A decorative pattern of overlapping green squares and rectangles of various shades, creating a geometric, pixelated effect on the right side of the slide.

Metodología de Sistemas II

Clase 3

Manejo de dependencias

¿Qué son las dependencias?

Definición práctica

¿Qué son las dependencias?

En el contexto del desarrollo de software, una dependencia es cualquier librería o módulo externo que un proyecto necesita para funcionar.

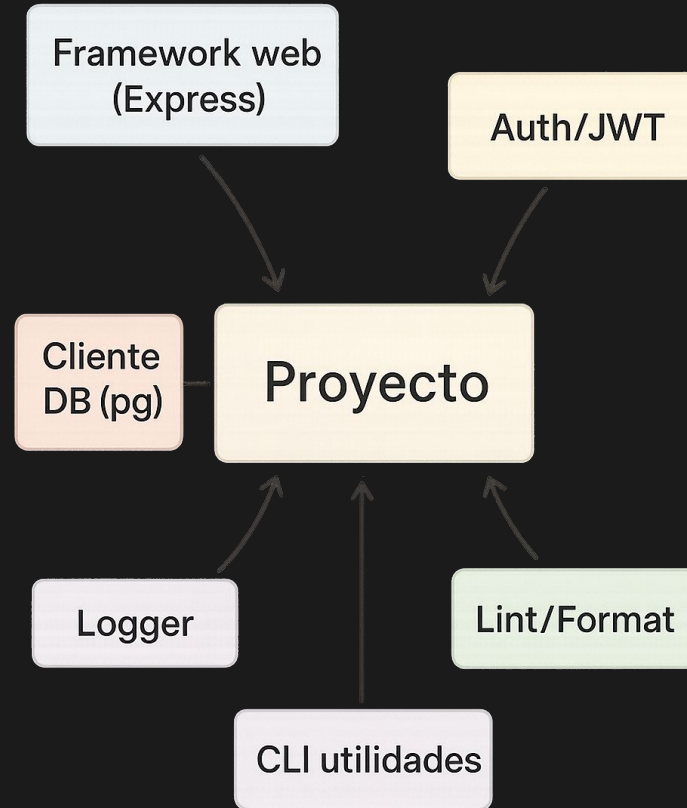
Ejemplos:

- Una web app que usa un framework HTTP (Express en Node.js o Flask en Python). Ese framework es una dependencia.
- Para testear se puede usar Jest (JS) o pytest (Python), esas herramientas también son dependencias.

¿Qué son las dependencias?

- **Idea clave:** un proyecto rara vez vive “solo”; se apoya en un ecosistema de paquetes o librerías que resuelven muchos problemas comunes (parsing, web, autenticación, logs, pruebas, linting, etc.).

¿Qué son las dependencias?



Tipos de dependencias: Producción vs Desarrollo

En cualquier stack (JavaScript o Python) conviene separar para qué se usa cada dependencia:

- Dependencias de Producción (runtime): Necesarias cuando la aplicación corre en un entorno real, es decir, brinda servicios a usuarios reales (servidor, contenedor Docker, etc.).

Ejemplos:

- JS: `express`, `pg` (cliente Postgres), `jsonwebtoken`.
- Python: `flask`, `sqlalchemy`, `requests`.

Si estas dependencias faltan, la app no arranca o falla en tiempo de ejecución.

Tipos de dependencias: Producción vs Desarrollo

- Dependencias de Desarrollo (dev): Se usan solo para desarrollar, probar y asegurar calidad. No son necesarias cuando la app ya está desplegada, es decir, en producción.

Ejemplos:

- JS: `jest`, `eslint`, `prettier`, `vite` (si no se usa en producción).
- Python: `pytest`, `black`, `flake8`, `mypy`.

Si estas faltan en el servidor de producción, la app puede seguir funcionando sin problemas.

Tipos de dependencias: Producción vs Desarrollo

¿Por qué separarlas?

- Tamaño y performance: la imagen Docker/servidor es más liviana si no incluye herramientas de desarrollo.
- Seguridad y superficie de ataque: menos paquetes en producción, menos riesgo.
- Claridad del equipo: cualquiera que lee el manifiesto de dependencias entiende qué es crítico para correr y qué es solo para trabajar mejor.

Gestores de paquetes

Los gestores de paquetes (o dependencias) (npm/yarn/pnpm en JavaScript; pip/poetry en Python) abordan tres desafíos fundamentales:

1. Versionado
2. Instalación automática
3. Resolución de conflictos

1. Versionado

Cuando se agrega una dependencia, no solo importa "qué librería", sino también qué versión exacta de esa librería.

- Las versiones suelen seguir SemVer (Semantic Versioning):
MAJOR.MINOR.PATCH (por ejemplo, 2.4.1).
 - PATCH: correcciones de bugs sin romper compatibilidad.
 - MINOR: nuevas funciones compatibles.
 - MAJOR: cambios incompatibles (pueden romper tu código).

Gestores de paquetes

En JavaScript, `package.json` y su lock file (`package-lock.json` o `yarn.lock`) fijan versiones resueltas para instalaciones reproducibles.

En Python, `requirements.txt` (pip) o `pyproject.toml` + `poetry.lock` (poetry) cumplen el mismo rol: asegurar que todos en el equipo y los servidores instalen exactamente lo mismo.

Beneficio: reproducibilidad. Si hoy todo funciona, el día de mañana se puede reinstalar el proyecto y tener el mismo conjunto de paquetes, evitando "a mí me anda" o "ayer andaba".

2. Instalación automática

Antes, integrar una librería implicaba descargar archivos manualmente y acomodarlos en el proyecto. Ahora los gestores:

- Descargan la librería automáticamente desde un registro central (npm registry / PyPI).
- Resuelven dependencias transitivas (dependencias de las dependencias).
- Guardan el estado en archivos de manifiesto y lock para poder reinstalar todo con un solo comando (`npm ci`, `yarn install`, `pip install -r requirements.txt`, `poetry install`).

Beneficio: **productividad** y consistencia en todo el equipo.

3. Resolución de conflictos

Los proyectos pueden depender de muchas librerías, y cada una puede requerir otra versión de otra librería. Los gestores resuelven:

- Determinan qué versión final de cada paquete se debe instalar.
- Evitan (en lo posible) que dos paquetes colisionen por solicitar versiones incompatibles.
- En caso de conflicto irresoluble, lo informan para fijar una versión, actualizar, o excluir una combinación.

Beneficio: **estabilidad** del ecosistema del proyecto. Es posible crecer en funcionalidades, sin que cada incorporación implique resolución de versiones.

Gestores de paquetes. Resumen

- Un proyecto es un grafo de dependencias: la app está en la raíz y de ella salen aristas (flechas) hacia librerías; esas librerías pueden depender de otras, formando un árbol/grafó.
- Determinismo > Azar: los lock files y los manifests (`package.json`, `pyproject.toml`, `requirements.txt`) son la receta exacta para reconstruir ese grafo siempre igual.
- Aislamiento: los entornos virtuales (`venv`) y el uso de versiones de Node controladas (`nvm`) evitan que un proyecto “ensucie” a otro.
- Seguridad y mantenimiento: menos es más. No agregar librerías “por las dudas”; mantener actualizado lo necesario y auditar vulnerabilidades (`npm audit` / `pip-audit` / `dependabot`, etc.).

Gestores de paquetes. Resumen

- “Manejo de dependencias” no es solo saber un comando; es gobernar la complejidad de terceros en sus proyecto: elegir bien, fijar versiones, aislar entornos, auditar riesgos y documentar. Quien domina esto, evita muchos problemas y acelera el desarrollo.

Gestión de dependencias en JavaScript (npm / yarn)

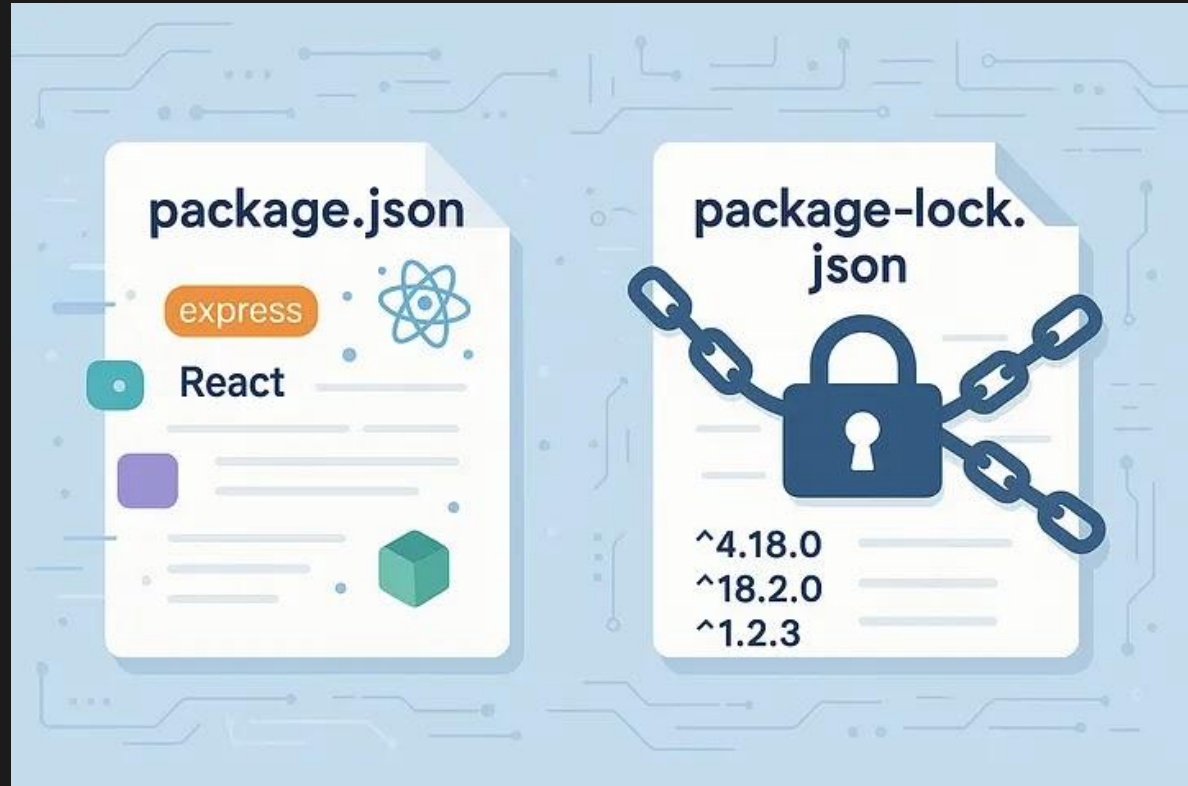
- Archivos clave y roles
- Dependencias: producción vs desarrollo en JS
- SemVer y rangos de versión
- Instalación y reproducibilidad
- Scripts de npm/yarn
- nvm y versión de Node
- Seguridad y mantenimiento

Archivos clave y roles

- `package.json`: manifiesto del proyecto.
Define: Nombre/versión del proyecto, scripts, dependencias (`dependencies`) y dependencias de desarrollo (`devDependencies`), campo `engines` (versión de Node recomendada), `type` (`commonjs/module`), etc.
- Lockfile (asegura instalaciones reproducibles):
 - `npm` → `package-lock.json`
 - `yarn` → `yarn.lock`

Guardan versiones exactas resueltas (incluyendo dependencias transitivas) e integrity hashes para verificar descargas.

package.json vs package-lock.json



Dependencias: prod vs dev JS

`dependencies` (runtime): lo que la app necesita para correr (ej.: `express`, `pg`, `jsonwebtoken`).

`devDependencies` (dev): herramientas de desarrollo (ej.: `jest`, `eslint`, `prettier`, `vite` si no se usa en prod).

- Cómo se instalan:
 - `npm`:
 - `prod: npm install express`
 - `dev: npm install -D jest` (equivalente a `--save-dev`)
 - `yarn`:
 - `prod: yarn add express`
 - `dev: yarn add -D jest`

Dependencias: prod vs dev JS

Buenas prácticas

- Mantener claro qué va en prod vs dev. Esto impacta en el tamaño de imagen Docker, seguridad y performance.
- No “commitear” `node_modules/` (salvo entornos muy específicos). El lockfile sí se versiona.

SemVer y rangos de versión

Lo que realmente pasa con ^ y ~

- Formato: MAJOR.MINOR.PATCH (ej.: 2.4.1).
- Prefijos comunes en package.json:
 - ^2.4.1 → permite MINOR y PATCH ($\geq 2.4.1$ < 3.0.0).
 - ~2.4.1 → permite solo PATCH ($\geq 2.4.1$ < 2.5.0).
 - 2.4.1 exacto → sin rangos (congelada).

Recomendación:

- En proyectos educativos: usar ^ para experimentar y aprender a actualizar.
- En producción: congelar más (usar ~ o exactas) y confiar en **lockfile** + `npm ci` para reproducibilidad.

Instalación y reproducibilidad

- npm
 - `npm install` → instala según rangos y actualiza lock si hace falta.
 - `npm ci` → instalación limpia y reproducible basada exactamente en `package-lock.json` (falla si no coincide). Ideal para CI/CD.
- yarn
 - `yarn install` → usa `yarn.lock`. Con `--frozen-lockfile` evita modificarlo (ideal para CI).
- Caso práctico (CI/CD):
 - En pipelines: preferir `npm ci` (npm) o `yarn install --frozen-lockfile` (yarn).

Scripts de npm/yarn

Es el “pegamento” que une el flujo de trabajo. En package.json, el tag `scripts` define tareas:

```
{  
  "scripts": {  
    "dev": "node src/index.js",  
    "test": "jest --runInBand",  
    "lint": "eslint .",  
    "build": "tsc -p tsconfig.json",  
    "start": "node dist/index.js"  
  }  
}
```

Se ejecutan con:

- `npm` → `npm run dev` / `npm test` / `npm run build`
- `yarn` → `yarn dev` / `yarn test` / `yarn build`

nvm y versión de Node

Evita el “en mi máquina funciona” o “a mí me anda”. nvm permite tener múltiples versiones de Node.

- Archivo .nvmrc con la versión recomendada (p. ej., 20.15.0).
- Flujo típico para el equipo:
 1. nvm install (lee .nvmrc).
 2. nvm use.
 3. Instalar dependencias con la versión correcta de Node.

Alternativas: volta, o Docker para encapsular entorno.

Seguridad y mantenimiento

- Auditoría:
 - npm: `npm audit`, `npm audit fix`
 - yarn: `yarn npm audit (v3)` o herramientas externas.
- Actualizaciones:
 - npm: `npm outdated`, `npm update`, o `npx npm-check-updates -u + npm install`
 - yarn: `yarn outdated`, `yarn upgrade`, `yarn upgrade-interactive (v1)`
- Buenas prácticas:
 - Revisar changelogs al subir MAJOR.
 - Mantener lockfile actualizado y committeado.
 - Evitar paquetes innecesarios.

Errores comunes

- Borrar el lockfile → se pierde reproducibilidad.
 - No hacerlo.
- Meter todo en dependencies → imágenes pesadas y más superficie de ataque.
 - Separar dev/prod.
- No alinear versión de Node → builds inconsistentes.
 - Usar nvm o Docker.
- Actualizar a MAJOR “a ciegas” → romper producción.
 - Revisar changelog y probar en una rama nueva.

Gestión de dependencias en Python (pip / poetry)

- pip – Instalador estándar
- virtualenv / venv – Entornos virtuales
- poetry – Una alternativa moderna

pip – Instalador estándar

pip (Python Package Installer) es la herramienta oficial y más utilizada para instalar librerías en Python.

- Ejemplo básico:
 - `pip install flask`

Esto descarga Flask y sus dependencias desde PyPI (Python Package Index).

Para registrar las dependencias de un proyecto se utiliza `requirements.txt`, un archivo de texto plano con una lista de paquetes y versiones que se genera con:

- `pip freeze > requirements.txt`

pip – Instalador estándar

Ejemplo:

```
flask==2.3.2  
sqlalchemy==2.0.20  
pytest==7.4.0
```

Con esto, cualquiera puede reconstruir el mismo entorno con:

- `pip install -r requirements.txt`

Idea: `pip + requirements.txt` permite compartir y reproducir entornos de manera sencilla.

virtualenv / venv – Entornos virtuales

Python permite crear entornos virtuales para evitar conflictos entre proyectos.

¿Qué problema que resuelven?: si un proyecto requiere Django 3.2 y otro Django 4.1, no podés tener ambas versiones instaladas globalmente sin que colisionen.

Solución: aislar cada proyecto en su propio entorno virtual.

virtualenv/venv – Entornos virtuales

¿Cómo se usan?

```
# Crear un entorno virtual
python3 -m venv venv
```

```
# Activar entorno (Linux/Mac)
source venv/bin/activate
```

```
# Activar entorno (Windows)
venv\Scripts\activate
```

```
# Instalar dependencias solo en ese entorno
pip install flask
```

virtualenv/venv – Entornos virtuales

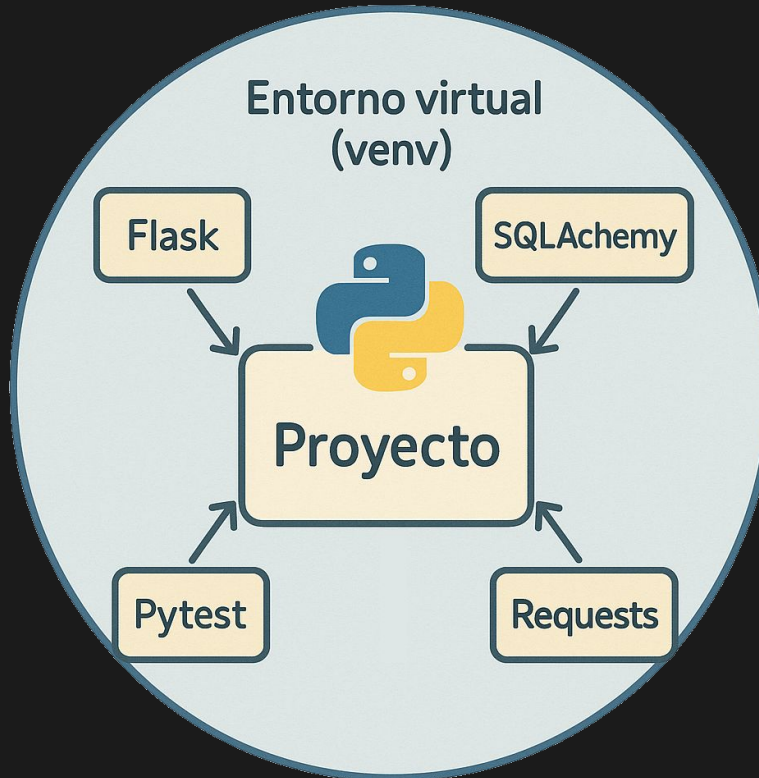
Al activar el entorno, todos los `pip install` quedan contenidos dentro de la carpeta del entorno, sin afectar el sistema global.

El entorno se desactiva con:

- `deactivate`

Beneficio: cada proyecto tiene su propio “sandbox” de dependencias.

venv – Entorno virtual



poetry – Una alternativa moderna

poetry es una herramienta más reciente que busca simplificar y modernizar la gestión de dependencias en Python.

Diferencias principales respecto a pip + requirements:

- Usa `pyproject.toml` para definir dependencias y configuración del proyecto.
- Genera un `poetry.lock` para asegurar versiones exactas (igual que `package-lock.json` en JS).
- Maneja entornos virtuales automáticamente (ya no hay que crear venv a mano).

poetry – Una alternativa moderna

Ejemplo básico:

- `poetry init` # inicializa un nuevo proyecto con `pyproject.toml`
- `poetry add flask`
- `poetry install` # instala todas las dependencias

Ventajas de poetry:

- Más ordenado y moderno.
- Incluye gestión de entornos virtuales.
- Ideal para proyectos medianos o grandes donde se necesita control estricto de dependencias y versiones.

Comparación general

- npm/yarn vs pip/poetry

Comparación general

Cuando hablamos de gestión de dependencias, tanto en JavaScript como en Python, los conceptos son equivalentes aunque con diferentes herramientas. Lo importante es entender que la problemática es la misma en cualquier stack:

- definir dependencias
- fijar versiones
- aislar entornos
- y separar desarrollo de producción

Comparación general

Característica

- Archivo de dependencias

Node.js (npm/yarn)

`package.json`

Python (pip/poetry)

`requirements.txt/`
`pyproject.toml`

- Lock file (versiones exactas)

`package-lock.json /`
`yarn.lock`

`poetry.lock`

- Dependencias dev/prod

`--save-dev` vs `--save`

`[tool.poetry.dev-depe`
`ndencies]`

- Entornos aislados

`nvm` (versiones Node)
o `Docker`

`venv / virtualenv` o
`Docker`

- Popularidad

`npm` es el más usado;
`yarn` optimiza
velocidad

`pip` es el estándar;
`poetry` crece en
adopción

Similitudes: ambos ecosistemas separan manifiesto, lockfile, dependencias de desarrollo y mecanismos de aislamiento.




Diferencias:

- En JS, el aislamiento suele resolverse con nvm o directamente con Docker.
- En Python, el aislamiento es más “natural” con venv y herramientas modernas como poetry.

En la práctica: lo importante no es la herramienta específica, sino aprender a reproducir entornos, documentarlos y mantenerlos estables.

“La gestión de dependencias no es un detalle técnico menor: es la base de la colaboración en equipo y del despliegue confiable en cualquier lenguaje.”

Recursos:

- 
[Curso intensivo sobre NPM](#)
- 
[Python Poetry in 8 Minutes](#)
- 
[NPM vs PIP: A Quick & Comprehensive Comparison](#)

Muchas Gracias

Jeremías Fassi

Javier Kinter