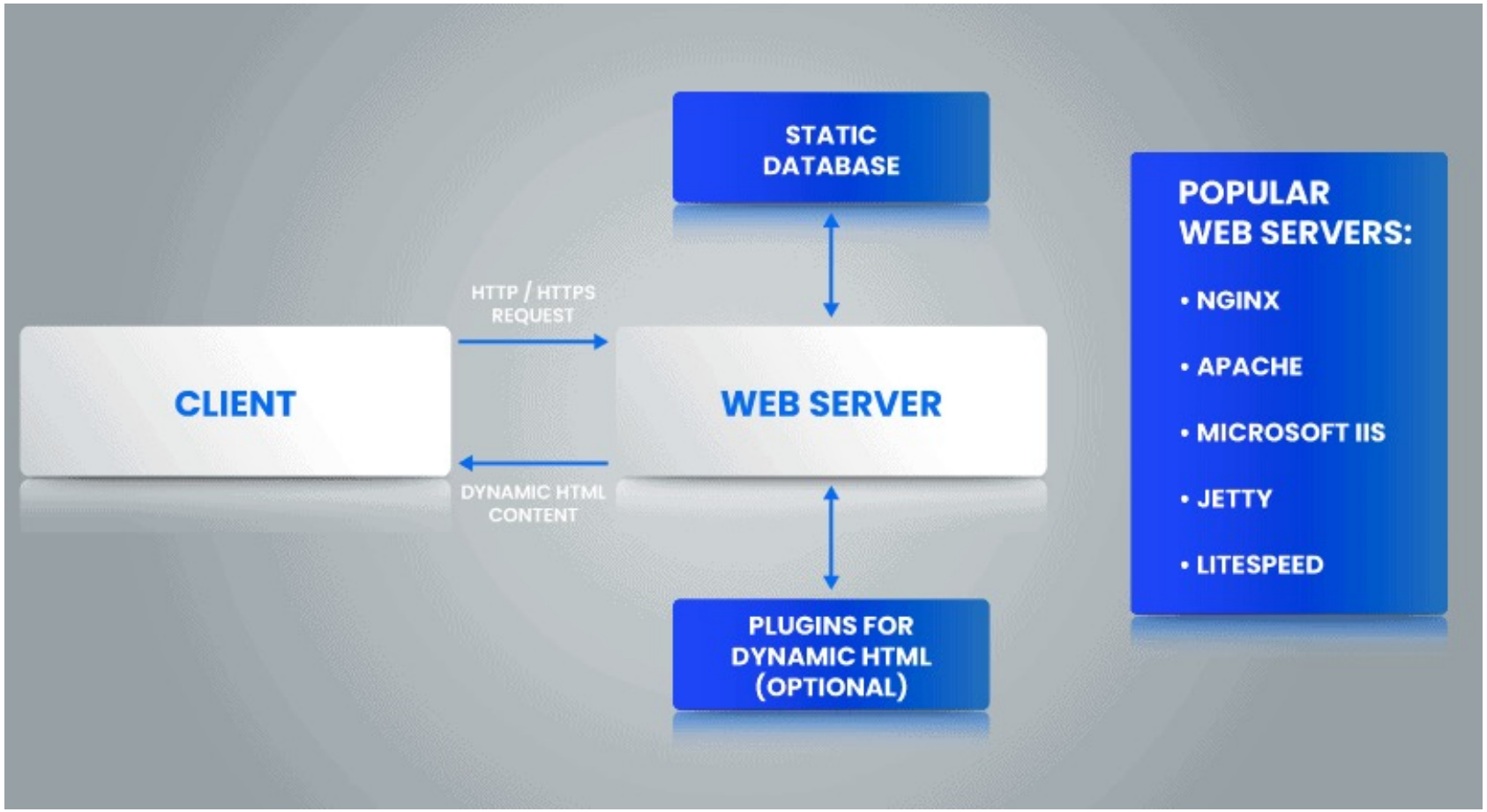
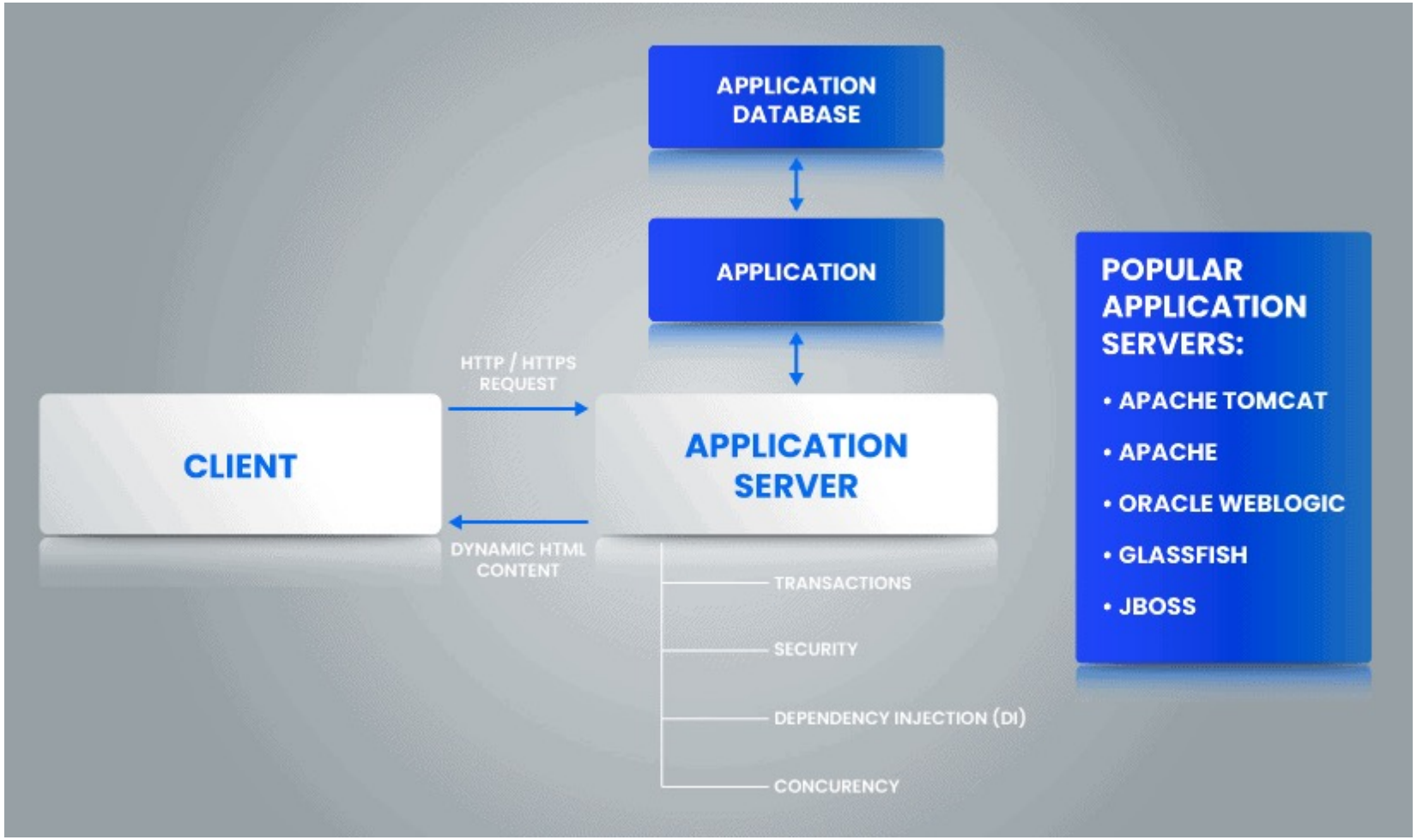
Web e Rest Java Framework

Web server
(HTTP only, host websites, mainly static contents, etc)



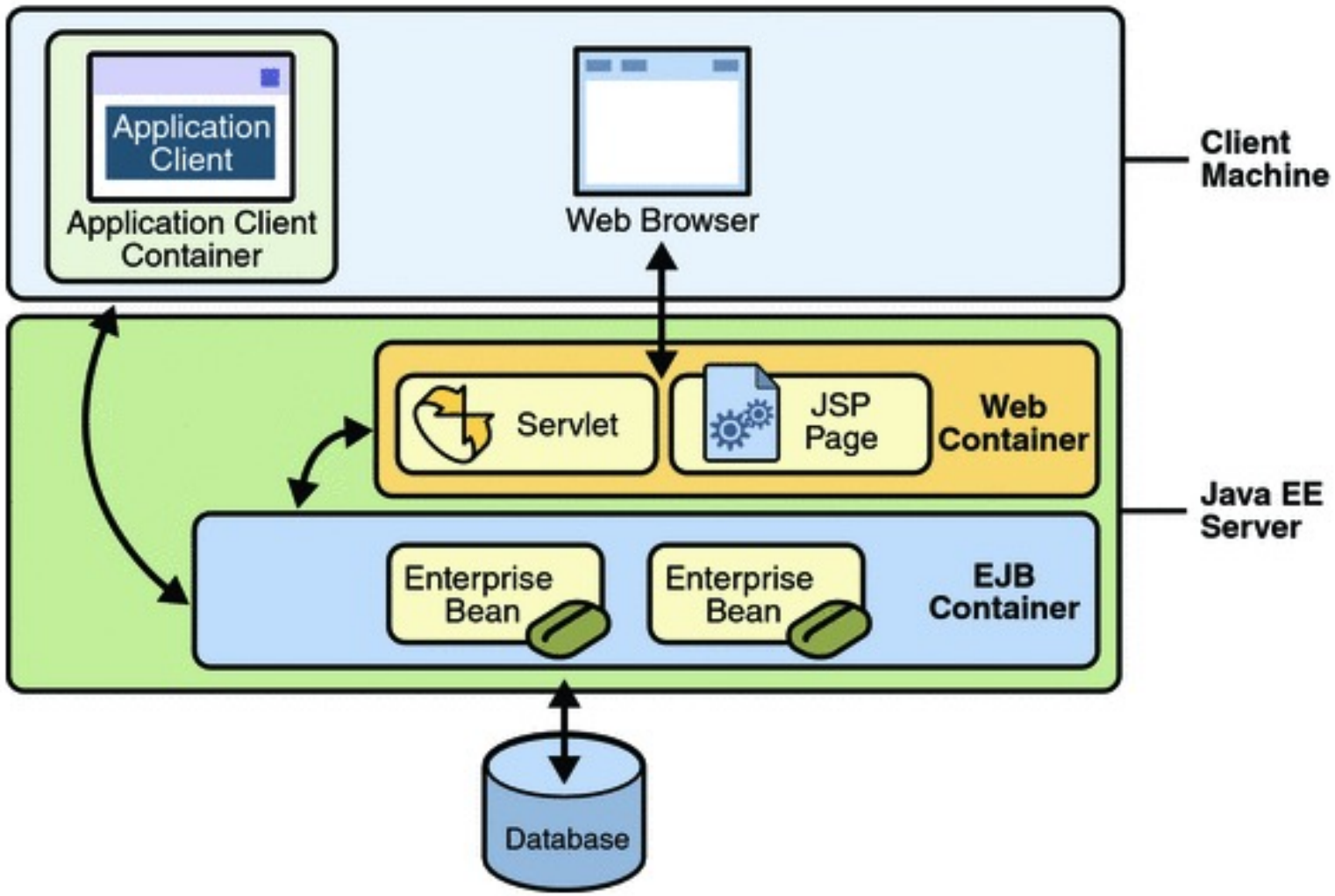
Application server (multiprotocol, transactions and concurrency, different type of containers, etc)



Java framework

- Esistono numerosi framework per costruire applicazioni web con Java
- Le Servlet sono la tecnologia di base introdotta da J2EE
- I framework si appoggiano su application server come Tomcat, Glassfish o altri container
  - Spring (J2EE) https://spring.io/
  - JavaServer Faces (basato su MVC) https://jakarta.ee/specifications/faces/

J2EE


Application Client
Application Client Container
Web Browser
Client Machine
Servlet
JSP Page
Web Container
Java EE Server
Enterprise Bean
Enterprise Bean
EJB Container
Database

Dai framework a micro framework

- Uno dei problemi più frequenti nell'adozione di questi framework è
  - Configurazione complessa (tramite XML ecc)
  - Il container stesso può diventare più costoso dell'applicazione da sviluppare soprattutto per server REST che hanno solo la funzione di ridirezionare le richieste HTTP verso altri servizi (es un server DB) come spesso capita con architetture multi-tier
- Da alcuni anni sono stati creati dei framework per microservizi che semplificano e alleggeriscono la creazione di server REST/web
- I micro framework semplificano lo sviluppo e spesso sono multi linguaggio (Java, Kotlin, Node, Go, ecc)

Esempi di sistemi per micro Java framework

- Sparkjava http://sparkjava.com/
- Dropwizard https://www.dropwizard.io/en/latest/
- Javalin https://javalin.io/
- Micronaut https://micronaut.io/
- …
- Vedremo più in dettaglio Sparkjava da non confondere con Apache Spark

=/=

Sparkjava

- Spark is a Free and Open Source Software (FOSS) application framework written in Java create nel 2011
- It is a lightweight library that you link into your application to start serving up data.
- It allows you to define routes and dispatch them to functions that will respond when those paths are requested.
- This is done in code using a straight-forward interface, without the need for XML configuration files or annotations like some other web frameworks require.

Building blocks

- Route
  - **verb** (get, post, put, delete, head, trace, connect, options)
  - **path** (/hello, /users/:name)
  - **callback** (request, response) -> {})
- Le route sono valutate nell'ordine in cui sono scritte
- Le callback si possono scrivere come lambda function di Java8 (bisogna modificare Java Compiler Compliance Level di Eclipse)

Esempio

```
import static spark.Spark.*;

public class HelloWorld {
    public static void main(String[] args) {
        get("/hello", (req, res) -> "Hello World");
    }
}
```

```
http://localhost:4567/hello
```

```
get("/", (request, response) -> {
    // Show something
});

post("/", (request, response) -> {
    // Create something
});

put("/", (request, response) -> {
    // Update something
});

delete("/", (request, response) -> {
    // Annihilate something
});

options("/", (request, response) -> {
    // Appease something
});
```

CRUD

Route patterns can include named parameters, accessible via the `params()` method on the request object:

```
// matches "GET /hello/foo" and "GET /hello/bar"
// request.params(":name") is 'foo' or 'bar'
get("/hello/:name", (request, response) -> {
    return "Hello: " + request.params(":name");
});
```

By calling the stop() method the server is stopped and all routes are cleared.

```
stop();
```

Filters

Before-filters are evaluated **before each request**, and can read the request and read/modify the response.
To stop execution, use `halt()`:

```
before((request, response) -> {
    boolean authenticated;
    // ... check if authenticated
    if (!authenticated) {
        halt(401, "You are not welcome here");
    }
});
```

After-filters are evaluated **after each request**, and can read the request and read/modify the response:

```
after((request, response) -> {
    response.header("foo", "set by after filter");
});
```

Waiting for Initialization

You can use the method `awaitInitialization()` to check if the server is ready to handle requests. This is usually done in a separate thread, for example to run a health check module after your server has started.
The method causes the current thread to wait until the embedded Jetty server has been initialized. Initialization is triggered by defining routes and/or filters. So, if you're using just one thread don't put this before you define your routes and/or filters.

```
awaitInitialization(); // Wait for server to be initialized
```

Copy

Concorrenza

- Sparkjava è
  - blocking per le richieste
    Il client riceve response come valore di ritorno della callback che definisce handler
  - non blocking per il networking
    Le callback sono eseguite concorrentemente nel threadpool associato all'embedded server web (Jetty)
    bisogna quindi porre attenzione alle race condition su strutture dati in memoria

ThreadPool (embedded web server)

You can set the maximum number of threads easily:

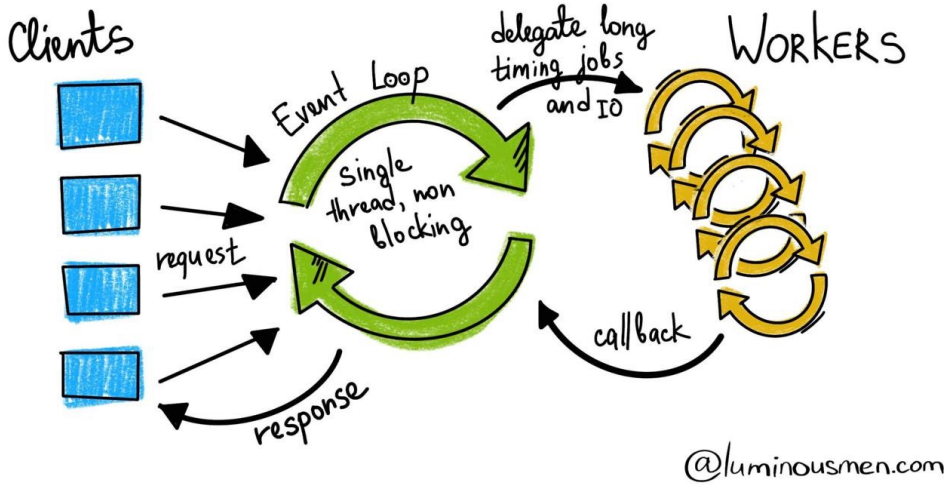
```
int maxThreads = 8;
threadPool(maxThreads);
```

You can also configure the minimum numbers of threads, and the idle timeout:

```
int maxThreads = 8;
int minThreads = 2;
int timeOutMillis = 30000;
threadPool(maxThreads, minThreads, timeOutMillis);
```

Asincronia

- Potremmo pensare di usare nuovi Thread per spostare su thread dedicati il processing delle richieste e quindi liberare i thread usati dal server web embedded



Async + Continuation (thenAccept)

Si può ridefinire il SessionHandler di Jetty inserendo invocazione asincrona e continuation

Oppure si può usare AsyncContext come segue:

```
public static void main(String[] args) {
    get("/benchmark", (request, response) -> {
        AsyncContext ac = request.raw().startAsync();
        CompletableFuture<Void> cf = CompletableFuture.supplyAsync(() -> getMeSomethingBlocking());
        cf.thenAccept((Void) -> ac.complete());
        return "ok";
    });
}
```

Non blocking per il client ma il risultato?

Il client usa polling per vedere risultato sul server (altre chiamate REST)

Si usano altri meccanismi come Server Sent di HTML5 (notifiche update da server a client) o publish/subscribe ecc

Si usa Websocket (REST + socket bidirezionale che rimane aperto durante sessione)

WebSockets

WebSockets provide a protocol full-duplex communication channel over a single TCP connection, meaning you can send message back and forth over the same connection.

```
webSocket("/echo", EchoWebSocket.class);
init(); // Needed if you don't define any HTTP routes after your WebSocket routes
```

```
@WebSocket
public class EchoWebSocket {

    // Store sessions if you want to, for example, broadcast a message to all users
    private static final Queue<Session> sessions = new ConcurrentLinkedQueue<>();

    @OnWebSocketConnect
    public void connected(Session session) {
        sessions.add(session);
    }
```