Awesome 👍 Let's go deep into **4. Package Management** because it's **super important in Node.js interviews** and in your daily work as a backend dev (like you're already used to with Composer in Laravel).

---

4. Package Management in Node.js

Node.js relies on **npm** (Node Package Manager) and alternatives like **yarn** or **pnpm** to manage external libraries.

4.1 npm (Node Package Manager)

- Default package manager that comes with Node.js.
- Stores modules in `node_modules/`.
- Manages dependencies via `package.json`.

Common Commands

```
npm init -y              # Create package.json
npm install express      # Install latest version
npm install express@4.18 # Install specific version
npm install --save-dev jest  # Dev dependency
npm uninstall express    # Remove package
npm outdated             # Check outdated packages
npm update               # Update packages
```

👉 **Note**: In modern npm (v5+), `--save` is default.

4.2 Yarn

- Facebook's alternative to npm.
- Faster due to **parallel downloads** and better caching.

Example:

```
yarn init -y
yarn add express
yarn add jest --dev
yarn remove express
```

4.3 pnpm (Optional but Popular in 2025)

- Uses **symlinked node_modules** to save disk space.

- Extremely fast and used in **monorepos**.

4.4 package.json

- Core file that defines **project metadata & dependencies**.
- Example:

```
{
  "name": "my-app",
  "version": "1.0.0",
  "description": "Sample Node.js app",
  "main": "index.js",
  "scripts": {
    "start": "node index.js",
    "test": "jest"
  },
  "dependencies": {
    "express": "^4.18.2"
  },
  "devDependencies": {
    "jest": "^29.0.0"
  }
}
```

👉 Fields explained:

- **name**: package name
- **version**: project version
- **scripts**: custom commands (like Laravel's artisan shortcuts)
- **dependencies**: production deps
- **devDependencies**: only for development/testing

4.5 package-lock.json

- Auto-generated when you install deps with npm.
- Stores **exact versions** of installed packages.
- Ensures **same environment across machines**.

👉 **Interview Tip**:

- `package.json` = flexible dependency versions (e.g., `^4.18.0`).
- `package-lock.json` = exact resolved versions (e.g., `4.18.2`).

4.6 Semantic Versioning (SemVer)

Version format = **MAJOR.MINOR.PATCH**

- `1.2.3` →
    - **1** = Major (breaking changes)
    - **2** = Minor (new features, no breaking changes)
    - **3** = Patch (bug fixes)

Prefixes:

- `^1.2.3` → Updates **minor & patch** (1.x.x)
- `~1.2.3` → Updates **patch only** (1.2.x)
- `1.2.3` → Exact version

👉 Example:

```
"dependencies": {
  "express": "^4.18.2"
}
```

- Allows updates like `4.19.0` but **not** `5.0.0`.

4.7 Global vs Local Packages

- **Local** → Installed in project (`node_modules/`).
- **Global** → Installed system-wide (`npm install -g nodemon`).

👉 Example:

```
npm install -g nodemon   # Run everywhere
npx nodemon server.js    # Use without global install
```

4.8 npx

- Comes with npm v5.2+.
- Used to **execute binaries** without global install.

Example:

```
npx create-react-app myapp
```

👉 Instead of globally installing `create-react-app`, `npx` runs it directly.

📌 Interview Q&A

✅ **Q: Difference between dependencies and devDependencies?**
👉 `dependencies` are needed in production, `devDependencies` are only for development/testing.

✅ **Q: Why do we need package-lock.json?**
👉 To ensure the same dependency tree across environments → avoids "it works on my machine" issues.

✅ **Q: What is Semantic Versioning?**
👉 A system that defines versioning rules → MAJOR.MINOR.PATCH (breaking, feature, bugfix).

✅ **Q: What is npx vs npm?**
👉 `npm` installs packages, `npx` executes them directly without install.

✅ **Q: Difference between npm and yarn?**
👉 Yarn is faster (parallel install), has better caching, and more secure lockfiles.

📌 Key Takeaways

- `npm`, `yarn`, `pnpm` are package managers.
- `package.json` defines dependencies & scripts.
- `package-lock.json` ensures consistent installs.
- SemVer (`^`, `~`) defines version upgrade rules.
- Use **npx** for one-off executions.

👉 Next, we can dive into **5. Asynchronous Programming (callbacks, promises, async/await, error handling)** → one of the most important **Node.js interview topics**.

Do you want me to explain async programming with **step-by-step evolution (callback → promise → async/await)**, or just give **direct interview-style notes**?