Assembling the Infinity Gauntlet

DI and IoC with Decorators in Vanilla JavaScript

Goal: keep DI benefits while staying close to the platform.

Lean Web, explicit code paths, standards-first metaprogramming.

The runnable demo uses plain function calls to simulate decorator-applied metadata, so it stays valid JavaScript without depending on `@` syntax.

Agenda

1. Problem framing: DI value vs framework cost
2. DI and IoC concepts
3. Standards reality: Stage 3 and Stage 1
4. Build: minimal IoC container and decorators
5. Demo: dependency graph resolution
6. Limits, tradeoffs, and adoption path

The Tension

Framework DI gives us:

- decoupling
- testability
- composability

But often adds:

- runtime weight
- framework lock-in
- opaque magic

Lean Web Question

Can we get DI/IoC with:

- Vanilla JavaScript
- explicit code paths
- standards-first metaprogramming

What DI Is

- Dependency Injection means a class receives what it needs from outside.
- The class does not call `new` for its own collaborators.
- That makes behavior easier to test, replace, and reason about.

```
class UserService {
  constructor(apiClient) {
    this.apiClient = apiClient;
  }
}
```

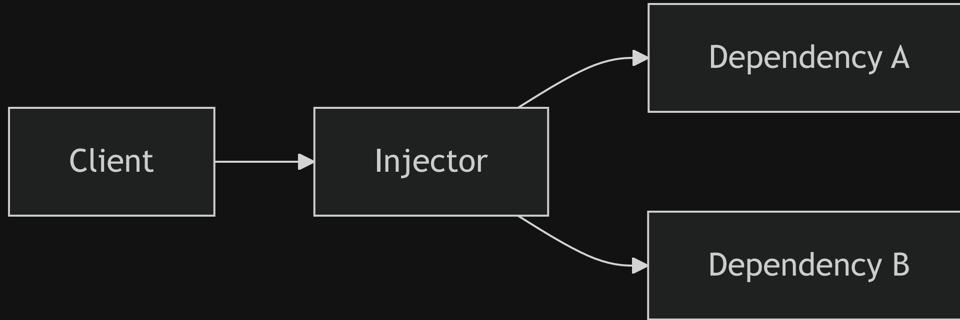
Separation of Concerns

- Classes describe behavior, not assembly.
- Construction and wiring move to a dedicated place.
- This keeps change localized: business logic, composition, and lifecycle can evolve independently.

What IoC Is

- Inversion of Control means object creation and wiring move out of the class.
- A composition root or container decides how the graph is assembled.
- Classes focus on behavior while orchestration happens elsewhere.

The Main Actors



- Three actors shape the flow: client, dependencies, injector.
- The next slides unpack each role briefly, then show how they interact at runtime.

Dependency

- A dependency is any service, helper, or resource another class needs.
- Examples: `Logger`, `ApiClient`, configuration, feature flags.
- In DI, these are supplied explicitly instead of constructed ad hoc.

Client

- The client is the class that uses dependencies to do useful work.
- It should express requirements clearly, usually through the constructor.
- It should not know how the dependency graph is built.

```
class App {
  constructor(userService, logger) {
    this.userService = userService;
    this.logger = logger;
  }
}
```

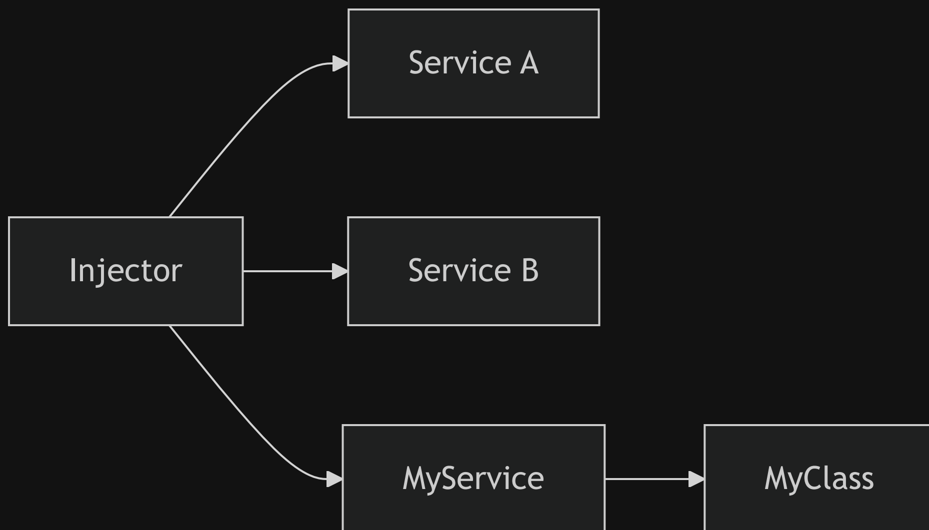
Injector

- The injector creates or looks up dependencies and supplies them to the client.
- In simple setups it can be manual wiring.
- In larger systems it is often an IoC container.

```
const logger = new Logger();
const apiClient = new ApiClient(logger);
const userService = new UserService(apiClient);

const app = new App(userService, logger);
```

Behind the Scenes

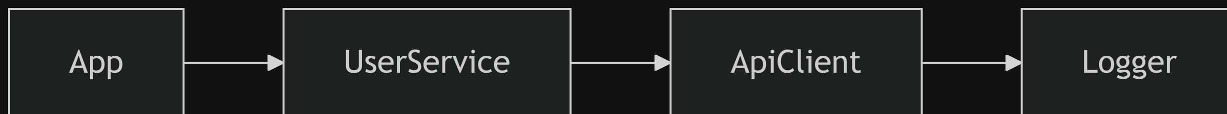


- Once the roles are defined, the injector resolves the graph before the client starts doing work.
- The client receives ready-to-use collaborators instead of assembling them manually.
- This is the bridge from the conceptual model to the container implementation in the next section.

DI vs IoC in One Slide

- DI: dependencies are provided, not constructed internally.
- IoC container: central resolver that controls object creation.
- Result: classes focus on behavior, not wiring.

Soul Stone: classes keep their identity and responsibility because wiring lives outside them.



Standards Reality (2026)

- Decorators proposal: **Stage 3**.
- Parameter decorators proposal: **Stage 1**.

Implication:

- We can build a practical approach now using class-level metadata and current tooling.
- Native support is improving, but not every runtime/toolchain supports the same decorator story end to end.
- Constructor-parameter ergonomics are a likely future improvement.
- In this demo, metadata is stored explicitly on the class, so we do not depend on a separate reflection API.

Reference links in abstract:
`tc39/proposal-decorators` and `tc39/proposal-class-method-parameter-decorators`

Reality Stone: decorators can attach metadata or wrap behavior at definition time.

Container: State and Registration

```
export class Container {
  constructor() {
    this.providers = new Map();
    this.singletons = new Map();
  }

  register(token, factory, { lifecycle = "transient" } = {}) {
    this.providers.set(token, { factory, lifecycle });
  }
}
```

Power Stone: the composition root controls creation.

Container: `registerClass()`

```
registerClass(token, Type, { lifecycle = "transient", deps = [] } = {}) {
  this.register(
    token,
    (container) => {
      const args = deps.map((dep) => container.resolve(dep));
      return new Type(...args);
    },
    { lifecycle }
  );
}
```

Power Stone: the composition root decides how a type becomes an instance.

Container: `resolve()`

```
resolve(token) {
  const provider = this.providers.get(token);
  if (!provider) {
    throw new Error(`No provider registered for token: ${String(token)}`);
  }

  if (provider.lifecycle === "singleton") {
    if (!this.singletons.has(token)) {
      this.singletons.set(token, provider.factory(this));
    }
    return this.singletons.get(token);
  }

  return provider.factory(this);
}
```

Time Stone: singleton vs transient controls lifetime.

- This container is intentionally didactic: it shows lookup, lifecycle, and graph resolution clearly, not every capability a production DI framework would need.

Injectable Decorator

```
export const INJECT_TOKENS = Symbol("di:inject_tokens");

export function Injectable({ deps = [] } = {}) {
  return function applyInjectable(Type) {
    Type[INJECT_TOKENS] = deps;
    return Type;
  };
}

export function getDependencies(Type) {
  return Type[INJECT_TOKENS] ?? [];
}
```

- In the runnable demo, we apply decorator logic with ordinary function calls, keeping it valid plain JavaScript.

Mind Stone: metadata on the class tells the container what it needs without hardcoding constructor calls.

Services and Wiring

```
export class UserService {
  constructor(apiClient) {
    this.apiClient = apiClient;
  }

  listUsers() {
    return this.apiClient.fetchUsers().slice(1);
  }
}
Injectable({ deps: [ApiClient] })(UserService);

export class App {
  constructor(userService, logger) {
    this.userService = userService;
    this.logger = logger;
  }

  start() {
    const users = this.userService.listUsers().join(", ");
    return this.logger.info(`users: ${users}`);
  }
}
Injectable({ deps: [UserService, Logger] })(App);
```

Space Stone: the dependency graph connects services across files and layers without each class having to know the whole universe.

Live Resolution

```
import { Container } from "./container.js";
import { getDependencies } from "./decorators.js";
import { App, ApiClient, Logger, UserService } from "./services.js";

const container = new Container();
const singletonServices = new Set([App, UserService]);

[Logger, ApiClient, UserService, App].forEach((Type) => {
  container.registerClass(Type, Type, {
    lifecycle: singletonServices.has(Type) ? "singleton" : "transient",
    deps: getDependencies(Type),
  });
});

const app = container.resolve(App);
console.log(app.start());
```

Experimental Future: Parameter Decorators

This shape is not production-ready today, but shows direction:

```
// Stage 1 concept sketch only (not used by runtime demo).
// Syntax and semantics may change before standardization.

class UserService {}
class Logger {}

function inject(_token) {
  return function () {};
}

class AppController {
  constructor(
    @inject(UserService) userService,
    @inject(Logger) logger
  ) {
    this.userService = userService;
    this.logger = logger;
  }
}

export { AppController };
```

Tradeoffs

- Stage-3 decorators still require build-tool alignment.
- Parameter decorator syntax is proposal-only for now.
- Metadata design is your responsibility in Vanilla JS.
- If you want `@decorator` syntax in production, your toolchain must support the current decorators semantics.
- Benefit: full control, low ceremony, framework-independent architecture.
- AI era point of view: plain JavaScript is easier to inspect, fix, and customize than framework-bound generated code.
- AI-assisted experiments are easier to port into production when the core runtime is platform-native.
- **The Snap**: when proposals and tooling mature, remove compatibility scaffolding and keep the explicit DI core.

Conclusion

Upcoming ECMAScript features make declarative DI viable with platform-native code.

Start minimal:

1. explicit container
2. lightweight metadata decorators
3. clear boundaries between runtime and experimental syntax

WEB DAY 2026

Assembling the Infinity Gauntlet: DI and IoC with Decorators in Vanilla JavaScript

Giorgio Galassi
Senior Frontend Engineer
Freelancer

improove