JCAD

JavaScript Connector API for Decorators

Last update: 05/11/2018

What is JCAD?

JCAD stands for JavaScript Connector API for Decorators.

Both, JAVA annotations and TypeScript decorators, use the @ character to add metadata to source code.

JAVA Annotations:

- declarative interface + class implementation
- members defined through a one dimension array

TypeScript Decorators:

- use decorator pattern to modify behavior of the target object
- properties defined as function parameters

JCAD turns TypeScript decorators into an abstraction layer that let developers choose implementations to execute routines depending on injected metadata.

TypeScript Decorators 1/2

TypeScript decorators are special kind of declarations that can be attached to classes declarations, methods, accessors, propertys, or parameters.

Decorators use the form `@expression`, where `expression` must evaluate to a function that will be called at runtime with information about the decorated declaration:

```
export class MyClass {

  @log()
  public myMethod(arg:any):string {
    return "Message: " + arg;
  }
}
```

Given the decorator `@log` we might write the `log()` function as follows:

```
function log(target:any) {
  // do something with 'target' ...
}
```

TypeScript Decorators 2/2

We define the concrete behavior of the `@log` decorator as shown below:

```
export function log(target:any, propertyKey:string,
                                descriptor:TypedPropertyDescriptor<any>) {

  const originalMethod = descriptor.value;

  descriptor.value = function(...args:any[]):any {
    console.log("The method args are: " + JSON.stringify(args));
    const result = originalMethod.apply(this, args);
    console.log("The return value is: " + result);
    return result;
  };

  return descriptor;
}
```

concrete code

EcmaScript (*TypeScript*) decorators are built over the Decorator Pattern, where `@expression` is used to directly invoke its implementation.

JCAD allows to inject concrete code into the decorator function.

JCAD Decorators

JCAD decorators:

- are POJOs *(Plain Old JavaScript Objects)*
- implement the `Decorator` interface

```
export class Log implements Decorator  {

  constructor(){}

  public decorate(target:any, key:string, descriptor:PropertyDescriptor,
                  ...args:any[]):any {

    const originalMethod = descriptor.value;

    descriptor.value = function(...args:any[]):any {
      console.log("The method args are: " + JSON.stringify(args));
      const result = originalMethod.apply(this, args);
      console.log("The return value is: " + result);
      return result;
    };

    return descriptor;
  };
}
```

concrete code

JCAD Abstraction

We use JCAD registries to define abstraction for `@expression`.

JCAD connectors:

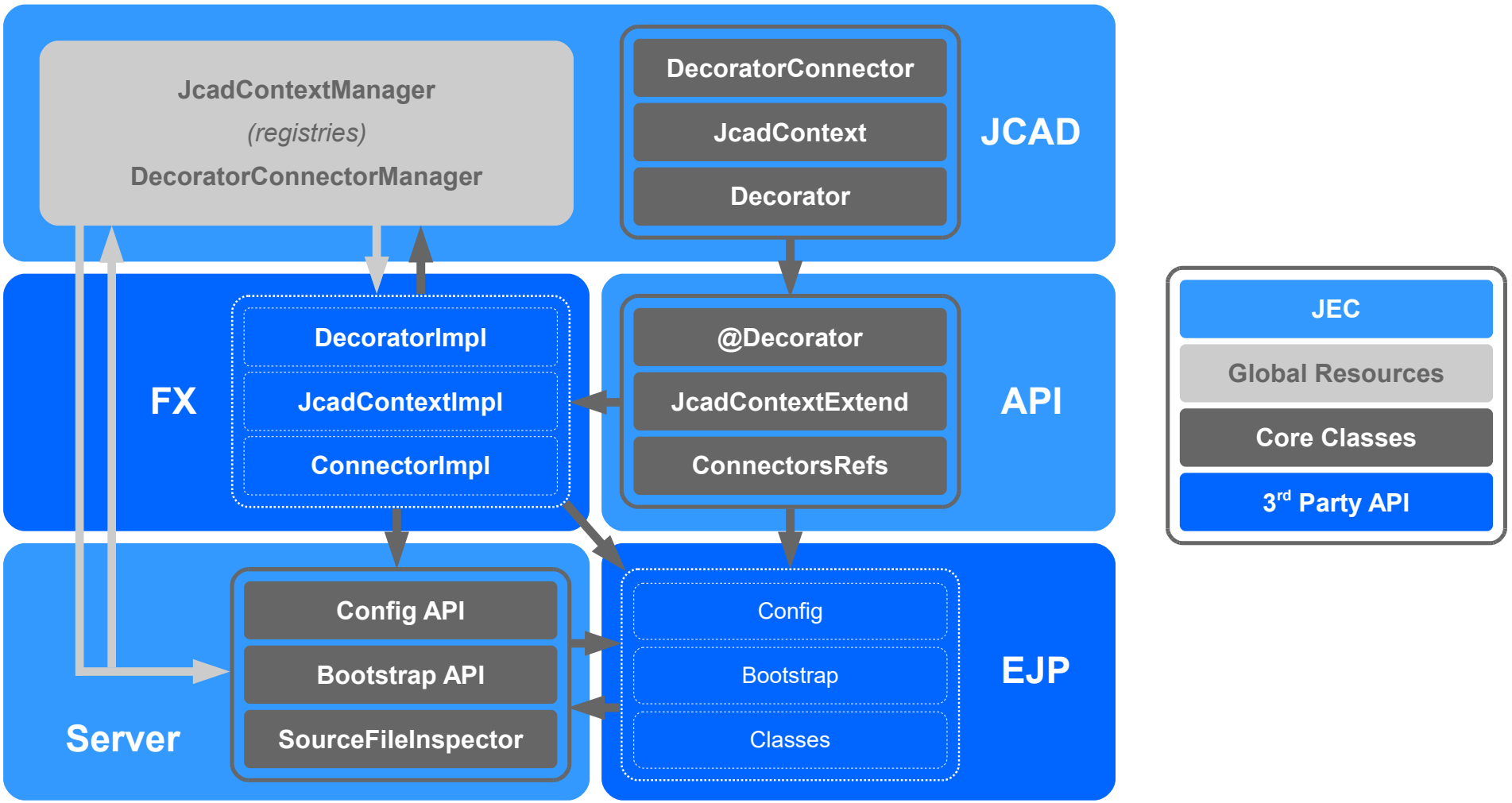
- inject concrete code into the abstract decorator function
- are registered by the implementator (3rd party framework) and are associated to a specific JCAD context

```
const REF:string = LogConnectorRef.CONNECTOR_REF;

export function log(...args:any[]):Function {

    return function (target:any, key:string, descriptor:PropertyDescriptor):any {

      var ctx:JcadContext = JcadContextManager.getInstance().getContext(REF);
      return DecoratorConnectorManager.getDecorator(REF, ctx)
                                      .decorate(target, key, descriptor, args);

    }
}
```

JCAD abstraction

JCAD Architecture

JCAD is designed as a **S**ervice **P**rovider **I**nterface *(SPI)*.



Flexibility

The JCAD abstraction layer allows to:

- create specifications and APIs based upon TypeScript decorators
- make EJPs based on these specifications portable
- create custom implementations of abstract top-level APIs

Frameworks integration is made through the bootstrap API:

```
import { BootstrapScript, Bootstrap, JecContainer } from "jec-commons";
import { SandcatBuilder } from "jec-sandcat";

@Bootstrap()
export class InitApp extends BootstrapScript {

  public run(container:JecContainer):void {
    new SandcatBuilder().build(container)
                        .process((err:any)=>{});
  }
}
```

Processing

Principle

TypeScript decorators are processed at runtime during instantiation phases.

- JCAD detect TypeScript decorators before instantiation phases
- frameworks use JCAD to perform the autowiring process

Framework containers are responsible for:

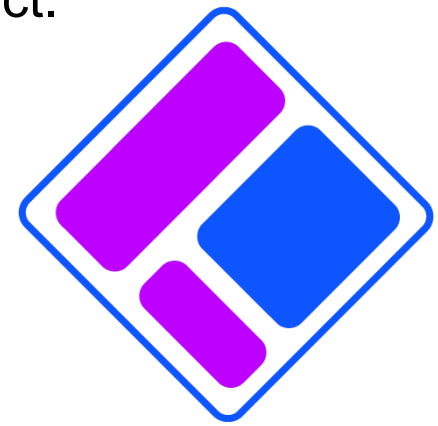
- managing objects depending on annotations (e.g. instantiation)
- injecting metada into managed objects
- establishing communications with the current JEC implementation

Performances

- developers can create their own processors
- JEC provides access to each container processor to improve performances and save development time

Where to go from here?

JCAD is part of the JEC project.



JEC

JavaScript Enterprise Container

For more information and documentation on JCAD and JEC visit:

- JEC project website

JEC implementations that are based on JCAD:

- jec-glasscat-core
- jec-sandcat
- jec-tiger