Programação Orientada a Objetos com Spring

<Cristian César Martins - 799714>

<Eduardo Santiago Rosalin - 802539>

<Vitor Gabriel Orsin - 801575>

<Vitor Matheus da Silva - 800260>

Referências

- https://docs.spring.io/spring-framework/reference/
- https://docs.spring.io/spring-framework/reference/core/beans.html
- https://www.baeldung.com/spring-framework-design-patterns
- https://medium.com/@bubu.tripathy/design-patterns-used-in-spring-framework-60df94fd3400
- https://medium.com/@saygiligozde/applying-solid-principles-to-spring-boot-applications-191d7e50e1b3
- Spring in Action, Craig Walls e Ryan Breidenbach - Livro

Tópicos

1. Introdução ao Spring
2. Anotações Spring para POO
3. Gerenciamento de Beans e Ciclo de Vida
4. Container de Inversão de Controle ⇔ Injeção de Dependências do Spring
5. Padrões de Design no Spring
6. Princípios SOLID aplicados no Spring

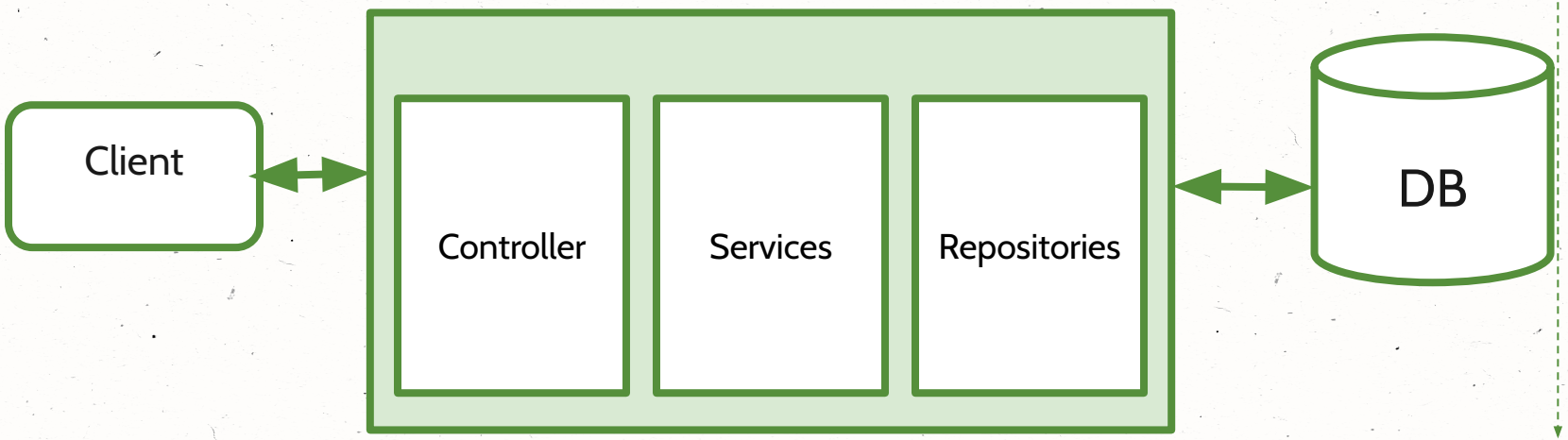
Introdução ao Spring

Framework open-source para desenvolvimento de aplicações Java, com objetivo de simplificar o Java EE

Principais características:

- Leve e modular
- Inversão de Controle (IoC) e Injeção de Dependências (DI)
- Abstração de complexidade técnica
- Suporte a POO e padrões de projeto

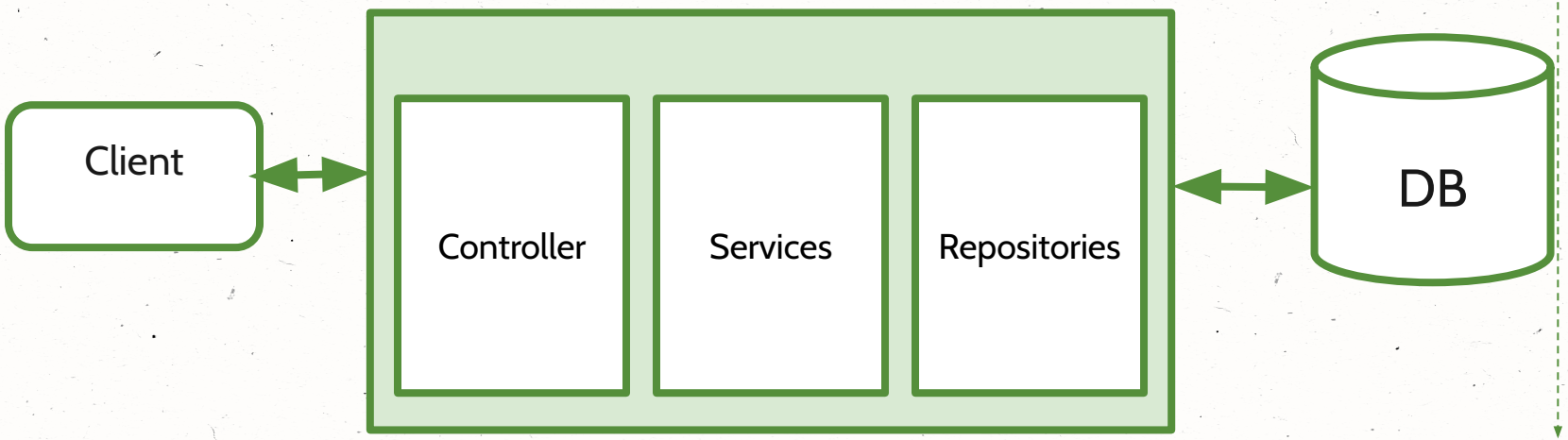
Introdução ao Spring: Inversion of Control




```
Class Controller {

        Service service;

}
```

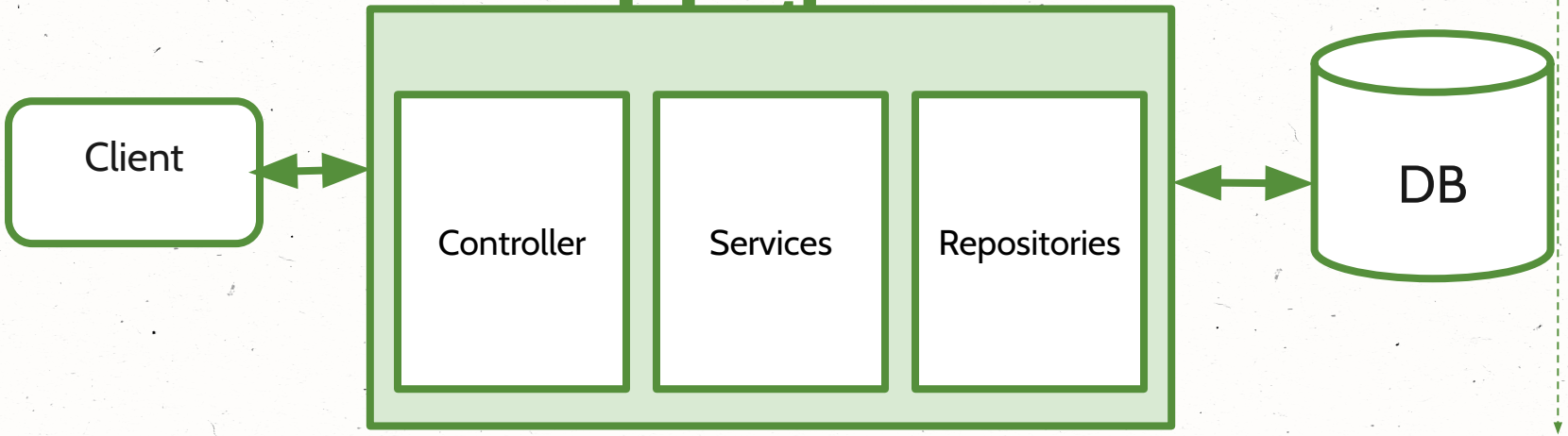
```
Class Service{

        Repository repository;

}
```

Introdução ao Spring: Inversion of Control



Deixar o controle de criação de objetos para outra “pessoa”. O Spring faz isso por meio de Dependency Injection!

Introdução ao Spring: Dependency




```
// Constructor injection

Class Controller {

        Public Controller ( Service service) { ... }

}
```

```
//Field Injection

Class Service{

        @Autowired

        Private Repository repository;
```

Introdução ao Spring

Framework open-source para desenvolvimento de aplicações Java

Módulos Principais:

- Spring Core: Beans, Context, SpEL: Linguagem de expressões
- Spring MVC: Framework para aplicações web, Programação reativa, Comunicação bidirecional
- Spring Data: JdbcTemplate para operações SQL, Integração com Hibernate
- Spring Security: Autenticação e autorização, Proteção contra ataques comuns
- Spring Boot: Autoconfiguração, Starter dependencies, Embedded servers

Anotações Spring para POO

Anotações básicas para definição de componentes:

- @Component - Marca uma classe como componente gerenciado pelo Spring
- @Service - Especialização para classes de serviço (camada de negócios)
- @Repository - Especialização para classes de acesso a dados
- @Controller/@RestController - Para classes controller (MVC)
- @Transactional - ACID (Atomicidade, Consistência, Isolamento, Durabilidade) nas transações

Anotações Spring para POO

Anotações para Injeção de Dependência (DI):

- @Autowired - Injeção automática de dependências
- @Qualifier - Especifica qual bean injetar quando há múltiplas implementações
- @Primary - Define a implementação padrão para injeção
- @Value - Injeta valores de propriedades

Anotações Spring para POO

Anotações para Configuração e Escopos:

- @Configuration - Define uma classe como fonte de definições de beans
- @Bean - Método que produz um bean para ser gerenciado pelo container
- @Scope - Define o escopo de um bean (singleton, prototype, etc.)
- @Profile - Ativa beans/configurações específicas para perfis

Anotações Spring para POO

```java
// @Service indica que esta classe é um componente de serviço
// (camada de lógica de negócios) que será gerenciado pelo Spring
@Service
public class MeuServico {

    // Boa prática: declarar dependências como final
    private final MeuRepositorio repositorio;

    // @Autowired faz a injeção automática da dependência
    // @Qualifier especifica qual implementação usar quando há múltiplas opções
    @Autowired
    public MeuServico(@Qualifier("meuRepositorioEspecial") MeuRepositorio repositorio) {
        this.repositorio = repositorio;
    }

    // @Transactional gerencia automaticamente:
    // - Inicia transação antes do método
    // - Commita se sucesso
    // - Rollback se exception
    @Transactional
    public void executarLogica() {
        // Lógica de negócios que usa o repositório
        repositorio.processarDados();
    }
}
```

- **Abstração**: Interface `MeuRepositorio`
- **Polimorfismo**: `@Qualifier` selecionando implementação
- **Encapsulamento**: Dependências injetadas, não instanciadas diretamente

```java
// @Repository marca como componente de acesso a dados
// (trata exceções específicas de persistência)
@Repository
// @Profile ativa este bean apenas quando o perfil "producao" estiver ativo
@Profile("producao")
public class MeuRepositorioEspecial implements MeuRepositorio {
    // Implementação específica para ambiente de produção
    @Override
    public void processarDados() {
        System.out.println("Processando em produção com validações extras");
    }
}
```

Anotações Spring para POO

```java
// Classe principal do Spring Boot
@SpringBootApplication
public class DemoApplication {

    public static void main(String[] args) {
        // Obtém o ApplicationContext
        ApplicationContext context = SpringApplication.run(DemoApplication.class, args);

        // Obtém o componente do contexto
        MeuComponente componente = context.getBean(MeuComponente.class);

        // Usa o componente
        componente.executarTarefa();
    }
}
```

O Spring controla a instância!

```java
// Componente gerenciado pelo Spring
@Component
public class MeuComponente {

    private static final Logger logger = LoggerFactory.getLogger(MeuComponente.class);

    public void executarTarefa() {
        logger.info("Tarefa executada pelo componente!");
    }
}
```

Como Spring se relaciona com POO

- **Princípios POO aplicados no Spring**:
  a. **Abstração**: Interfaces e classes abstratas são amplamente utilizadas
  b. **Encapsulamento**: Beans gerenciados pelo Spring têm seu estado protegido
  c. **Herança**: Suporte a herança entre classes de configuração
  d. **Polimorfismo**: Injeção de diferentes implementações de interfaces

- **Spring implementa/amplia conceitos POO**:
  a. Padrão Factory (BeanFactory)
  b. Inversão de Controle
  c. Baixo acoplamento através de DI

Gerenciamento de Beans e Ciclo de Vida

O que é um Bean?

No contexto do Spring, um Bean é qualquer objeto que é criado, configurado e gerenciado pelo Inversion of Control (IoC) Container.

Ao registrar uma classe como bean, você delega ao Spring a responsabilidade de cuidar do ciclo de vida dessa instância, desde a sua criação até sua eventual destruição.

Em vez de instanciar manualmente com *new*, você anota a classe com @Component, @Service, etc., e o Spring vai ser o responsável.

Escopos

- **singleton** *(padrão)*: Um único bean para toda a aplicação. Compartilhado e gerenciado pelo Spring, incluindo destruição automática.
- **prototype**: Um novo bean a cada solicitação. O Spring cria, mas **não gerencia a destruição**. Útil para objetos com estado isolado.
- **request**: Um bean por requisição HTTP. Usado em aplicações web para controlar dados temporários.
- **session**: Um bean por sessão de usuário. Usado para armazenar estado entre requisições, como login ou carrinho de compras.

Como Spring identifica os Beans?

1. A classe principal é anotada com @SpringBootApplication, que contém a anotação @ComponentScan
2. @ComponentScan irá escanear todas as classes com anotações abaixo dela.
3. Será feita uma varredura de todas as classes anotadas (Component, Service, Repository, Controller)
4. Para cada classe encontrada:
   a. É criadas uma instância (bean)
   b. Será registradas no container (*ApplicationContext*)

```
@Target({ElementType.TYPE})  2 usages
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Inherited
@SpringBootConfiguration
@EnableAutoConfiguration
@ComponentScan(
        excludeFilters = {@Filter(
                type = FilterType.CUSTOM,
                classes = {TypeExcludeFilter.class}
        ), @Filter(
                type = FilterType.CUSTOM,
                classes = {AutoConfigurationExcludeFilter.class}
        )}
)
public @interface SpringBootApplication {
```

Processo de Ligação de Beans

1. O contêiner encontra a definição do bean e **instancia o bean.**
2. Usando injeção de dependência, o **Spring preenche todas as propriedades**.
3. Se o bean implementa a interface BeanNameAware, a fábrica chama o método setBeanName(), passando o ID do bean.
4. Se o bean implementa a interface BeanFactoryAware, a fábrica chama setBeanFactory(), passando uma instância de si mesma.
5. Se o bean implementar a interface ApplicationContextAware, o método setApplicationContext() será chamado.
6. Se houver BeanPostProcessors associados ao bean, seus métodos postProcessBeforeInitialization() serão chamados.
7. Se um método de inicialização (init-method) estiver especificado para o bean, ele será chamado.
8. Finalmente, se houver BeanPostProcessors associados ao bean, seus métodos postProcessAfterInitialization() serão chamados.

- *Tradução do trecho do livro Spring Action, pg. 48*

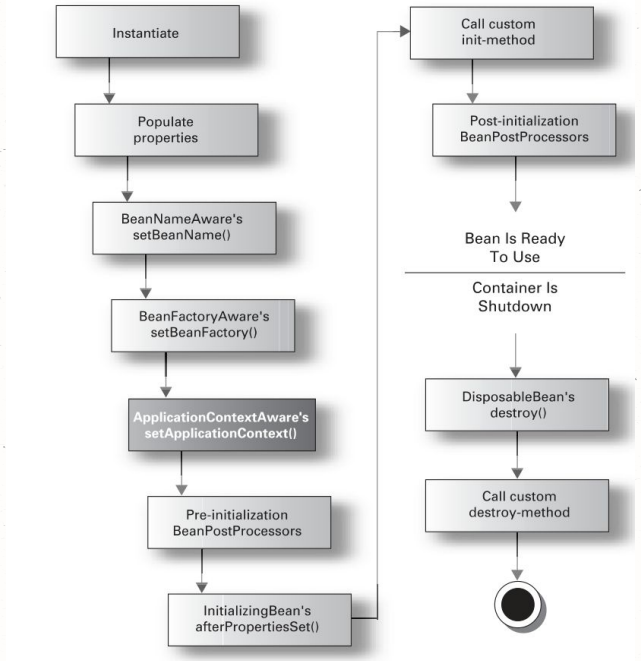


Figure 2.2 The life cycle of a bean in a Spring application context

Ciclo de Vida

- **De forma resumida:**
  1. Instanciação
     a. Identificado pelo *@ComponentScan*
  2. Injeção de dependência
     a. Anotação *@Autowired*
  3. Inicialização
     a. Chama os métodos anotados como *@PostConstruct*
  4. Uso do Bean
     a. Pronto para ser usado
  5. Destruição (quando aplicável pelo escopo)
     a. Quando o *ApplicationContext* é encerrado

Beans nas Configurações

- Declarar métodos @Bean dentro de uma classe marcada com @Configuration garante que o Spring controle do ciclo de vida.
- Evita que o mesmo método seja chamado manualmente, criando múltiplas instâncias.
- Assegura que o bean seja registrado corretamente no ApplicationContext.

```java
@Configuration  no usages
public class AppConfig {

    @Bean  no usages
    public MyServiceImpl myService() {
        return new MyServiceImpl();
    }
}
```

Exemplo do Ciclo de Vida

Classe com anotação @Component

```java
@Component  no usages
public class Knight {

    private final Sword sword;  1 usage

    public Knight(Sword sword) {  no usages
        this.sword = sword;
    }

    //! executado automaticamente após o bean ser criado e as dependências injetadas.
    @PostConstruct  no usages
    public void init() {
        System.out.println("Bom dia grupo!");
    }


    //! executado quando o contexto da aplicação está sendo encerrada.
    @PreDestroy  no usages
    public void destroy() {
        System.out.println("Cavaleiro está indo dormir, grupo!");
    }
}
```

Classe sem a anotação @Component mas usada em outro bean gerenciado

```java
//@Component
public class Sword {  2 usages
    public void attack() {  1 usage
        System.out.println("ATTACK!");
    }
}
```

Container do Spring não consegue encontrar a classe:

```
***************************
APPLICATION FAILED TO START
***************************

Description:

Parameter 0 of constructor in com.example.ATA01.Knight required a bean of type 'com.example.ATA01.Sword' that could not be found.


Action:

Consider defining a bean of type 'com.example.ATA01.Sword' in your configuration.
```

Anotações @PostConstruct e @PreDestroy

Exemplo da classe Knight:

```java
//! executado automaticamente após o bean ser criado e as dependências injetadas.
@PostConstruct  no usages
public void init() {
    System.out.println("Bom dia grupo!");
}


//! executado quando o contexto da aplicação está sendo encerrada.
@PreDestroy  no usages
public void destroy() {
    System.out.println("Cavaleiro está indo dormir, grupo!");
}
```

- @PostConstruct: executado após a injeção de dependências
  a. Carregar configurações
  b. Iniciar recursos (conexões e caches)
- @PreDestroy: executado antes da destruição do bean
  a. Fechar conexões com banco de dados
  b. Encerrar threads
  c. Salvar informações pendentes

Imprimindo Beans encontrados no Application Context

```java
@SpringBootApplication
public class Ata01Application {

    public static void main(String[] args) {
        ConfigurableApplicationContext app = SpringApplication.run(Ata01Application.class, args);
        for (String s : app.getBeanDefinitionNames()) {
            System.out.println(s);
        }
    }

}
```

```
2025-06-10T21:01:50.768-03:00  INFO 27236 --- [ATA01] [           main] w.s.c.ServletWe
 WebApplicationContext: initialization completed in 787 ms
Bom dia grupo!
2025-06-10T21:01:51.092-03:00  INFO 27236 --- [ATA01] [           main] o.s.b.w.embedde
 8080 (http) with context path '/'
2025-06-10T21:01:51.100-03:00  INFO 27236 --- [ATA01] [           main] com.example.ATA
 in 1.639 seconds (process running for 1.92)
org.springframework.context.annotation.internalConfigurationAnnotationProcessor
org.springframework.context.annotation.internalAutowiredAnnotationProcessor
org.springframework.context.annotation.internalCommonAnnotationProcessor
org.springframework.context.event.internalEventListenerProcessor
org.springframework.context.event.internalEventListenerFactory
ata01Application
org.springframework.boot.autoconfigure.internalCachingMetadataReaderFactory
knight
sword
org.springframework.boot.autoconfigure.AutoConfigurationPackages
```

- Durante a inicialização, o método init() da classe Knight foi executado
- No início da aplicação, podemos ver que a classe Sword e Knight foram encontradas

- Ao fim da aplicação:

```
2025-06-10T21:02:22.437-03:00  INFO 27236 -
2025-06-10T21:02:22.555-03:00  INFO 27236 -
Cavaleiro está indo dormir, grupo!

Process finished with exit code 130
```

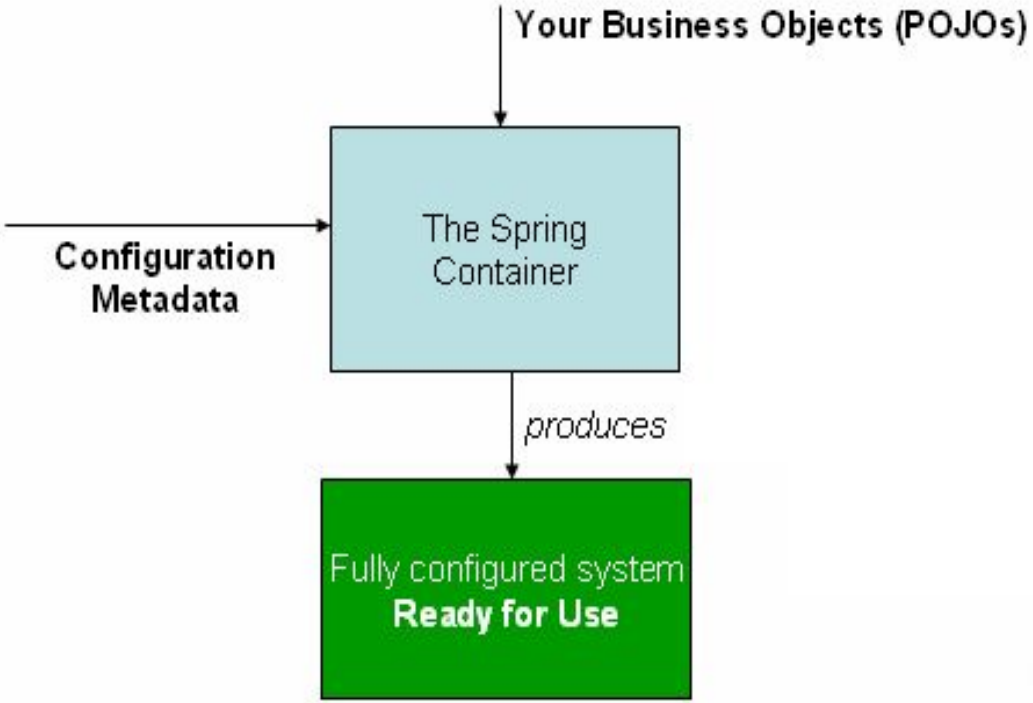
Container IoC

- **Injeção de Dependência**
  - “Módulos de alto nível não devem depender de módulos de baixo nível.”

- **Inversão de Controle**
  - Forma de Inversão de Dependência;
  - As dependências de um módulo só podem ser definidas através do construtor ou por Setters;
  - As dependências são injetadas no módulo usando desses métodos.

Container IoC

- **Container IoC**
  - Faz parte do core do *framework* Spring;
  - É responsável por injetar as dependências do módulo durante a criação do Bean;



ApplicationContext

- Representa o Container IoC no código. É uma interface essencial do *framework*, e possui diversas implementações para usos distintos, com destaque para AnnotationConfigApplicationContext e ClassPathXmlApplicationContext.
- Cada implementação dessa interface representa uma estratégia para realizar a injeção de dependência dos Beans.
- A interface também fornece métodos para configurar e acessar os Beans.

```
1  // create and configure beans
2  ApplicationContext context = new ClassPathXmlApplicationContext("services.xml", "daos.xml");
3
4  // retrieve configured instance
5  PetStoreService service = context.getBean("petStore", PetStoreService.class);
6
7  // use configured instance
8  List<String> userList = service.getUsernameList();
```

Configura ção de Metadados - Annotations vs XML

Annotations

- A definição dos Beans é feita por anotações do Java (ex.: @Autowired);
- Mais intuitivo, e mantém a lógica mais próxima do componente;
- O desenvolvedor precisa se envolver muito pouco com a estruturação dos Beans;

XML

- A definição dos Beans é feita em arquivo externo em XML;
- Código mais “Limpo”;
- Organização dos Beans mais clara e bem estruturada;
- Mais trabalhoso de produzir e manter atualizado

Configura ção de Metadados - XML vs

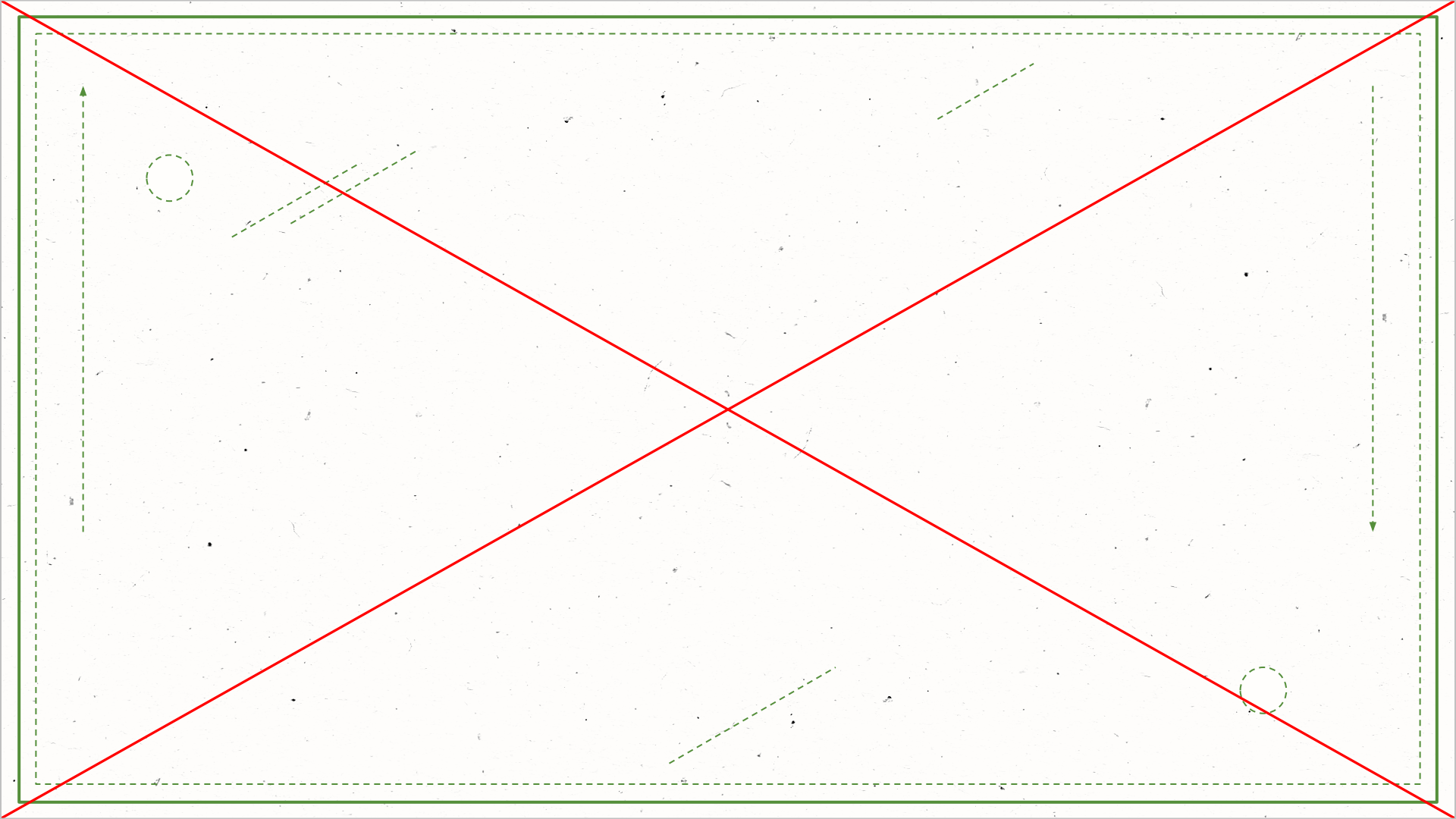
```java
public class MovieRecommender {

    private final CustomerPreferenceDao customerPreferenceDao;

    @Autowired
    public MovieRecommender(CustomerPreferenceDao customerPreferenceDao) {
        this.customerPreferenceDao = customerPreferenceDao;
    }

    // ----- //

    private MovieFinder movieFinder;

    @Autowired
    public void setMovieFinder(MovieFinder movieFinder) {
        this.movieFinder = movieFinder;
    }

    // ----- //

    @Autowired
    private MovieCatalog movieCatalog;

    @Autowired
    public MovieRecommender(CustomerPreferenceDao customerPreferenceDao) {
        this.customerPreferenceDao = customerPreferenceDao;
    }
}
```

Configura ção de Metadados - XML vs Anotetions

```xml
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        https://www.springframework.org/schema/beans/spring-beans.xsd">

    <bean id="petStore" class="org.springframework.samples.jpetstore.services.PetStoreServiceImpl">
        <property name="accountDao" ref="accountDao"/>
        <property name="itemDao" ref="itemDao"/>
        <!-- additional collaborators and configuration for this bean go here -->
    </bean>

    <bean id="accountDao"
        class="org.springframework.samples.jpetstore.dao.jpa.JpaAccountDao">
        <!-- additional collaborators and configuration for this bean go here -->
    </bean>

    <bean id="itemDao" class="org.springframework.samples.jpetstore.dao.jpa.JpaItemDao">
        <!-- additional collaborators and configuration for this bean go here -->
    </bean>

</beans>
```

Padrões de Design

–Spring–



Padroes de Design em Spring - Singleton

Anotações que criam singletons automaticamente:

- @Service
- @Component
- @Repository
- @Controller

```java
@Service
public class ConfigurationService {
    // Por padrão, Spring cria apenas uma instância (Singleton)
    private Properties config = new Properties();

    public String getConfig(String key) {
        return config.getProperty(key);
    }
}
```

Padroes de Design em Spring - Factory

- Conseguimos utilizar com o @Value os tipos de pagamento, neste exemplo, e com isso fazer a criação do tipo de serviço de pagamento com o Factory, onde a criação se torna independente da aplicação.

```java
@Configuration
public class ServiceFactory {

    @Bean
    public PaymentService paymentService(@Value("${payment.type}") String type) {
        // Factory pattern - cria diferentes implementações
        switch (type) {
            case "credit":
                return new CreditCardPaymentService();
            case "pix":
                return new PixPaymentService();
            default:
                return new DefaultPaymentService();
        }
    }
}
```

Padroes de Design em Spring - Template

Muito usado para queries em banco de dados, seguindo um padrão de etapas:

- Estabelecer conexão
- Executar a query
- Finalização (Cleanup)
- Fechar conexão

Lida automaticamente com Exceptions

```java
@Service
public class UserDataService {
    private final JdbcTemplate jdbcTemplate;

    public UserDataService(JdbcTemplate jdbcTemplate) {
        this.jdbcTemplate = jdbcTemplate;
    }

    public List<User> findAllUsers() {
        // Template Method - Spring gerencia conexão, transação, etc.
        return jdbcTemplate.query(
            "SELECT * FROM users",
            (rs, rowNum) -> new User(
                rs.getString("name"),
                rs.getString("email")
            )
        );
    }
}
```

Padroes de Design em Spring - Template

- Classe presente no Spring, fazendo automaticamente os requisitos do padrão Template. Com isso, é garantido que o funcionamento não tenha erros comuns de acesso a banco.

```
public class JdbcTemplate {

    public <T> T query(final String sql, final ResultSetExtractor<T> rse) throws DataAccessException {
        // Execute query...
    }

    // Other methods...
}
```

Padroes de Design em Spring - Observer

- Criação de Observer bem próxima do padrão

```java
// Evento personalizado
public class UserCreatedEvent extends ApplicationEvent {
    private final User user;

    public UserCreatedEvent(Object source, User user) {
        super(source);
        this.user = user;
    }

    public User getUser() { return user; }
}
```

Padroes de Design em Spring - Observer

- Ao criar um usuário, um evento é enviado com o eventPublisher.
- Este será observado por outras classes, tendo uma anotação para facilitar esse trabalho.

```java
// Publisher
@Service
public class UserService {
    private final ApplicationEventPublisher eventPublisher;

    public UserService(ApplicationEventPublisher eventPublisher) {
        this.eventPublisher = eventPublisher;
    }

    public User createUser(String name, String email) {
        User user = new User(name, email);
        // Publica evento
        eventPublisher.publishEvent(new UserCreatedEvent(this, user));
        return user;
    }
}
```

Padroes de Design em Spring - Observer

- A anotação @EventListener facilita a criação desse padrão de projeto, tendo um listener para o evento de criação de Usuário e enviando um email para o usuário.

```java
// Observer/Listener
@EventListener
@Component
public class EmailNotificationListener {

    public void handleUserCreated(UserCreatedEvent event) {
        User user = event.getUser();
        System.out.println("Enviando email de boas-vindas para: " + user.getEmail(
    }
}
```

Padroes de Design em Spring - Strategy

- Criando “estratégias” diferentes de desconto, podendo aqui ser de porcentagem ou valor fixo.
- Configura com **@**Service os nomes para serem chamados posteriormente, não precisando saber o nome correto das classes.

```java
public interface DiscountStrategy {
    BigDecimal calculateDiscount(BigDecimal amount);
}

@Service("percentageDiscount")
public class PercentageDiscountStrategy implements DiscountStrategy {
    public BigDecimal calculateDiscount(BigDecimal amount) {
        return amount.multiply(new BigDecimal("0.10")); // 10%
    }
}

@Service("fixedDiscount")
public class FixedDiscountStrategy implements DiscountStrategy {
    public BigDecimal calculateDiscount(BigDecimal amount) {
        return new BigDecimal("50.00"); // R$ 50 fixo
    }
}
```

Padroes de Design em Spring - Strategy

- Com o @Qualifier é possível determinar a classe específica que será utilizada para a estratégia de desconto.
- Rápido de entender e configurar a estratégia atual.

```java
@Service
public class PricingService {
    private final DiscountStrategy discountStrategy;

    // Injeta estratégia específica
    public PricingService(@Qualifier("percentageDiscount") DiscountStrategy strategy) {
        this.discountStrategy = strategy;
    }

    public BigDecimal calculateFinalPrice(BigDecimal originalPrice) {
        BigDecimal discount = discountStrategy.calculateDiscount(originalPrice);
        return originalPrice.subtract(discount);
    }
}
```

Princípios SOLID

–Spring–

Principios SOLID em Spring

- Ligação entre Single Responsibility Principle e as anotações de spring (@Service, @Repository, @Controller), que auxiliam em manter uma única responsabilidade para cada classe.
- Open/Closed principle possui @Aspect, deixando com que aquele código possa ser estendido, podendo algo ser executado em algumas partes específicas do fluxo.
- Liskov's Substitution Principle não possui grande diferença
- Interface Segregation Principle não possui grandes diferenças, temos interfaces para utilização apenas de métodos necessários para um componente.
- Ligação grande entre Dependency Inversion Principle e IoC's - Utilizando para fazer a inversão de dependência, desacoplando o código.

Principios Open/Closed em Spring

- @Aspect “avisa” o Spring que tem adições de métodos naquela classe que serão rodadas em algum momento do fluxo de execução
- logBefore será executado antes de createUser()
- logAfter será executado depois da finalização do método createUser()

```java
// Original service class - closed for modification
@Service
public class UserService {
    public void createUser(String username) {
        System.out.println("Creating user: " + username);
    }
}

// Extension through AOP - open for extension
@Aspect
@Component
public class LoggingAspect {

    @Before("execution(* com.example.UserService.createUser(..))")
    public void logBefore(JoinPoint joinPoint) {
        System.out.println("LOG: About to create user");
    }

    @After("execution(* com.example.UserService.createUser(..))")
    public void logAfter(JoinPoint joinPoint) {
        System.out.println("LOG: User creation completed");
    }
}
```

Dúvidas?

CONTENTS OF THIS TEMPLATE

Here's what you'll find in this **Slidesgo** template:

- A slide structure based on a meeting presentation, which you can easily adapt to your needs. For more info on how to edit the template, please visit **Slidesgo School** or read our **FAQs**.
- An assortment of graphic resources that are suitable for use in the presentation can be found in the **alternative resources slide**.
- A **thanks slide**, which you must keep so that proper credits for our design are given.
- A **resources slide**, where you'll find links to all the elements used in the template.
- **Instructions for use**.
- Final slides with:
  - The **fonts and colors** used in the template.
  - A **selection of illustrations**. You can also customize and animate them as you wish with the online editor. Visit **Storyset** to find more.
  - More **infographic resources**, whose size and color can be edited.
  - Sets of **customizable icons** of the following themes: general, business, avatar, creative process, education, help & support, medical, nature, performing arts, SEO & marketing, and teamwork.

You can delete this slide when you're done editing the presentation.

MEETING OBJECTIVES

ENERGY SYSTEMS

Mercury is the smallest planet in the entire Solar System

SMARTER GRIDS

Despite being red, Mars is a cold place. It's full of iron oxide dust

ENERGY SUPPLY

Venus has a beautiful name and very high temperatures

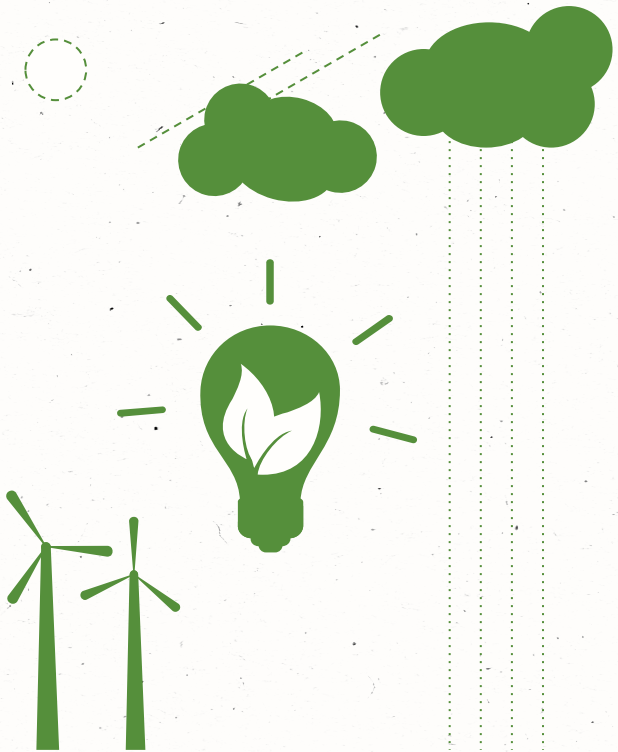
AGENDA

01
PROGRAM SCHEDULE
You can describe the topic here

02
PROGRAM TIMELINE
You can describe the topic here

03
STATUS REPORT
You can describe the topic here

04
UPCOMING EVENTS
You can describe the topic here



ABOUT THE PROJECT

You can give a brief description of the topic you want to talk about here. For example, if you want to talk about Mercury, you can say that it's the smallest planet in the entire System

BRIDGE INITIATIVE

The aim of BRIDGE is to share knowledge and expertise between projects, as well as to foster a dialogue on innovation and market regulation, primarily through different working groups. These working groups cover the areas:

- Data management
- Regulation
- Citizen and consumer engagement
- Business models

“This is a quote, words full of wisdom that someone important said and can make the reader get inspired.”

—SOMEONE FAMOUS

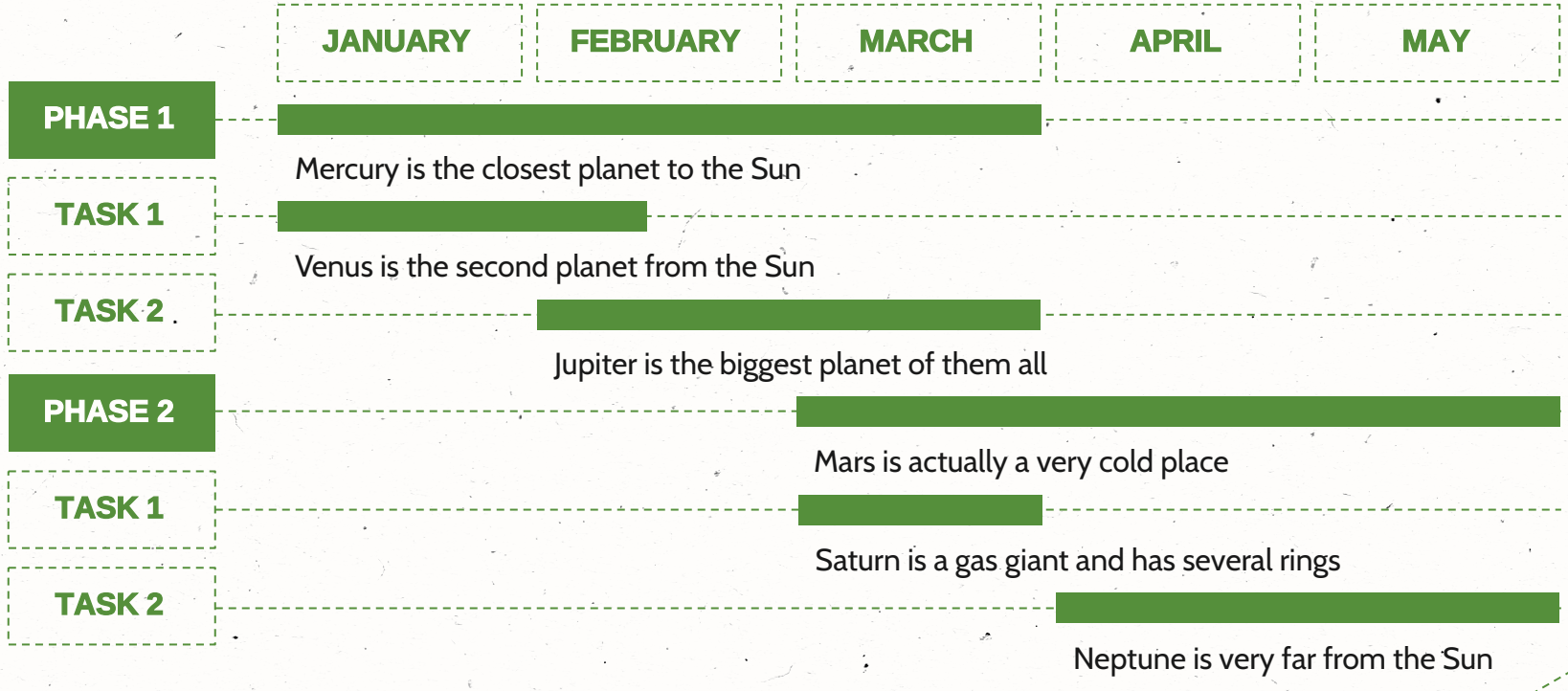
AWESOME WORDS

01

PROGRAM SCHEDULE

You can enter a subtitle here if you need it

PROGRAM SCHEDULE



CHECKLIST

	STRATEGY 1	STRATEGY 2	STRATEGY 3	STRATEGY 4
Describe your task 1 here	✗	✗	✓	✓
Describe your task 2 here	✓	✓	✓	✓
Describe your task 3 here	✓	✗	✓	✓
Describe your task 4 here	✗	✓	✓	✗
Describe your task 5 here	✗	✓	✓	✗

PROGRAM TIMELINE

Mercury is the closest planet to the Sun

MERCURY

01

Venus is the second planet from the Sun

VENUS

02

JUPITER

Jupiter is the biggest planet of them all

03

MARS

Mars is actually a very cold place

04

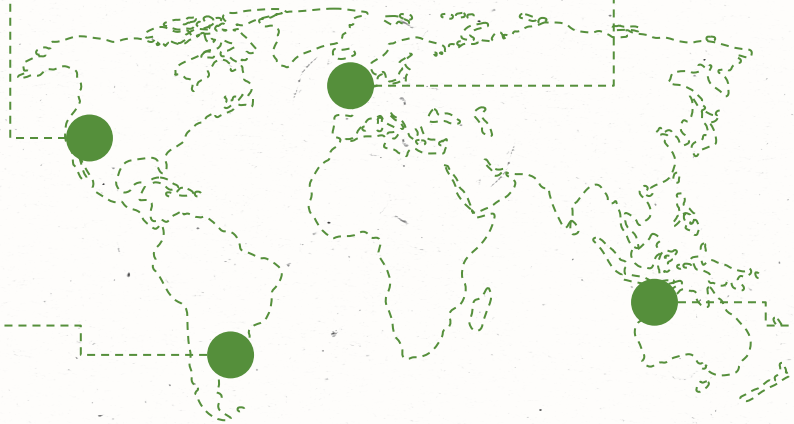
ENERGY SYSTEM WORLDWIDE

VENUS

Venus has a beautiful name

MERCURY

Mercury is the smallest planet



JUPITER

Jupiter is the biggest planet

MARS

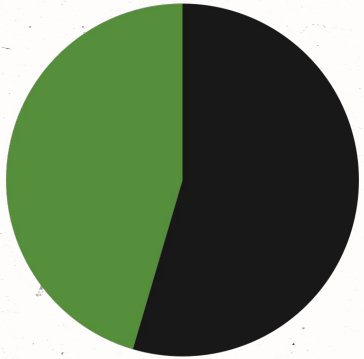
Mars is actually a very cold place


A PICTURE IS
WORTH A
THOUSAND WORDS

A PICTURE ALWAYS REINFORCES THE CONCEPT

Images reveal large amounts of data, so remember: use an image instead of a long text. Your audience will appreciate it

KPI DASHBOARD



Follow the link in the graph to modify its data and then paste the new one here. **For more info, click here**

$28.87

Avg. price per unit

30 DAYS

Avg. days on market

MERCURY

Mercury is the smallest planet

MARS

Despite being red, Mars is a cold place

45 SHOPS

Retailers

150,000

Big numbers catch your audience's attention

STATUS REPORT

	20%	50%	100%	
SCHEDULE		●		The project schedule is on track
RESOURCING			●	Resourcing is adequate
BUDGET			●	Project is within budget
RISKS	●			All project risks are under control
ISSUES		●		Project issues need to be solved
BENEFITS	●			Project benefits do not meet the expectations

RAID SUMMARY

RISKS

Venus is the second planet from the Sun

ASSUMPTIONS

Jupiter is the biggest planet of them all

ISSUES

Despite being red, Mars is a very cold place

DEPENDENCIES

Saturn is a gas giant and has several rings

WHERE WE ARE

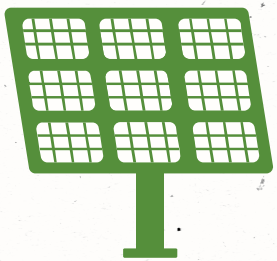
Mercury is the closest planet to the Sun and the smallest one in the Solar System. This planet's name has nothing to do with the liquid metal

WHERE WE WANT TO BE

Venus has a beautiful name and is the second planet from the Sun. It's terribly hot, even hotter than Mercury, and its atmosphere is extremely poisonous

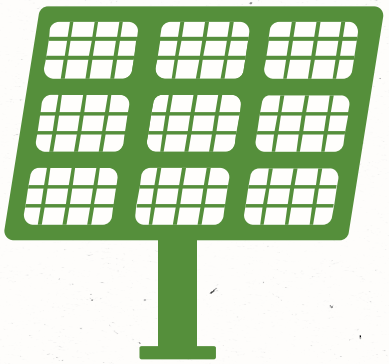
9h 55m 23s

is Jupiter's rotation period



333,000,000

Earths fit in the Sun's mass



386,000 km

is the distance between Earth and the Moon

UPCOMING EVENTS

14TH AUG.

Mercury is the closest planet to the Sun

23RD AUG.

Venus is the second planet from the Sun

8TH SEP.

Mars is actually a very cold place

28TH OCT.

Jupiter is the biggest planet of them all

16TH NOV.

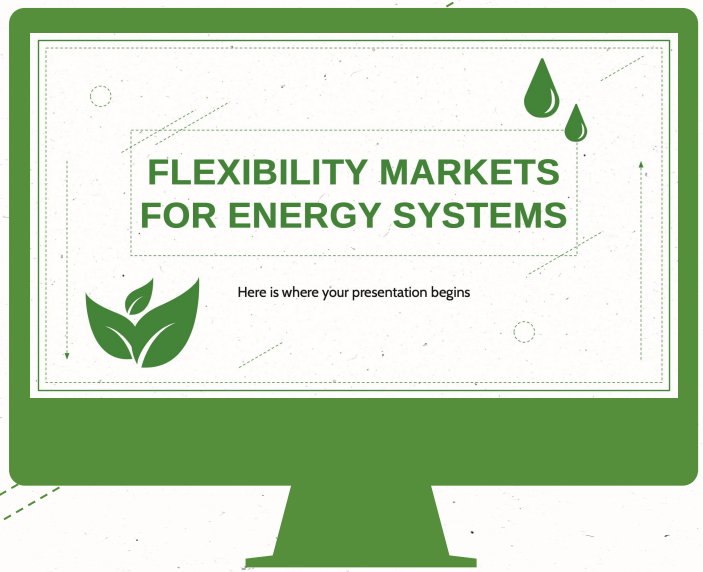
Saturn is made of hydrogen and helium

05TH DEC.

Neptune is very far from the Sun

ONLINE INFORMATION

You can replace the image on the screen with your own work. Just right-click on it and select **“Replace image”**



THANKS

Do you have any questions?
youremail@freepik.com
+91 620 421 838
yourcompany.com

RENEWABLE ENERGY

ALTERNATIVE RESOURCES

Here's an assortment of alternative resources whose style fits that of this template

ALTERNATIVE RESOURCES

Here's an assortment of alternative resources whose style fits that of this template

VECTOR

- Flat design ecology concept with natural elements

RESOURCES

Did you like the resources on this template? Get them for free at our other websites:

VECTOR

- Flat design ecology concept with natural elements
- Hand drawn ecology concept with natural elements

PHOTO

- Front view of wind turbines with copy space
- Field with lots of solar panels

ICON

- Renewable Energy

Instructions for use

In order to use this template, you must credit **Slidesgo** by keeping the **Thanks** slide.

You are allowed to:

- Modify this template.
- Use it for both personal and commercial projects.

You are not allowed to:

- Sublicense, sell or rent any of Slidesgo Content (or a modified version of Slidesgo Content).
- Distribute Slidesgo Content unless it has been expressly authorized by Slidesgo.
- Include Slidesgo Content in an online or offline database or file.
- Offer Slidesgo templates (or modified versions of Slidesgo templates) for download.
- Acquire the copyright of Slidesgo Content.

For more information about editing slides, please read our FAQs or visit Slidesgo School:

https://slidesgo.com/faqs and https://slidesgo.com/slidesgo-school

Instructions for use (premium users)

In order to use this template, you must be a Premium user on **Slidesgo**.

You are allowed to:

- Modify this template.
- Use it for both personal and commercial purposes.
- Hide or delete the “Thanks” slide and the mention to Slidesgo in the credits.
- Share this template in an editable format with people who are not part of your team.

You are not allowed to:

- Sublicense, sell or rent this Slidesgo Template (or a modified version of this Slidesgo Template).
- Distribute this Slidesgo Template (or a modified version of this Slidesgo Template) or include it in a database or in any other product or service that offers downloadable images, icons or presentations that may be subject to distribution or resale.
- Use any of the elements that are part of this Slidesgo Template in an isolated and separated way from this Template.
- Register any of the elements that are part of this template as a trademark or logo, or register it as a work in an intellectual property registry or similar.

For more information about editing slides, please read our FAQs or visit Slidesgo School:

https://slidesgo.com/faqs and https://slidesgo.com/slidesgo-school

Fonts & colors used

This presentation has been made using the following fonts:

Ramabhadra
(https://fonts.google.com/specimen/Ramabhadra)

Cabin
(https://fonts.google.com/specimen/Cabin)

#558f3b

#fffefc

#181818

Storyset

Create your Story with our illustrated concepts. Choose the style you like the most, edit its colors, pick the background and layers you want to show and bring them to life with the animator panel! It will boost your presentation. Check out How it Works.

Pana

Amico

Bro

Rafiki

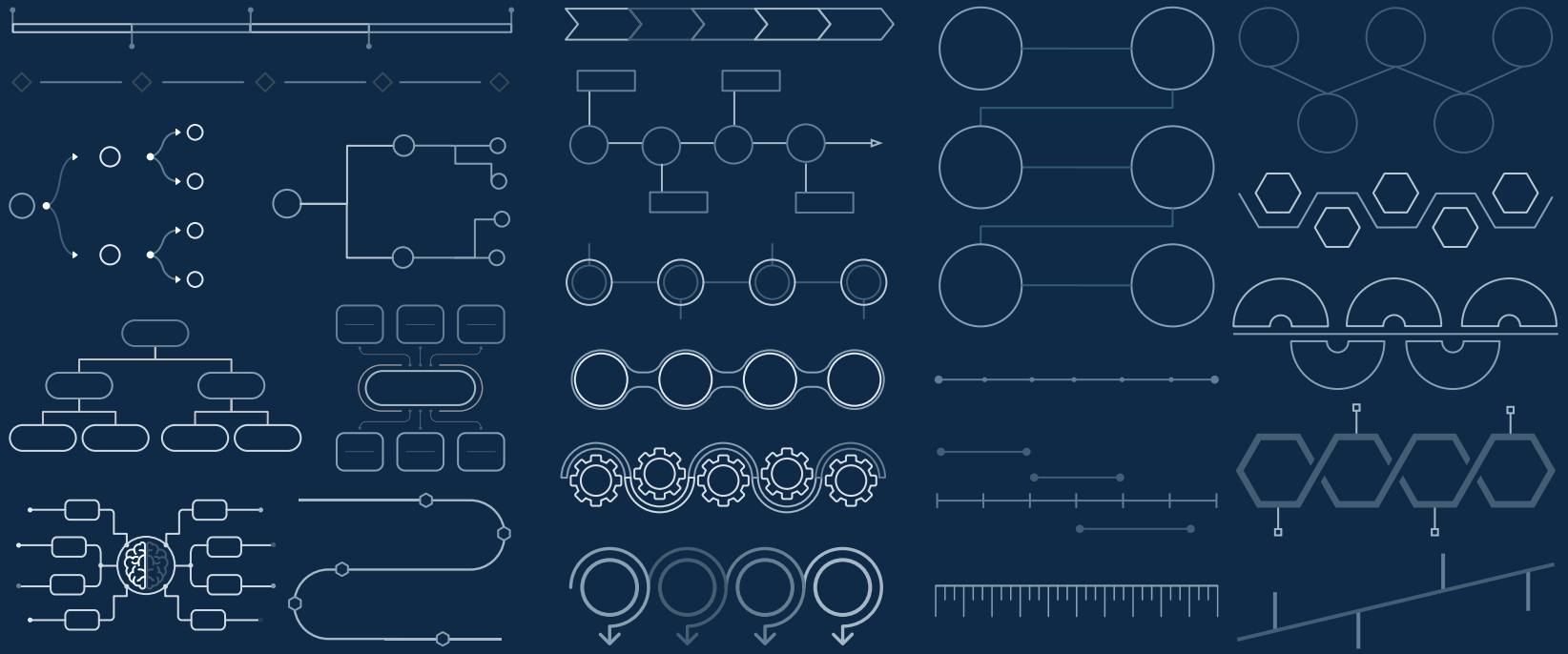
Cuate

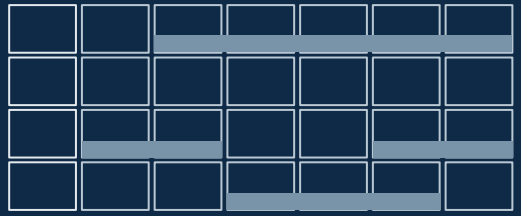
Use our editable graphic resources...

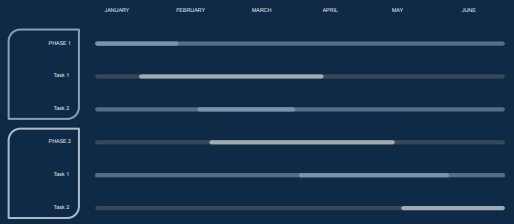
You can easily resize these resources without losing quality. To change the color, just ungroup the resource and click on the object you want to change. Then, click on the paint bucket and select the color you want. Group the resource again when you're done. You can also look for more infographics on Slidesgo.

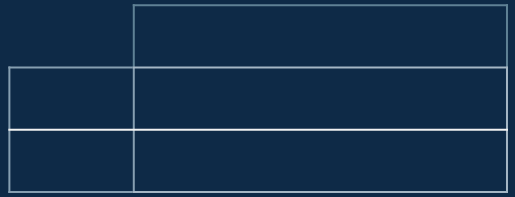


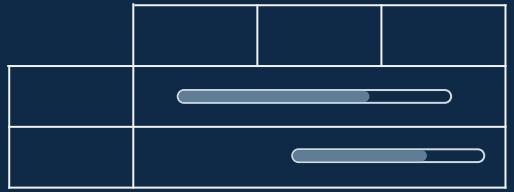


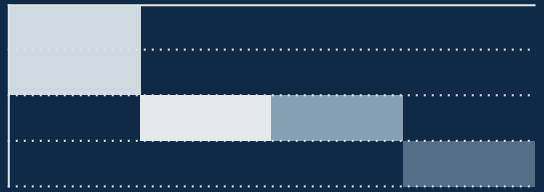



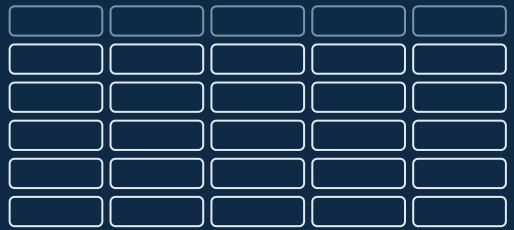


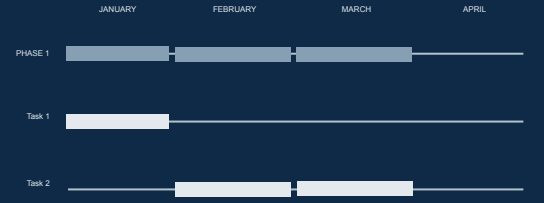

JANUARY
FEBRUARY
MARCH
APRIL
MAY
JUNE
PHASE 1
Task 1
Task 2
PHASE 2
Task 1
Task 2

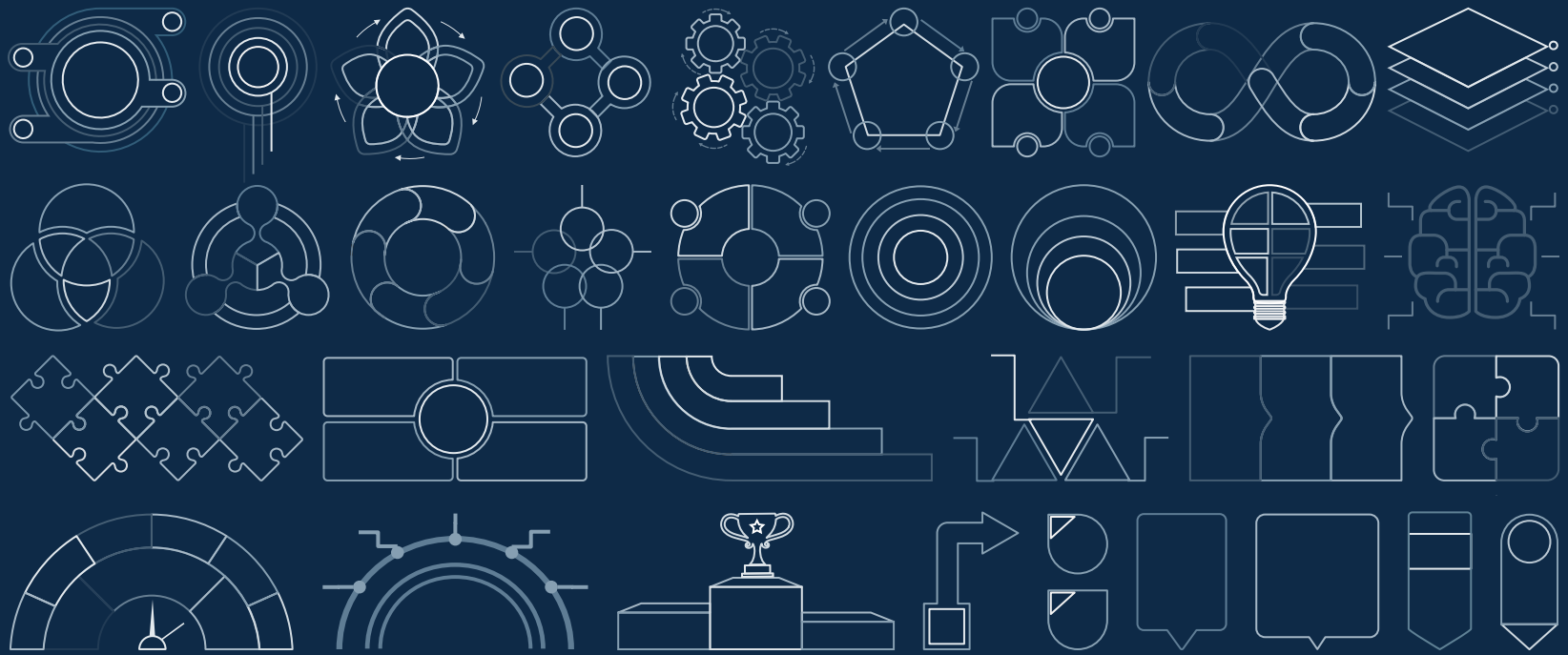


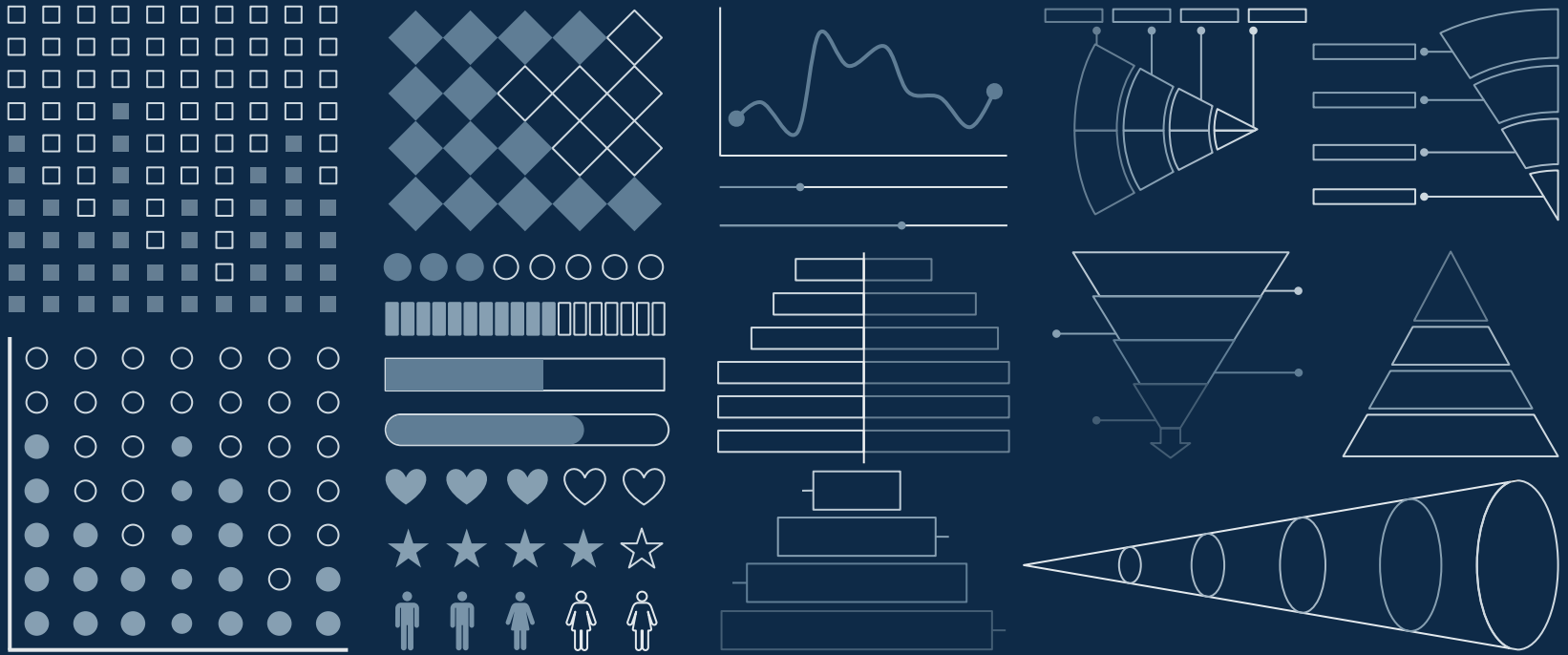







JANUARY
FEBRUARY
MARCH
APRIL
PHASE 1
Task 1
Task 2





...and our sets of editable icons

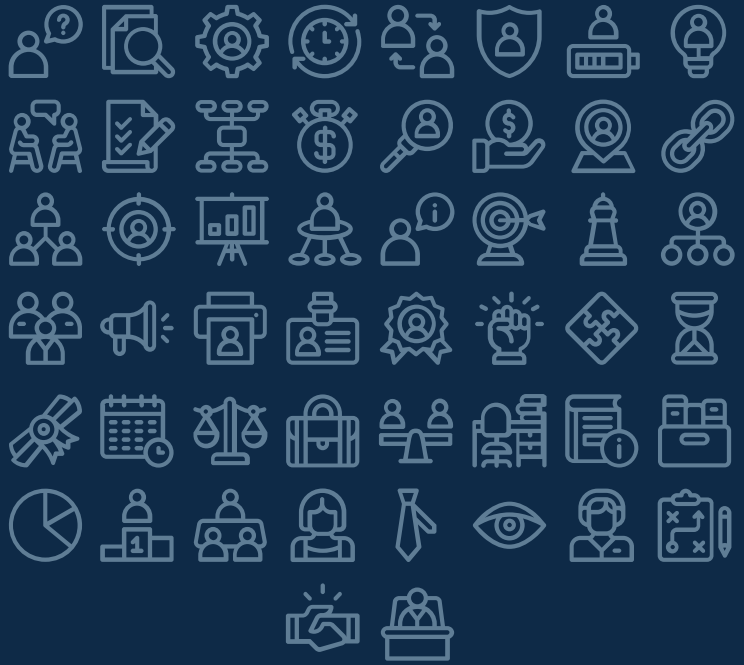
You can resize these icons without losing quality.
You can change the stroke and fill color; just select the icon and click on the paint bucket/pen.
In Google Slides, you can also use Flaticon's extension, allowing you to customize and add even more icons.

Educational Icons

Medical Icons

Business Icons

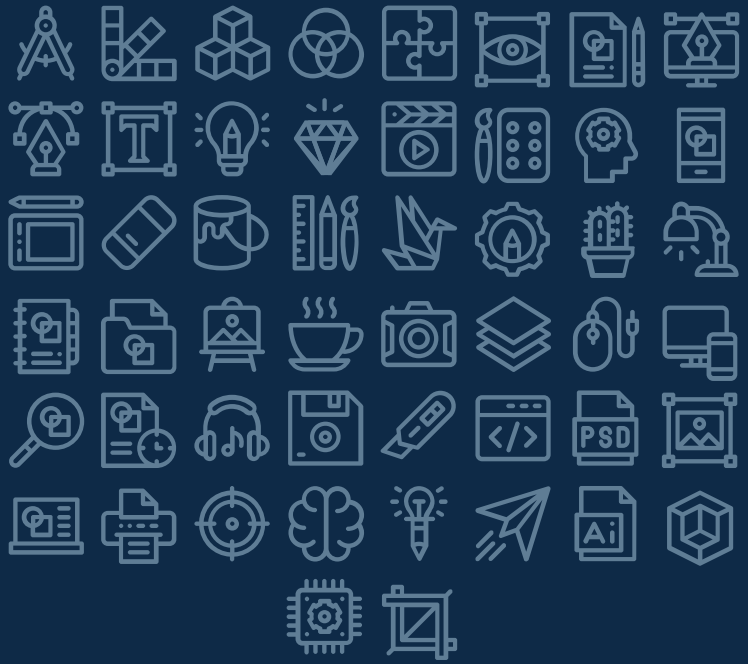
Teamwork Icons



Help & Support Icons

Avatar Icons

Creative Process Icons



Performing Arts Icons

Nature Icons

SEO & Marketing Icons


slidesgo