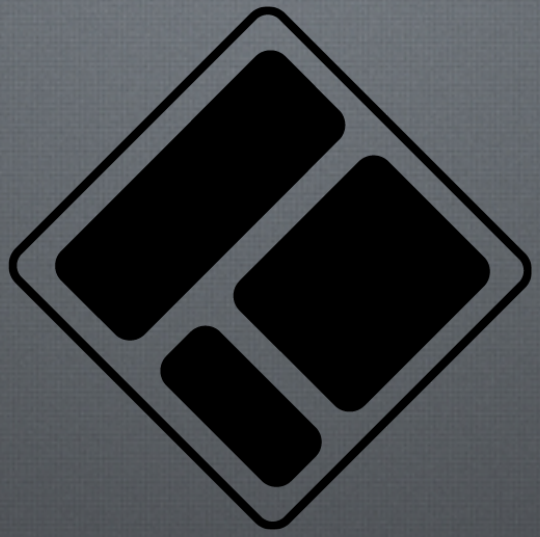
App Sample



Let's create a JEC modular application!

Last update: 09/27/2017

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

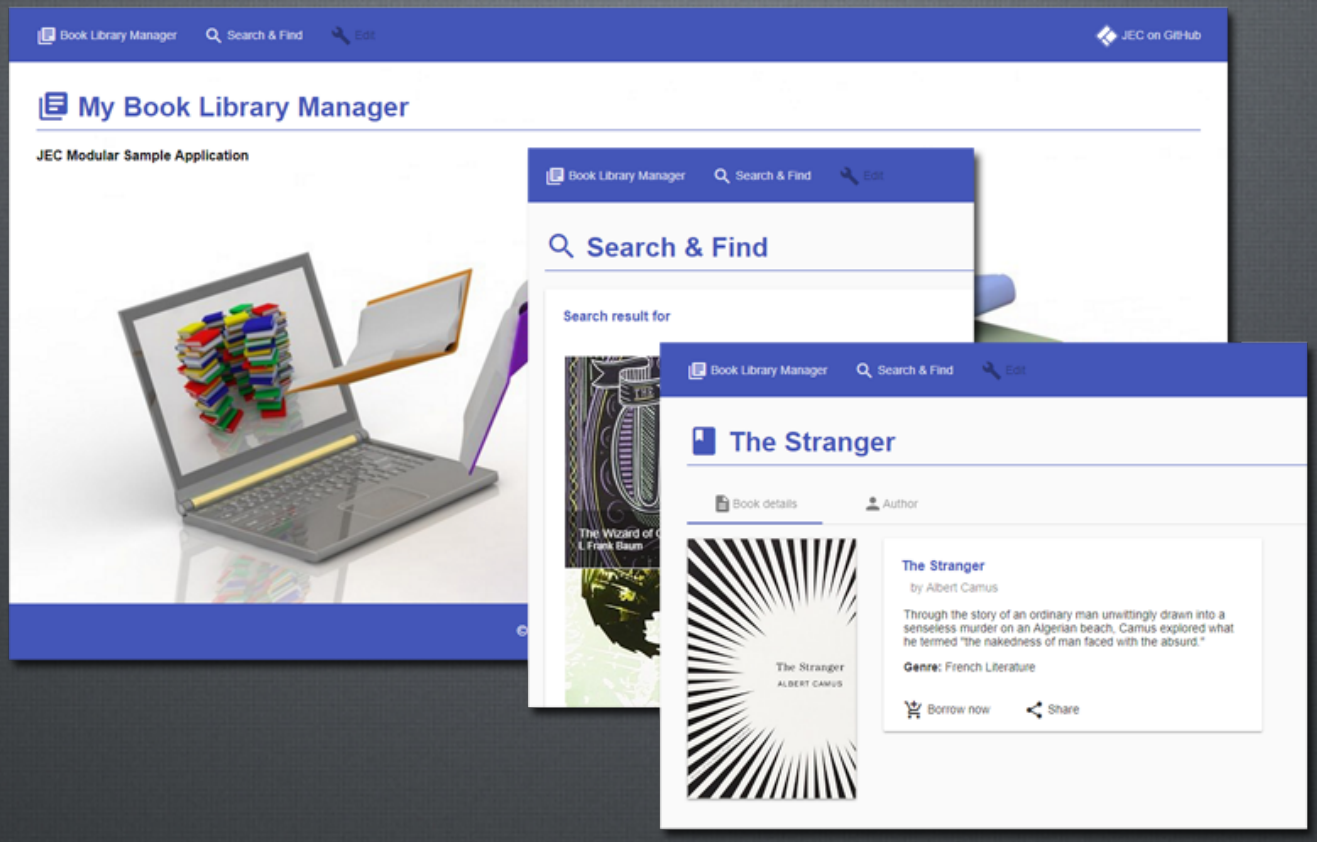
JEC and GlassCat

to easily build modern scalable applications.

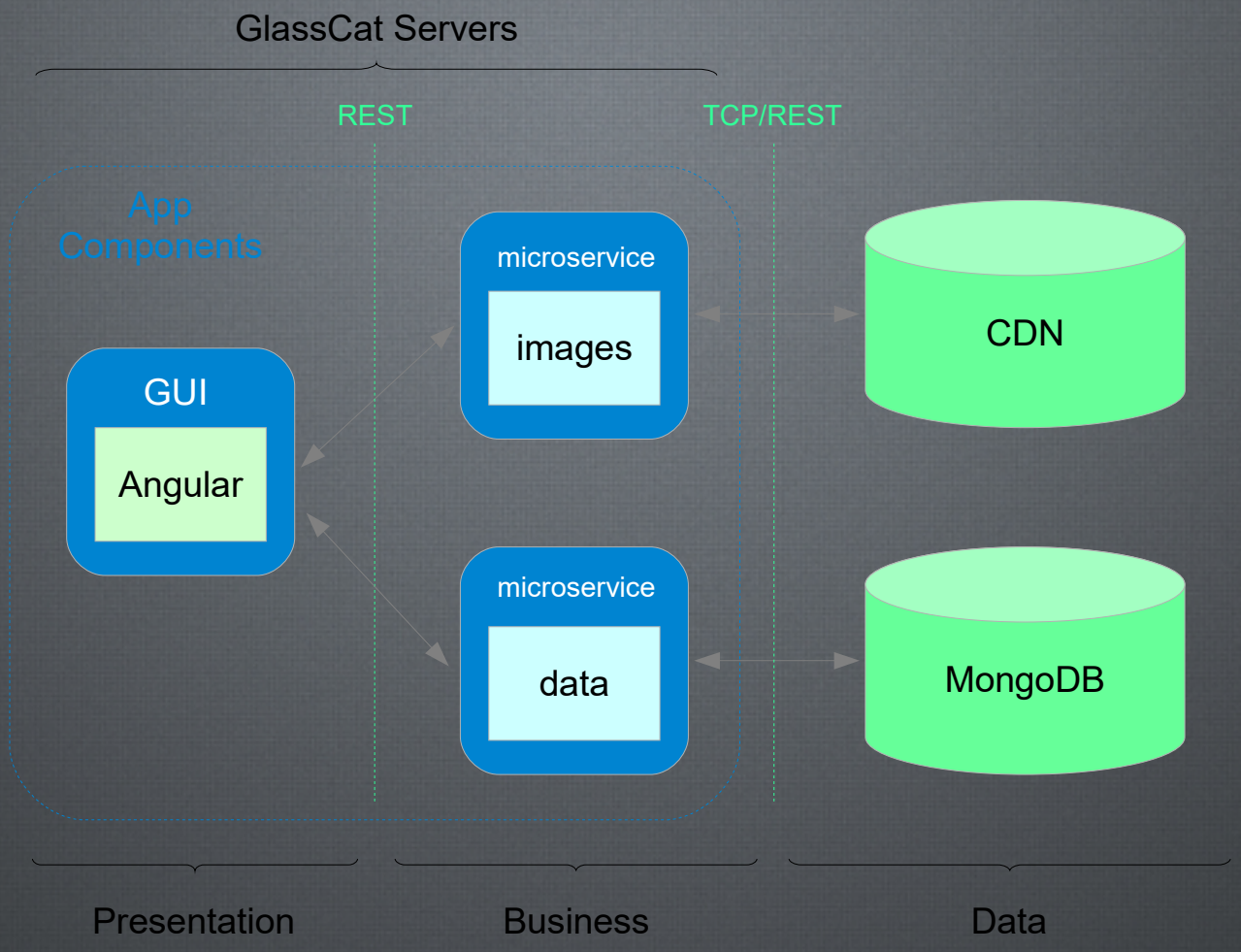
Source Code

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:

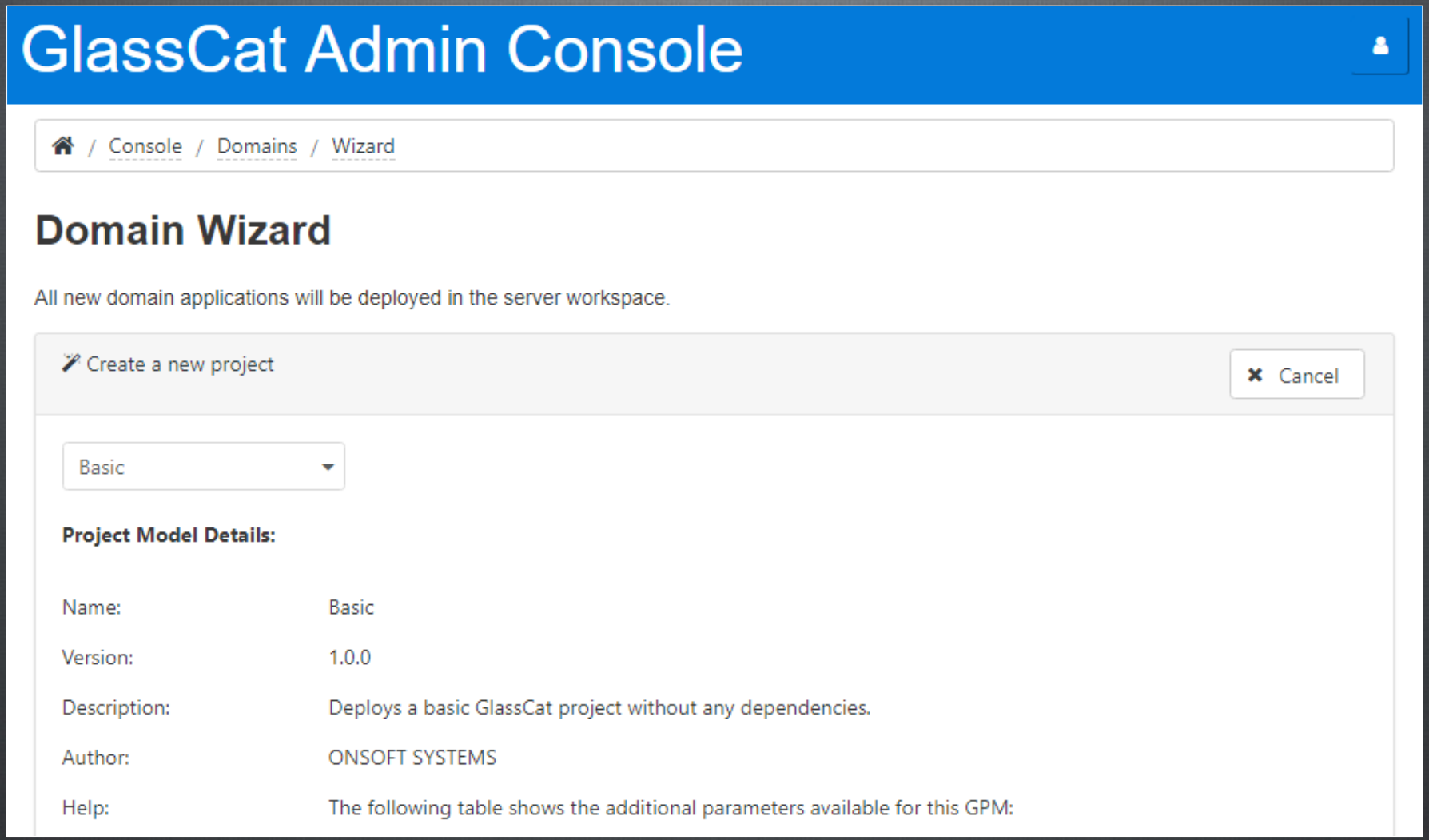
```
$ glasscat archetype --gpm=[basic] --projectName=[myProject]
                     --directory=[myDirectory] --contextRoot=[myContextRoot]
```

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:



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
`basic`	`sample-blm-images`	`sample-blm-images`	`sample-blm-images`
`microservice`	`sample-blm-books`	`sample-blm-books`	`sample-blm-books`
`angular`	`sample-blm-app`	`sample-blm-app`	`sample-blm-app`

Each project is used for specific purpose:

Project	Description
`sample-blm-images`	A microservice app that serves static images.
`sample-blm-books`	A microservice app that provides access to books information through a REST API.
`sample-blm-app`	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".

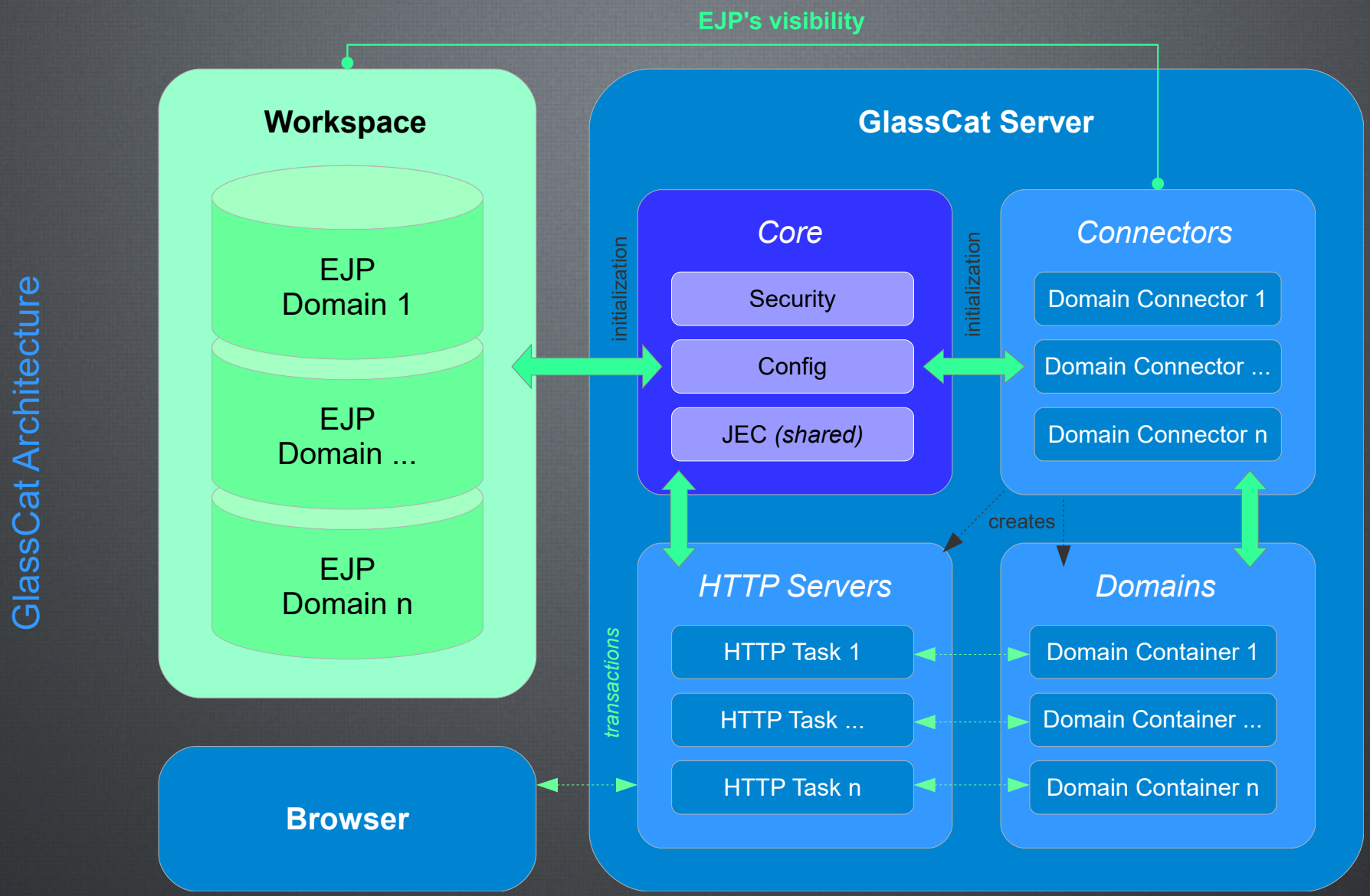
- Contrary to JAVA EARs, domains cannot contain more than one application module.
- JEC does not specify any gateway to communicate from one domain to another.

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.

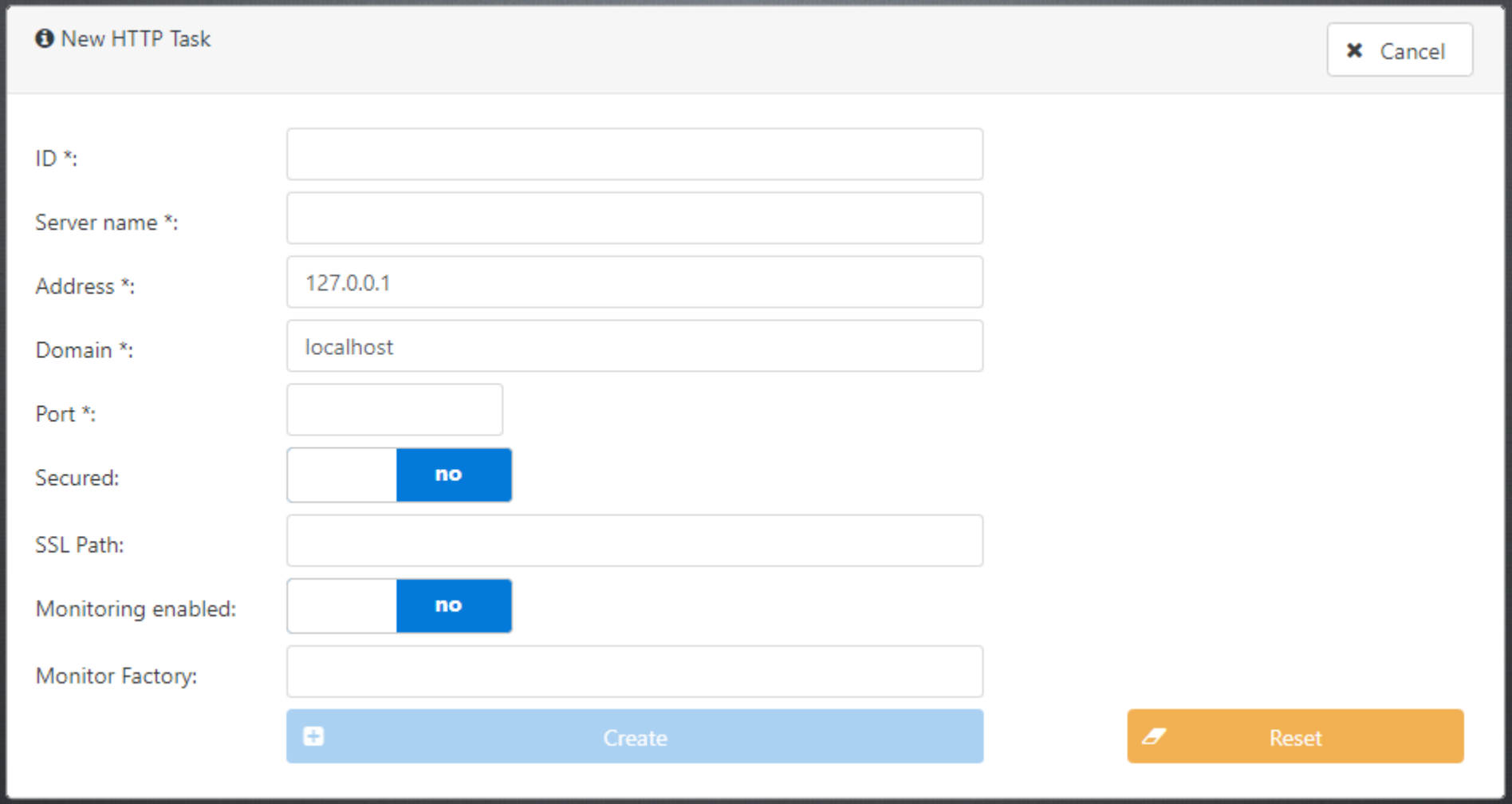
GlassCat architecture facilitates microservices isolation by using connectors and separating HTTP servers (slide #8).

Domains Separation 2/4



Domains Separation 3/4

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



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.[1]

JEC microservices are:

- deployed trhough 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 { HttpHeader } 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(HttpHeader.ACCESS_CONTROL_ALLOW_ORIGIN, "http://localhost:3001");
    res.setHeader(HttpHeader.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("services.BooksDao")
  public dao:BookDao;

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