp then the application root is myWebApp and we can create urls relative to it. For example:

```
<a th:href="@{/home}"></a>
```

Thymeleaf Link Expressions

- We can manually add request parameters to the url in a link expression by using `(parameterName=parameterValue)`.

- For example:

```
<a th:href="@{/home(username=myUsername)}"></a>
```

- We can also use a variable expression as the value:

```
<a th:href="@{/home(username=${username})}"></a>
```

Thymeleaf Attributes

- To Iterate through a collection of User objects with two attributes `id` and `username`, we can use `th:each` and `th:text`

- For example:

```
<div th:each="user : ${users}">
    <p th:text="${user.id}"/>
    <p th:text="${user.username}"/>
</div>
```

Thymeleaf Attributes

- We also can use if else logic by using `th:switch`. For example:

```
model.addAttribute("user", appContext.getBean(User.class, 1, "userOne"));
```

```
<div th:switch="${user.username}">
   <div th:case="userOne">
      <p>First User</p>
   </div>
   <div th:case="userTwo">
      <p>Second User</p>
   </div>
   <div th:case="*">
      <p>Default Case</p>
   </div>
</div>
```

Thymeleaf Attributes

- The same thing using `th:if` but without default case.

- For example:

```
<div>
   <div th:if="${user.username} eq userOne">
      <p>First User</p>
   </div>
   <div th:if="${user.username} eq userTwo">
      <p>Second User</p>
   </div>
</div>
```

Thymeleaf Attributes

- We can create HTML fragments (header/footer) using `th:fragment`

```
<header th:fragment="headerFragment">
```

- And to use a HTML fragment on another HTML file we can use `th:insert`

```
<div th:insert="fragmentLocation :: headerFragment">
```

Thymeleaf Attributes

- We can bind a HTML form to a url with Thymeleaf with `th:action`

```
<form th:action="@{url}">
</form>
```

- We can bind model objects to Thymeleaf forms. For that, we need to use `th:object` and `th:field` to bind the form to the User object added to the model.

```
<form th:action="@{url}" th:object="${user}" >
   <label>Username: </label>
   <input type="text" th:field="*{username}" />
</form>
```

Spring Data JPA

Week 11

CSCI 3230U – Web App Development

Dr. Rohollah Moosavi

Exam Announcement

1. **Date and Time**: The exam is scheduled for March 28th Thursday. It will take place during the synchronous lecture from 11:10 am to 12:30 pm.
2. **Location**: The exam will be conducted online, using Google Meet, the same platform used for classes. Students are expected to join the exam session using the same Google Meet link provided for classes.
3. **Format**: The exam consists of two parts:
   - **Part 1**: Short answer explanation questions, which constitute 8% of the final grade.
   - **Part 2**: Coding section, similar to class/lab demonstrations, worth 26% of the final grade.
4. **Coverage**: The exam will cover all material from the beginning of the course to the last lecture. This implies that students need to be familiar with all topics discussed throughout the course.
5. **Resources**: During the exam, students are permitted to refer to course lecture notes, labs, class demos, and their own code. However, they are not allowed to seek help from other students.
6. **Attendance**: Students are required to be online for the entire duration of the exam session, and attendance will be automatically recorded.

Spring Data JPA

- Spring Data provides a layer on top of JPA that offers convenient ways to reduce the amount of code you need to write.
- Spring also abstracts and automatically manages the injection of the JPA EntityManager through its Repository interfaces:
  - **Repository interface** takes the **domain class** and the **ID** type of the domain class as type arguments.
  - **CrudRepository interface** extends **Repository** to provide CRUD functionality.
  - **JpaRepository interface** extends **CrudRepository** to give us the JPA specific features.

Spring Data JPA

- Here is the list of the methods can use:
  - `getOne(ID id)`
  - `exists(T entity)`
  - `existsById(ID id)`
  - `deleteById(ID id)`
  - `deleteAll(Iterable<? extends T> entities)`
  - `saveAll(Iterable<S> entities)`
  - …

https://docs.spring.io/spring-data/jpa/docs/current/api/org/springframework/data/jpa/repository/JpaRepository.html

Spring Data JPA

- The following entries will provide the database connection details for H2 DB, that need to be added in application.properties file:

```
spring.jpa.hibernate.ddl-auto=update
spring.h2.console.enabled=true
spring.datasource.url=jdbc:h2:C:/H2/springclassjpa;AUTO_SERVER=true
spring.datasource.driverClassName=org.h2.Driver
spring.datasource.username=sa
spring.datasource.password=
spring.jpa.database-platform=org.hibernate.dialect.H2Dialect
```

- And to direct access to EntityManager we can inject it with:

```
@PersistenceContext
EntityManager entityManager;
```

Spring Data JPA

- In addition to the mentioned JpaRepository methods, Spring also provides two extra techniques to help define your own queries. The first of these is **derived queries**.

- For simple queries, Spring can easily derive what the query should be from just the **method name**.

Spring Data JPA

- The convention is that method names have two main parts, separated with the **By** keyword. For example:

```
List<Book> findByName (String name);
```

- The first part "find" is the **introducer** and the remainder, in this case "ByName", is the **criteria**.

- **Derived queries** can also use find, read, query and get as the introducer:

```
List<Book> queryByName(String name);

List<Book> getByName(String name);
```

Spring Data JPA

- To create more complex queries with joins or with many parameters (custom queries), Spring Data provides an `@Query` annotation for JPQL or native queries.
- We can annotate a repository method with the `@Query` annotation where the value contains the JPQL or SQL to execute.
- For example:

```
@Query("select b from Book b where upper(b.title) like concat('%', upper(:title), '%')")
List<Book> findByTitle(@Param("title") String title);
```

  - Here, the parameter and placeholder is defined with `:title` and `@Param("title")`
  - Also, no method body required here as Spring will provide the boilerplate code in order to execute the JPA select statement.

Spring Data JPA