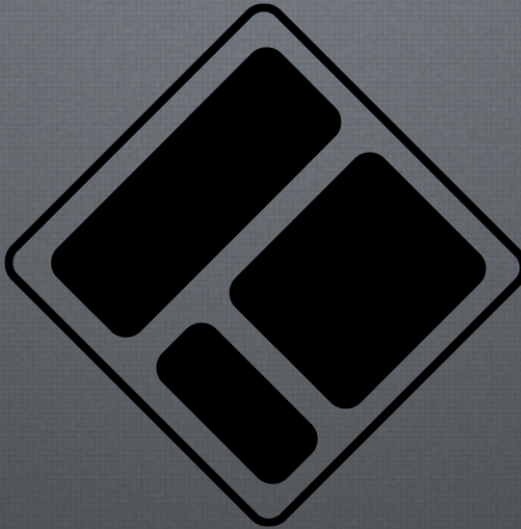


# App Sample



Let's create a JEC modular application!

# Objective



The aim of this presentation is to build a JEC modular application based on microservices and a SPA.

We will create a **Book Library Manager** from scratch, that gives users commons information about registered books and their authors.

This will show you how to:

- serve static ressources (e.g. book covers)
- access NoSQL Data Bases (MongoDB)
- expose data into an Angular app

This presentation demonstrates all the benefits of using

## JEC and GlassCat

to easily build modern scalable applications.

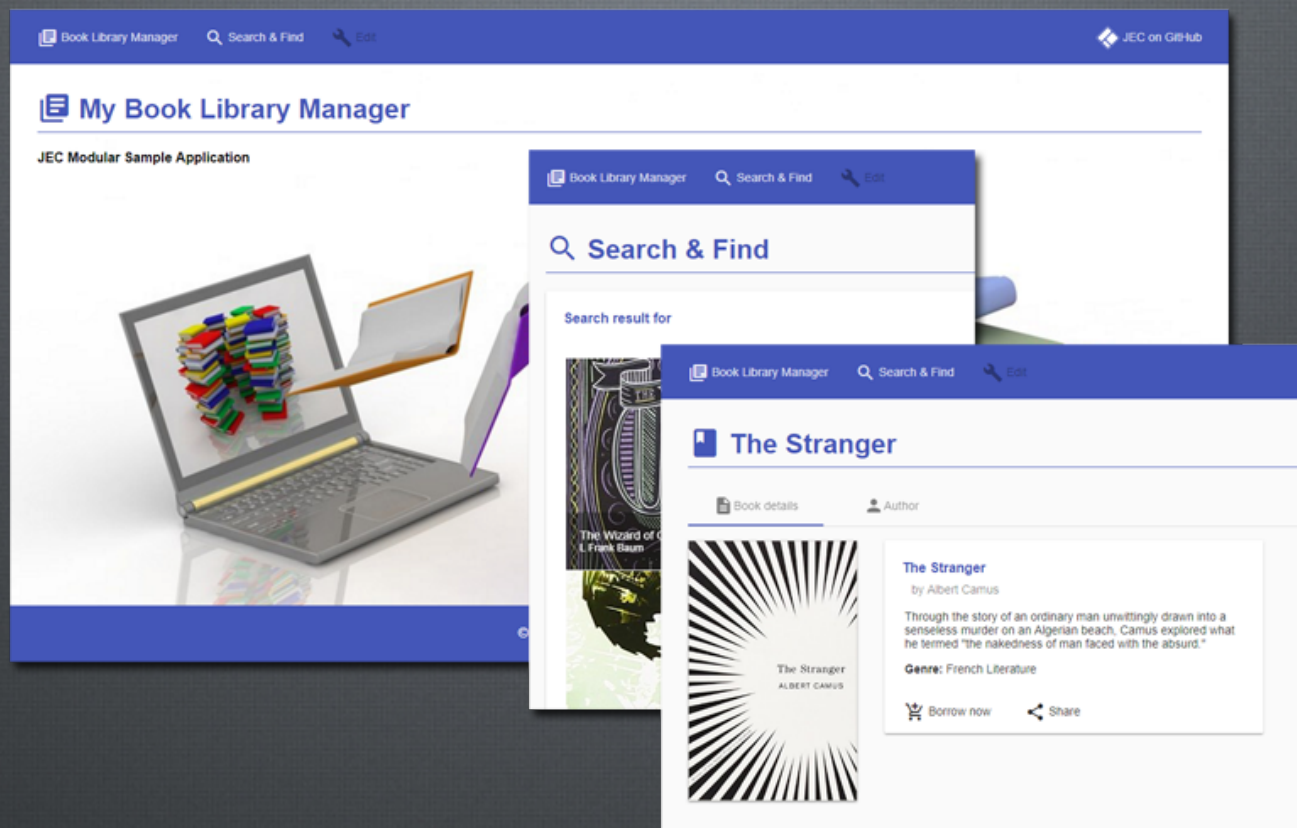


# Source Code

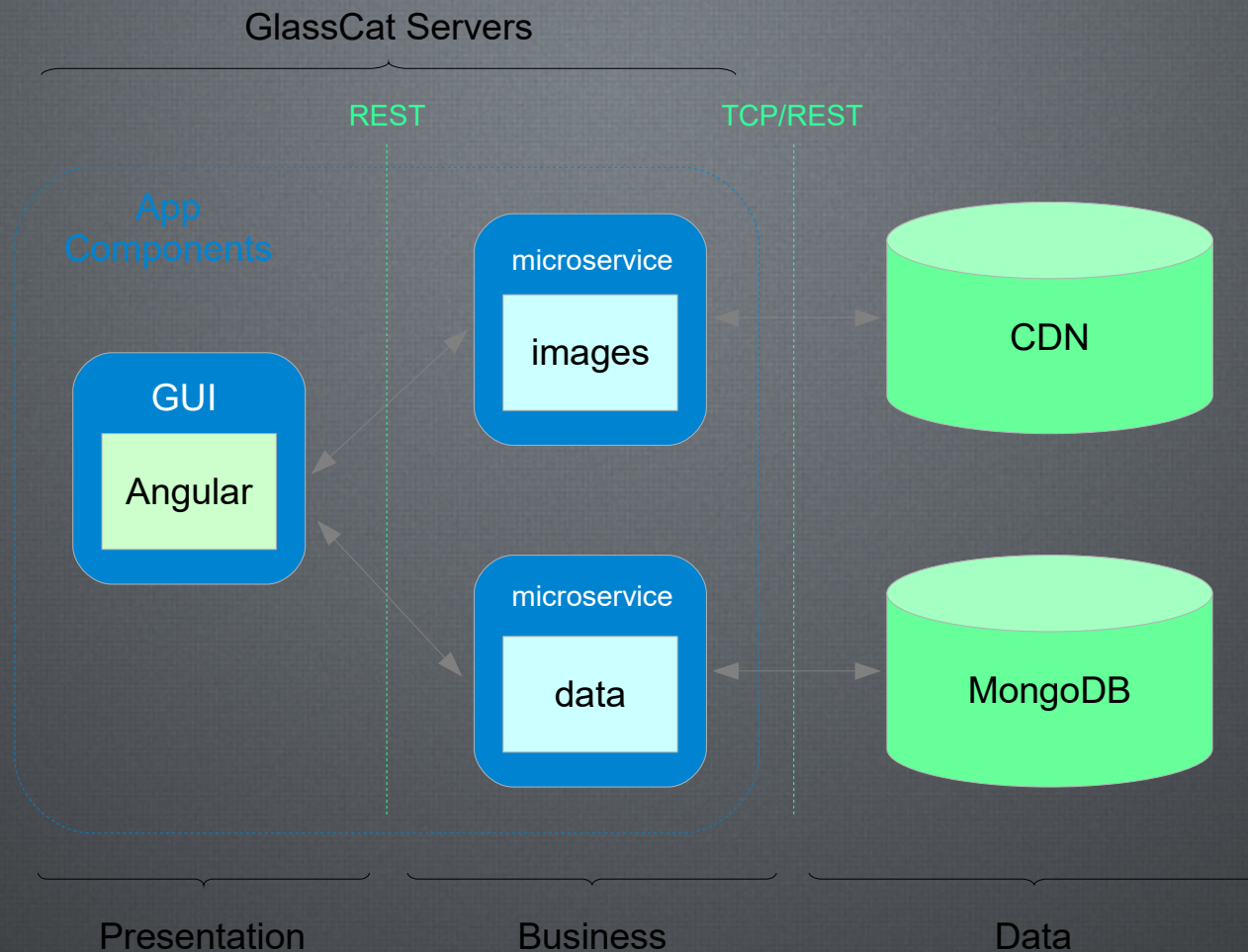


Fully functional code of the Book Library Manager application is available at:

- <https://github.com/pechemann/jec-app-samples/>



# App Architecture



Modular approach is the foundation of JEC apps.



# GlassCat Install



JEC-CLI is a command line tool that allows to quickly prototype with GlassCat.

## 1. Install JEC-CLI:

```
$ [sudo] npm install jec-cli -g
```

## 2. Create a directory where to install the server:

```
$ mkdir test-jec  
$ cd test-jec
```

## 3. Install a GlassCat server instance in the new directory:

```
$ jec install-glasscat
```

## 4. Start the server:

```
$ glasscat start
```

# Building JEC Archetypes 1/3



Archetype will help authors create EJP templates for users, and provides users with the means to generate parameterized versions of those project templates.

GlassCat Project Model (*GPM*) is the JEC project templating toolkit.

To create a new project based on an Archetype, you need to call `glasscat archetype` goal, like the following:

```
$ glasscat archetype --gpm=[basic] --projectName=[myProject]
                        --directory=[myDirectory] --contextRoot=[myContextRoot]
```

JEC provides several GPMs:

GPM	Description
basic	An archetype to generate a sample EJP.
microservice	An archetype to generate a RESTful EJP based on JARS and Sandcat.
angular	An archetype to generate an Angular application with Material dependencies.

# Building JEC Archetypes 2/3



All JEC archetypes can be built from the GlassCat Admin Console:

## GlassCat Admin Console

[Home](#) / [Console](#) / [Domains](#) / [Wizard](#)

### Domain Wizard

All new domain applications will be deployed in the server workspace.

Create a new project

✕ Cancel

Basic

**Project Model Details:**

Name:	Basic
Version:	1.0.0
Description:	Deploys a basic GlassCat project without any dependencies.
Author:	ONSOFT SYSTEMS
Help:	The following table shows the additional parameters available for this GPM:



# Building JEC Archetypes 3/3



This sample application shows the use of a `basic`, a `microservice` and an `angular` archetype, as defined below:

GPM	Name	Directory	Contextroot
<code>basic</code>	<code>sample-blm-images</code>	<code>sample-blm-images</code>	<code>sample-blm-images</code>
<code>microservice</code>	<code>sample-blm-books</code>	<code>sample-blm-books</code>	<code>sample-blm-books</code>
<code>angular</code>	<code>sample-blm-app</code>	<code>sample-blm-app</code>	<code>sample-blm-app</code>

Each project is used for specific purpose:

Project	Description
<code>sample-blm-images</code>	A microservice app that serves static images.
<code>sample-blm-books</code>	A microservice app that provides access to books information through a REST API.
<code>sample-blm-app</code>	The angular app that displays information provided by both microservices.

GPM archetypes have been designed to easily build modular and scalable applications.



# Domains Separation 1/4



GlassCat application servers define each EJP as a "domain".

- Contrary to JAVA EARs, domains cannot contain more than one application module.
- JEC does not specify any gateway to communicate from one domain to another.

Develop faster with JEC domains:

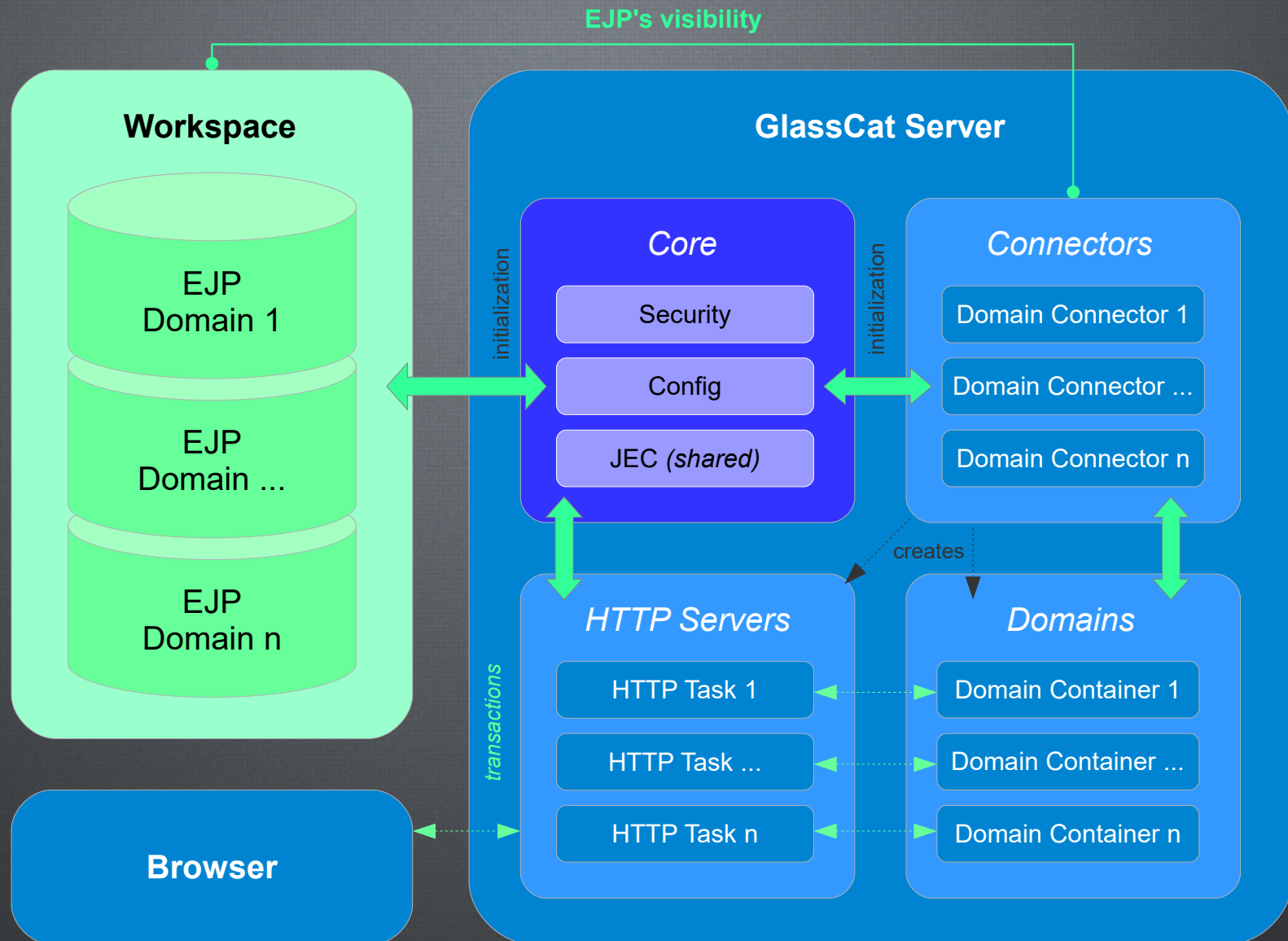
- Domains are a good way to split complex applications into microservice components.
- Contrary to JAVA EE and Spring Boot, you can use a single server instance to simulate container orchestration, and/or API management, in development environments.

GlassCat architecture facilitates microservices isolation by using connectors and separating HTTP servers (*see next slide*).

# Domains Separation 2/4



GlassCat Architecture





# Domains Separation 3/4



You use the GlassCat Admin Console to easily create and manage HTTP tasks:

New HTTP Task

Cancel

ID \*:

Server name \*:

Address \*:

127.0.0.1

Domain \*:

localhost

Port \*:

Secured:

no

SSL Path:

Monitoring enabled:

no

Monitor Factory:

Create

Reset

# Domains Separation 4/4



In order to deploy the sample application, we need to create 3 new HTTP tasks:

- [name/ID:testServer1 - port:3001]
- [name/ID:testServer2 - port:3002]
- [name/ID:testServer3 - port:3003]

Each HTTP task will be associated to only one domain:

GPM	Name	Server
basic	sample-blm-images	testServer2
microservice	sample-blm-books	testServer3
angular	sample-blm-app	testServer1

By applying this concept to all new domain, this application become highly scalable.



Domain containers are stateless by default.



# Creating Microservices



Microservices is a variant of the service-oriented architecture (SOA) architectural style that structures an application as a collection of loosely coupled services. [...] services should be fine-grained and the protocols should be lightweight.<sup>[1]</sup>

## JEC microservices are:

- deployed through EJPs
- wrapped within domain containers
- exposed by using REST APIs

## JEC microservices APIs:

- jslets can respond to any type of request (*including REST architectural style*)
- JavaScript API for RESTful Services (*JARS*) has been designed for building REST APIs

## JEC microservices philosophy:

- Expose a single resource by EJP and deploy only one EJP by server.

[1] Microservices, on Wikipedia

# Serving Static Resources 1/2



The easiest way to serve static resources is to use jslets.

## Jlets:

- are similar to JAVA EE servlets
- support both, file config and **auto-configuration**
- provide built-in methods to serve static files

Jslets can be used to expose static resources through a REST API.

## Static resources should be:

- stateless
- cacheable

You can use both, admin console and CLI, to create new jslets:

```
$ ejp create-jslet --name=[myJsletName] --path=[myJsletDirectory]
```



# Serving Static Resources 2/2



Jslets provide built-in functionalities to let developers manage HTTP responses:

```
import { HttpJslet, WebJslet, HttpRequest, HttpResponse } from "jec-exchange";
import { HttpHeaders } from "jec-commons";
import * as path from "path";

const PATH:string = process.pwd()+ "/path/to/data/images/books/covers/";

@WebJslet({
  name: "CoversJslet",
  urlPatterns: ["/covers/*"]
})
export class Covers extends HttpJslet {

  public doGet(req:HttpRequest, res:HttpResponse, exit:Function):void {
    let filePath:string = PATH + path.basename(req.getPath());
    res.setHeader(HttpHeaders.ACCESS_CONTROL_ALLOW_ORIGIN, "http://localhost:3001");
    res.setHeader(HttpHeaders.CACHE_CONTROL, "public, max-age=31536000");
    exit(req, res.sendFile(filePath), null);
  }
}
```

# Creating REST Services 1/2



JEC default JARS implementation is provided by the Sandcat framework.

The `microservice` GPM includes initialization script for the Sandcat framework:

```
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)=>{});
  }
}
```

Sandcat initialization provides support for automatic resources detection.



# Creating REST Services 2/2



`microservice`-GPM-projects (*MS-GPM*) are "lightweight" containers:

- JARS API and implementation dependencies are not included in the EJP
- each EJP owns a Sandcat container instance that manage its resources

## **MS-GPM projects start fast:**

- Sandcat resource detection process shares GlassCat's file pre-processor engine
- Sandcat is built over the jslet specification



Do not forget to remove unused resource classes (*\*.ts, \*.js files*).

## **MS-GPM projects are flexible:**

- REST API versioning support
- efficient annotations parameters

# Ease-of-use



JARS API is highly intuitive to learn and use.

It provides support for sub-routing, parameters extraction and MIME types treatment:

```
@ResourcePath({
  path: "/books",
  crossDomainPolicy: "http://localhost:3001",
  produces: "application/json"
})
export class Books {

  @Inject({ type: "services.BooksDao" })
  public dao:BooksDao;

  @GET()
  public getBooks(@Exit exit:Function):void {
    this.dao.getBooks((data:any, err:any)=> {
      exit(data, err);
    });
  }
}
```



# Book Search Functionality



The BLM sample application implements a basic functionality for searching books.

We use the `@QueryParam` decorator to easily work with HTTP queries:

```
export class Books {  
  
  @GET({  
    route: "/search"  
  })  
  public findBooks(@QueryParam token:string, @Exit exit:Function):void {  
    this.dao.findBooks(token, (data:any, err:any)=>{  
      exit(data, err);  
    });  
  }  
}
```

To get all books written by William Shakespeare you just call the following URI:

- `Http://localhost:3002/blm-sample-app/books/search?token=shakespeare`

# Dependency Injection



Dependency injection is:

- specified by the JavaScript Dependency Injection ( $J_E DI$ ) API
- implemented through the Bobcat framework

```
const FILE_PATH:string = process.pwd()+ "/path/to/data/books.json";

@Injectable({
  retention: "DEV"
})
export class LocalBooksDao implements BooksDao {

  public getBooks(result:(data:any, err:any)=>void):void {
    fs.readFile(
      FILE_PATH,
      EncodingFormat.UTF8,
      (err:NodeJS.ErrnoException, data:string)=>{
        exit(data, err);
      }
    );
  }
}
```



# Creating Angular GUIs 1/3



The `angular` GPM (*NG-GPM*) builds Angular-based EJPs.

NG-GPMs include:

- Angular 4 dependencies
- Angular Material dependencies



NG-GPMs use Google Web Fonts as default configuration to include Material icons.

NG-GPM projects use `SystemJS` to manage dynamic ES modules.

The Angular application is:

- packaged into the `webapp/app` folder
- served by the `jslets/AngularApp` jslet
- available at the `http://localhost:3001/sample-blm-app/app` URI

# Creating Angular GUIs 2/3



Angular `HttpClient` service allows to easily access to microservices APIs:

```
@Component({
  selector: "search-find",
  templateUrl: "./templates/search.html"
})
export class SearchComponent implements OnInit {

  constructor(private _httpClient:HttpClient){}

  public books:any[];

  ngOnInit():void {
    this._httpClient.get("http://localhost:3002/sample-blm-books/books")
      .subscribe(
        data => {
          this.books = data as any[];
          console.log(this.books);
        },
        (err:HttpErrorResponse)=> { console.log(err); }
      );
  }
}
```



# Creating Angular GUIs 3/3



Angular Material components allows to display and easily layout data returned by microservices:

```
<md-card class="example-card">
  <md-card-header>
    <md-card-title>
      List of available books:
    </md-card-title>
  </md-card-header>
  <md-card-content>
    <md-grid-list cols="4" rowHeight="250px">
      <md-grid-tile *ngFor="let book of books">
        
        <md-grid-tile-footer>
          <h3 md-line>{{book.title}}</h3>
          <h4 md-line>{{book.author.name}}</h4>
        </md-grid-tile-footer>
      </md-grid-tile>
    </md-grid-list>
  </md-card-content>
</md-card>
```

# Where to go from here?

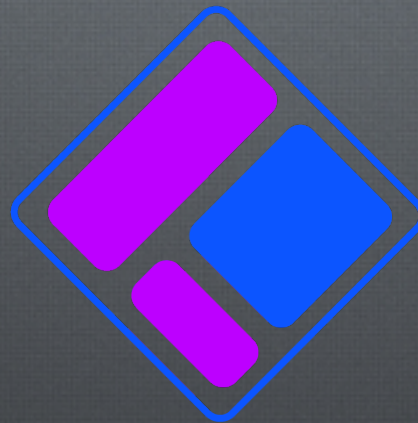


For more information and documentation on GlassCat and JEC please visit:

- GlassCat Project
- JEC Sample Projects
- JEC Youtube Channel

GlassCat is part of the JEC project:

- JEC project on GitHub



**JEC**  
JavaScript Enterprise Container