📡 Origens da Internet

- **Período**: Década de 1960 e início dos anos 1970.
- **Contexto histórico**: *Guerra Fria*, marcada por forte rivalidade entre Estados Unidos e União Soviética.
- **Motivação principal**: Garantir a comunicação entre centros militares e de pesquisa mesmo em caso de ataque nuclear. A ideia era que a rede fosse *descentralizada* e resistente a falhas.

🖥️ ARPANET: O embrião da Internet

- **Criada pela**: ARPA (*Advanced Research Projects Agency*), agência do Departamento de Defesa dos EUA.
- **Inovação técnica**: Utilização de **comutação de pacotes** (*packet switching*).
    - As mensagens eram divididas em pequenos blocos (pacotes).
    - Cada pacote podia seguir rotas diferentes pela rede até chegar ao destino.
    - Ao chegar, os pacotes eram remontados para formar a mensagem original.
- **Vantagem**: Mesmo que parte da rede fosse danificada, os dados poderiam circular por outros caminhos – um conceito chave na resiliência das redes modernas.

🌐 Nascimento da WWW (World Wide Web)

Embora não esteja no seu texto, é comum confundir Internet com WWW. Vale destacar:

- A **Internet** é a infraestrutura (rede de redes).
- A **WWW** foi criada posteriormente, em **1989**, por **Tim Berners-Lee**, no CERN.
    - Ela trouxe uma forma mais acessível de navegar pela Internet, por meio de **hipertextos**, **navegadores** e **páginas web**.
    - Foi o que popularizou a Internet para o público geral nos anos 1990.

🌐 HTTP – HyperText Transfer Protocol

📌 Definição

É um protocolo de comunicação usado na transferência de páginas web (HTML) entre um cliente (navegador) e um servidor na Internet. As URLs que começam com `http://` indicam que utilizam esse protocolo.

🔄 Funcionamento do HTTP

Características principais:

- **Request-Response**: Cliente faz a requisição, servidor responde.
- **Conexão temporária**: O cliente abre uma conexão, envia o request, recebe o response e fecha a conexão.
- **Stateless**: O servidor não armazena informações entre requisições – cada requisição é independente.
- **Identificação por URI**: Cada recurso é identificado por um *Uniform Resource Identifier*.
- **URL é um tipo de URI**: Especifica não só o recurso, mas também o protocolo e endereço (ex: `http`, `ftp`, `mailto`).

✉️ Estrutura de uma Mensagem HTTP

1. Linha inicial

- Para **request**: `Método URI Versão`
    - Ex: `GET /hello/HelloApp?userid=Joao HTTP/1.1`
- Para **response**: `Versão Código Mensagem`
    - Ex: `HTTP/1.1 200 OK`

2. Headers

- Informações sobre a mensagem: tipo de conteúdo, tamanho, codificação etc.

3. Linha em branco

4. Corpo da mensagem

- Presente apenas quando há envio de dados (ex: em um POST).

⚙️ Métodos HTTP mais comuns

Método	Função Principal
GET	Solicita dados do servidor (parâmetros enviados na URL).
POST	Envia dados ao servidor (corpo da requisição).
HEAD	Solicita apenas os headers, sem o corpo da resposta.
PUT	Cria ou atualiza um recurso no servidor.
DELETE	Solicita que um recurso seja removido.
OPTIONS	Verifica quais métodos o servidor suporta.
TRACE	Loopback de teste da requisição enviada.
CONNECT	Usado para comunicação via proxy (ex: HTTPS via túnel).

🧩 Passagem de Parâmetros

Via método GET:

http://localhost:1234/hello/HelloServlet?userid=10&nome=Ze

- Dados visíveis na URL.
- Separação por ? e &.
- Pode ser salvo, cacheado ou compartilhado.
- **Menos seguro**.

Via método POST:

- Dados enviados no **corpo da requisição**.
- **Mais seguro** (não visível na URL).
- Usado para formulários grandes ou dados sensíveis.

🔐 Segurança

- **GET**: Pode ser interceptado ou salvo em histórico.
- **POST**: Mais apropriado para dados sensíveis.
- Ambos não são criptografados *por si só* — para segurança real, use **HTTPS**.

🌐 Servidores Web

📌 O que é um servidor web?

Um **servidor web** é um software responsável por receber, processar e responder a requisições HTTP feitas por clientes (geralmente navegadores). Ele se comunica usando o protocolo **TCP/IP**, monitorando conexões em portas específicas — geralmente, a porta **80** para HTTP e **443** para HTTPS.

⚙️ Funcionamento básico

1. O servidor web **escuta** uma porta (geralmente 80).
2. Ao receber uma requisição HTTP de um cliente (navegador), ele:
    - Analisa o conteúdo da requisição.
    - Processa os dados (com ou sem ajuda de aplicações como Servlets ou scripts).
    - Envia uma **resposta HTTP**, normalmente com uma **página HTML**.

Esse processo é ilustrado pela **Figura 5** mencionada no seu material.

🗂️ Estrutura interna do servidor

- Os servidores mantêm uma **estrutura de diretórios configurada** para armazenar os arquivos acessíveis ao cliente (HTML, CSS, imagens, scripts etc).
- O **acesso externo** a esses arquivos é feito via **URL**, que aponta para o diretório virtual exposto pelo servidor.

Exemplo de acesso:

http://localhost:8080/meuapp/index.html

> Aqui, o navegador está acessando o arquivo `index.html` dentro da aplicação `meuapp`, hospedada localmente na porta 8080.

🧪 Ambiente de teste local

Não é necessário ter um servidor físico externo. Durante o desenvolvimento, você pode:

- Instalar um **servidor local** como **Tomcat** ou **GlassFish** no seu próprio computador.
- O cliente (navegador) e o servidor rodam **na mesma máquina**, permitindo testes sem infraestrutura externa.

Este cenário é ilustrado na **Figura 7**, que representa uma aplicação sendo acessada e servida localmente.

🧰 Exemplos de servidores web

Servidor	Finalidade
Apache HTTP Server	Servir arquivos estáticos (HTML, CSS) e atuar como proxy reverso.
Tomcat	Servidor web e container de Servlets/JSP (Java).
GlassFish	Servidor de aplicações Java EE (mais robusto que o Tomcat).
NGINX	Muito usado para alta performance, balanceamento de carga.

1.4 📱 Aplicações Web

As **aplicações web** são sistemas acessados por navegadores, que se comunicam com servidores web para renderizar páginas ou fornecer serviços. Elas podem ser classificadas em dois tipos:

🔹 Tipos de Aplicações Web

1. **Orientadas à apresentação**
    - Foco: Exibição de conteúdo ao usuário.
    - Tecnologias: HTML, XML, DHTML, XHTML etc.
2. **Orientadas a serviços**
    - Foco: Exposição e consumo de serviços por meio de APIs (como Web Services).
    - Exemplo: RESTful APIs, SOAP Web Services.

☕ Componentes Web na Plataforma Java

As aplicações web em Java são compostas por:

- **Servlets**
  Classes Java que processam requisições HTTP de forma programática.
- **Páginas JSP (JavaServer Pages)**
  Páginas HTML com trechos de código Java que são convertidas em Servlets pelo container.
- **Web Services**
  Interfaces para permitir que aplicações troquem dados e serviços via rede.

🔁 Ciclo de vida de uma Requisição Web (Figura 8)

1. O **cliente envia** uma requisição HTTP.

2. O **servidor** converte para um objeto `HttpServletRequest`.
3. O **componente web** (Servlet ou JSP) manipula a requisição.
4. O componente pode interagir com **JavaBeans** (modelos) e banco de dados.
5. Um objeto `HttpServletResponse` é gerado com o conteúdo dinâmico.
6. O servidor converte a resposta Java para uma **resposta HTTP** e a envia ao navegador.

🧩 Servlets vs JSP

Característica	Servlets	JSP
Tipo de componente	Classe Java	Página HTML com Java embutido
Uso ideal	Controlador, lógica de negócio	Exibição de conteúdo (view)
Conversão	Não precisa ser convertido	Convertido em Servlet pelo container

> Observação: Frameworks como **JSF** e **Struts** são implementados com base em Servlets.

📦 Web Container (ou Servlet Container)

Todos os componentes web rodam dentro de um **web container**, como o Tomcat ou GlassFish. Ele fornece:

- Gerenciamento do ciclo de vida dos componentes (Servlets, JSPs);
- Encaminhamento de requisições;

- Segurança;
- Gerenciamento de concorrência;
- Suporte a transações;
- Integração com serviços de e-mail etc.

1.4.1 📁 Módulo Web

Um **módulo web** é a **menor unidade de uma aplicação web que pode ser implantada (deployed)**. Pode ser:

- Um **diretório estruturado**, ou
- Um **arquivo .WAR (Web Application Archive)**.

🗂️ Estrutura do Módulo Web

```
meuapp/
├── index.jsp
├── imagens/
│   └── logo.png
├── css/
│   └── estilo.css
└── WEB-INF/
    ├── web.xml          ← Descritor de implantação (opcional em aplicações modernas)
    ├── lib/             ← Bibliotecas externas (.jar)
    └── classes/         ← Classes compiladas, Servlets, JavaBeans
```

Observações:

- O diretório `WEB-INF` **não é acessível diretamente via URL**.
- O `web.xml` é o **descritor de implantação**, usado para registrar Servlets, filtros, listeners etc. Atualmente, com anotações Java (como `@WebServlet`), ele pode ser omitido.
- O módulo pode ser implantado:
    - Desempacotado (estrutura de pastas); ou
    - Empacotado como `.WAR` para distribuição e deploy em servidores como Tomcat, GlassFish, WildFly.

Esse trecho apresenta uma explicação clara e didática sobre **aplicações web Java**, em especial sobre:

✅ Pontos Positivos:

1. **Divisão entre aplicações orientadas à apresentação e aos serviços** — facilita a compreensão dos dois principais tipos de aplicações web.

2. **Fluxo completo de requisições HTTP** — do cliente até a resposta, incluindo o uso de `HttpServletRequest` e `HttpServletResponse`.

3. **Explicação dos componentes web (Servlets, JSPs)** — com destaque para suas funções típicas.

4. **Definição do módulo web e estrutura de diretórios** — inclui a importância do `WEB-INF`, `web.xml`, e da estrutura padrão de pacotes.

5. **Explicação do contexto da aplicação** — com exemplos claros, como a configuração de contextos no Tomcat com arquivos `.xml`.

6. **Configuração completa do `web.xml`** — cobre desde parâmetros de contexto e inicialização até páginas de erro e arquivos de boas-vindas.

⚠️ Pontos a Melhorar:

1. **Nomenclatura inconsistente:**

    - As tags `<Servlet>` e `<Servlet-mapping>` estão com letra maiúscula, mas o padrão XML (e o especificado pela especificação do Servlet) usa minúsculas: `<servlet>`, `<servlet-mapping>`, etc.

    - Isso pode confundir quem tentar copiar o conteúdo e usar diretamente, pois XML é sensível a maiúsculas e minúsculas.

Path incorreto na tag `<Context>`:

<Context path="\app" docBase="C:/Aplicacao/web" />

O valor de `path="\app"` contém uma **barra invertida (`\`)**, que não é recomendada. Deve ser:

<Context path="/app" docBase="C:/Aplicacao/web" />

2.
3. **Atenção à formatação dos exemplos XML:**

Há um espaço desnecessário dentro da tag `<location>` em:

<location>/erro.jsp</ location>

Deve ser:

<location>/erro.jsp</location>

-

4. **Faltou referência à Servlet API moderna:**
    - Desde o **Servlet 3.0**, o `web.xml` é opcional em muitos casos, e o uso de anotações como `@WebServlet`, `@WebFilter` e `@WebListener` é a forma preferida de configuração. Isso deveria ser mais enfatizado.

✅ Sugestão de Reescrita de um trecho (com correções):

```
<servlet>
  <servlet-name>Cadastrar</servlet-name>
  <servlet-class>com.ead.servlet.Cadastrar</servlet-class>
  <init-param>
    <param-name>tipo</param-name>
    <param-value>2</param-value>
  </init-param>
  <load-on-startup>1</load-on-startup>
</servlet>


<servlet-mapping>
  <servlet-name>Cadastrar</servlet-name>
  <url-pattern>/Cadastrar</url-pattern>
</servlet-mapping>
```

Esse conteúdo explica, passo a passo, como criar uma aplicação Java Web no Eclipse usando a perspectiva **Java EE** e o recurso **Dynamic Web Project**, incluindo a configuração do servidor **Tomcat 7.0**, a geração do arquivo `web.xml` e a criação e execução de um **Servlet**.

Aqui vai um resumo claro e direto com os **principais pontos**:

✅ Criação de um projeto Java Web no Eclipse

1. **Perspectiva Java EE** ativada.
2. Criar novo projeto: `File > New > Dynamic Web Project`.
3. **Definir nome do projeto**.
4. Selecionar o servidor com **"New Runtime..."**.
    - Escolher **Apache Tomcat v7.0**.
    - Indicar pasta de instalação e versão do JRE.
5. Selecionar **versão do módulo web (Servlet 3.0)**.
6. Marcar a opção **“Generate web.xml deployment descriptor”**.
7. Finalizar com **Finish**.

✅ Criação e execução de um Servlet

1. Clique com botão direito no projeto → **New > Servlet**.
2. Informar **pacote e nome da classe**.
3. Definir nome do Servlet e **URL Pattern** (ex: `/exemplo`).
4. Eclipse gera a classe com métodos `doGet()` e `doPost()`.

Inserir código no `doGet()`:

protected void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {

```
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();
    out.println("<h1>Primeiro Servlet funcionando!</h1>");
}
```

5.
6. Executar: clicar na seta ao lado do botão Run → **"Run on Server"**.
7. Resultado exibido no **browser embutido do Eclipse**.
8. Para usar outro navegador, configurar em: `Window > Web Browser`.