Dirigible User's Guide v1.2

This User's Guide contains the latest documentation available on-line at:

> http://help.dirigible.io
>
> http://samples.dirigible.io

Enjoy Reading!

Dirigible Help

Dirigible Help Portal gives access to product documentation and also related information. Browse Dirigible Help Portal to get up-to-date information about the features, life-cycle aspects, tip-and-tricks and many more...

Project

Dirigible is an open source project that provides Integrated Development Environment as a Service (IDEaaS) as well as the runtime containers integration for the running applications. The applications created with Dirigible comply with the Dynamic Applications concepts and structure.

The main goal of the project is to provide all the required capabilities needed to develop and run an end-to-end, and yet meaningful, vertical scenario in the cloud for shortest time ever.

The environment itself runs directly in the browser, therefore does not require additional downloads and installations. It packs all the needed containers, which makes it self-contained and well integrated software bundle that can be deployed on any Java based Web Server such as Tomcat, Jetty, JBoss, etc.

The Dirigible project came out of an internal [SAP](http://www.sap.com] initiative to address the extension and adaptation use-cases around SOA and Enterprise Services. On one hand in this project were implied the lessons learned from the standard tools and approaches so far and on the other hand, there were added features aligned with the most recent technologies and architectural patterns related to [Web 2.0] (http://en.wikipedia.org/wiki/Web_2.0] and [HTML5](http://en.wikipedia.org/wiki/HTML5], which made it complete enough to be used as the only tool and environment needed for building and running on demand application in the cloud.

From the beginnig the project follows the principles of Simplicity, Openness, Agility, Completeness and Perfection which provides a sustainable environment where with minimal effort is achieved maximum impact.

Features section describes in detial what is included in the project. Concepts section gives you an overview about the internal and the chosen patterns. Samples shows you how to start and build your first dynamic application in seconds.

Dynamic Applications

We introduced the term *Dynamic Applications* as one that narrows the scope of the target applications, which can be created using *Dirigible*. The overall process of building Dynamic Applications lies on well-known and proved principles:

- In-System Development - known from microcontrollers to business software systems. Major benefit is working on a live system where all the changes made by a developer take effect immediately, hence the impact and side-effects can be realized in very early stage of the development process.
- *Content Centric* - known from networking to development processes in context of Dynamic Applications it comprises all the artifacts are text-based models or executable scripts stored on a generic repository (along with the related binaries such as images). This makes the life-cycle management of the application itself as well as the transport between the landscapes (Dev/Test/Prod) straight forward. In addition, desired effect is the ability to setup the whole system, only by pulling the content from a remote source code repository, such as git.
- Scripting Languages - programming languages written for a special run-time environment that can interpret (rather than compile) the execution of tasks. Dynamic languages existing nowadays as well as the existing smooth integration in the web servers make possible the rise of the in-system development in the cloud.
- *Shortest turn-around time* - instant access and instant value became the most important requirement for the developers, that's why it is also the driving principle for our tooling.

In general, a Dynamic Application consists of components, which can be separated to the following categories:

- *Data Structures* These are the artifacts representing the domain model of the application. In our case, we have choosen the well accepted JSON format for describing the normalized entity model. There is no intermediate adaptation layer in this case, hence all the entities represent directly the database artifacts - tables and views.
- *Entity Services* Once we have the domain model entities, next step is to expose them as web services. Following the modern web patterns we provide the scripting capabilities, so you can create your RESTful services in JavaScript, Ruby and Groovy.
- *Scripting Services* During the development you can use rich set of APIs which give you access to the database and HTTP layer, utilities as well as direct Java APIs underneath. Support for creating unit tests is important and it is integrated as atomic part of the scripting support itself - you can use the same language for the tests, which you are using for the services themselves.
- *User Interface* Web 2.0 paradigm as well as HTML5 specification bring the web user interfaces on another level. There are already many cool client side ajax frameworks, which you can be used depending on the nature of your application.
- *Integration Services* Following the principle of atomicity, one Dynamic Application should be as self-contained as possible. Unfortunately, in the real world there are always some external services that have to be integrated to your application - for data transfer, triggering external processes, lookup in external sources, etc. For this purpose we provide capabilities to create simple routing services and dynamic EIPs.
- *Documentation* The documentation is integral part of your application. The target format for describing services and overall development documentation is already well accepted - wiki.

Architecture

Dirigible architecture follows the well proved principles of simplicity and scalability in the classical service-oriented architecture.

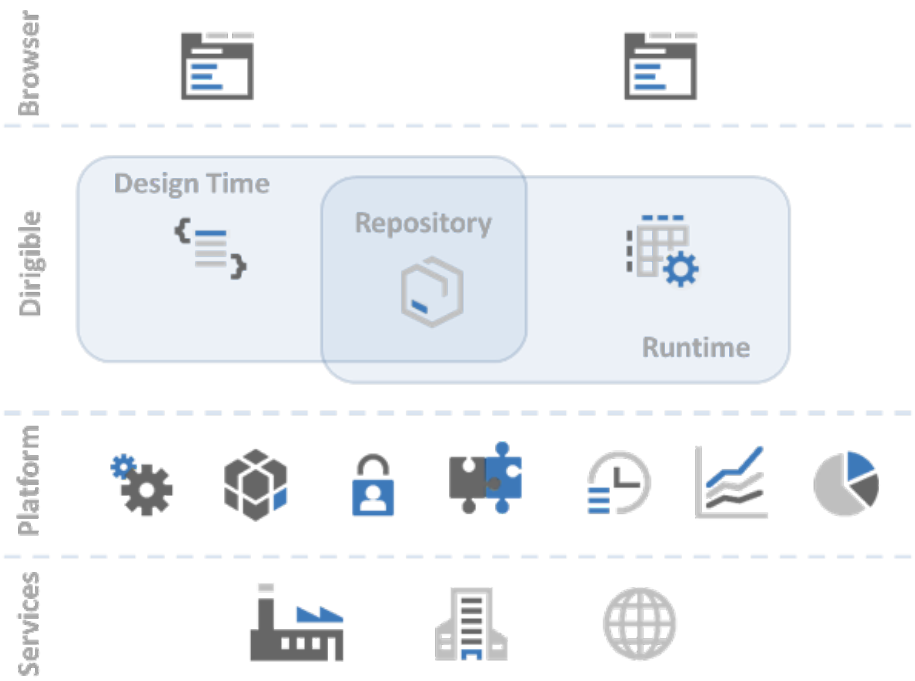
There is separation of components for design time (definition work, modeling, scripting) and the runtime (execution of services, content provisioning and monitoring). The transition between the design time and runtime is via repository component and the only linking part is the content itself.

At design time the programmers and designers use the Web-based integrated development environment. This tooling is mainly based on the Remote Application Platform (RAP) from Eclipse. Using this robust and powerful framework the tooling itself can be easily enhanced by using well known APIs and concepts - SWT, JFaces, OSGi, extension points, etc.

The Repository is the container of the project artifacts. It is a generic file-system like content repository on relation database.

After the creation of the cloud application, it is provided by the runtime components. The main part is Java Web Profile compliant application server. On top are the Dirigible's containers for services execution depending on the scripting language and purpose - Rhino, jRuby, Groovy, Camel, CXF, etc. The runtime can scale independently by the design time part, and even can be deployed without design time at all (for productive landscapes)

Depending on the target cloud platfrom, Dirigible can be integrated to use the services provided by the underlying platfrom.



API

There are several predefined injected objects, which can be used directly from the script code:

- *context* - a standard Map that can be used as a context holder during the execution
- *out* - the standard System output, which is redirected to the trace file
- *datasource* - the default JDBC Datasource configured at server instance level
- *db* - an utillity object with methods:
    - int createSequence(String sequenceName, int start)
    - int getNext(String sequenceName)
    - int dropSequence(String sequenceName)
    - boolean existSequence(String sequenceName)
    - String createLimitAndOffset(String limit, String offset) - used for paging
    - String createTopAndStart(int limit, int offset) - used for paging
- *request* - the standard HttpServletRequest object
- *response* - the standard HttpServletResponse object
- *user* - the name of the current logged in user
- *repository* - the reference to Dirigible Content Repository
- *io* - Apache Commons IOUtils
- *http* - Apache Commons Http Client wrapped utility object with methods:
    - HttpGet createGet(String strURL)
    - HttpPost createPost(String strURL)
    - HttpPut createPut(String strURL)
    - HttpDelete createDelete(String strURL)
    - DefaultHttpClient createHttpClient()
    - void consume(HttpEntity entity)
- *base64* - Apache Commons Codecs Base64
- *hex* - Apache Commons Codecs Hex
- *digest* - Apache Commons Codecs Digest
- *xss* - Apache Commons StringEscapeUtils
- *upload* - Apache Commons File Upload Servlet
- *url* - URLEncode and URLDecoder wrapped in an object with methods:
    - void encode(String s, String enc) and
    - void decode(String s, String enc)
- *uuid* - Universally Unique Identifier (UUID) 128-bit generator
- *input* - the object representing a Message body in case of Route step
- *extensionManager* - the object holding the extension points and extensions meta-data.
- *indexer* - utility uses Apache Lucene. *getIndex([index name])* returns a CustomMemoryIndexer instance
- *wiki* - Confluence to HTML converter utility. *toHtml([confluence text])* render it as HTML as also shown here.
- *storage* - simple binary storage - *put(path, data)*, *get(path)*, *clear()* and *delete(path)* are supported
- *xml* - XML to JSON and vice-versa - *toJson(xmlString)* and *fromJson(jsonString)*

Full Javadoc can be found here

Samples about how to use the APIs can be found here

Features

- *Data Structures*
  - Creation of *Table Model* (JSON formatted *.table descriptor) and actual creation of the corresponding database table during activation.
  - Creation of *View Model* (JSON formatted *.view descriptor) and actual creation of the corresponding database view during activation.
  - Creation of Delimiter Separated Values *data files* (*.dsv) and populating the corresponding database table during activation.
  - *Importing of data* files (*.dsv) on the fly as direct update to corresponding table.
  - *Automatic altering* of existing tables from the models on compatible changes (adding new columns)
- *Extension Definitions*
  - Creation of *Extension Points* (JSON formatted descriptor)
  - Creation of *Extensions* by a given extension point (JSON formatted descriptor)
- *Scripting Services*
  - Support of *JavaScript* language by using Mozilla Rhino as runtime container (*.js)
  - Support of *CommonJS* based *modularization* of JavaScript services (*.jslib)
  - Support of *Ruby* language by using jRuby as runtime container along with the standard for the language modularization
  - Support of *Groovy* language by using Groovy as runtime container along with the standard for the language modularization
  - Support of predefined API as *injected global objects* such as request, response, datasource, httpclient, repository, etc. for all supported languages
- *WebContent*
  - Support of *client side web* related artifacts such as html, css, js, pictures
- *WikiContent*
  - Support of *Confluence* format of Wiki pages
  - Support of customizable *header, footer and css* for wiki pages
- *Integration Services*
  - Support of dynamic routes by using Apache *Camel*
  - Support of *JavaScript* breakouts in routes
- *Tooling*
  - *Workspace* perspective for full support of project management (new, cut, copy, paste, delete, refresh, etc.)
  - *Database* perspective for RDBMS management including SQL Console
  - Enhanced *Code Editor* with highlight support for JavaScript, Ruby, Groovy, HTML, JSON, XML, etc.
  - *Preview* for easy testing of changes in web, wiki and scripting services
  - Configurable *Log Viewer* providing the server side logs and traces
  - Lots of *template based wizards* for creating new content and services
  - *Import* and *Export* of project(s) content
  - *Import of binary* files for external documents and pictures
  - *Repository* perspective for low level repository content management
- *Security*
  - *Role based* access management
  - *Security Constraints Model* (JSON formatted *.access) support
  - Few predefined roles which can be used out-of-the-box Everyone, Administrator, Manager, PowerUser, User, ReadWrite, ReadOnly
- *Registry*
  - *Activation* support - exposing the artifacts from the user's workspace publicly
  - *Auto-Activation* support for usability
  - User-Interface for *Browsing and Searching* of the activated content
  - Separate *lists of endpoints* and viewers per type of services - JavaScript, Ruby, Groovy, Routes
  - Separate browse user interface for *web and wiki* content

and more...

Data Structures

Overview

Data Structures term in the context of the cloud toolkit is used for refering the Domain Model of the application. For pragmatic reasons it is chosen that the actual entities in the domain model to correspond 1:1 to the underlying database entities - tables and views. There is no additional abstract layer between your application code and the actual model in target storage.

Tables

Table Model is a JSON formatted *.table descriptor which represents the layout of the database table which will be created during activation process.

Example descriptor:

```
{
  "tableName":"TEST001",
  "columns":
    [
      {
        "name":"ID",
        "type":"INTEGER",
        "length":"0",
        "notNull":"true",
        "primaryKey":"true",
        "defaultValue":""
      }
      ,
      {
        "name":"NAME",
        "type":"VARCHAR",
        "length":"20",
        "notNull":"false",
        "primaryKey":"false",
        "defaultValue":""
      }
      ,
      {
        "name":"DATEOFBIRTH",
        "type":"DATE",
        "length":"0",
        "notNull":"false",
        "primaryKey":"false",
        "defaultValue":""
      }
      ,
      {
        "name":"SALARY",
        "type":"DOUBLE",
        "length":"0",
        "notNull":"false",
        "primaryKey":"false",
        "defaultValue":""
      }
    ]
}
```

The supported database types are:

- *VARCHAR* - for text based fields up to 2K characters
- *CHAR* - for text based fields with fixed lenght up to 255 characters
- *INTEGER* - 32 bit
- *BIGINT* - 64 bit
- *SMALLINT* - 16 bit
- *REAL* - 7 digits of mantissa
- *DOUBLE* - 15 digits of mantissa
- *DATE* - represents a date consisting of day, month, and year
- *TIME* - represents a time consisting of hours, minutes, and seconds
- *TIMESTAMP* - represents DATE plus TIME plus a nanosecond field and time zone
- *BLOB* - binary object - images, audio, etc.

Activation of table descriptor is a process of creation of the database table in the target database. The activator constructs a "CREATE TBALE" SQL statement considering the dialect of the target database system. If the table with the given name already exists, the activator checks whether there is a compatible change (adding of new columns) and construct "ALTER TABLE" SQL statement. If the there is an incompatible change, then the activator returns an error which have to be solved manually via the SQL Console.

Views

View Model is also JSON formatted *.view descriptor of a database view. Usually it is a join between multiple tables used for reporting purposes. The script should follow the SQL92 standard, or have to be aligned with the dialect of the target database.

Data Files

Delimiter Separated Values *data files* *.dsv are used for importing some test data during the development or for defining a static content for some nomenclatures for instance. The convention is that the name of the data file should be the same as the name of the target table. The delimiter is the char "|" and the order of the data fields should be the same as the natural order of the target table. If you want to import some data only once, this can be done via the Import Data Wizard.

Be careful when using the static data in tables. Entity Services (generated by the templates) are using sequence algorithm for identity column starting from 1.

The automatic re-initialization of the static content from the data file can be achieved when you create a *.dsv file under the DataStructures folder of the given project.

Scripting Services

JavaScript

Services

Primary language used to implement services in Dirigible is JavaScript. Being quite popular as a client-side scripting, it became also prefered language for server-side business logic as well. For the underlying execution engine is used the most mature JavaScript engine written in Java - Rhino by Mozilla. You can write your algorithms in *.js files and store them within the ScriptingServices folder. After the Activation or Publishing they can be executed by accessing the endpoint respectively at the sandbox or public registry.

An example JavaScript service looks like this:

```
var systemLib = require('system');

var count;
var connection = datasource.getConnection();
try {
    var statement = connection.createStatement();
    var rs = statement.executeQuery('SELECT COUNT(*) FROM BOOKS');
    while (rs.next()) {
        count = rs.getInt(1);
    }
    systemLib.println('count: '  + count);
} finally {
    connection.close();
}

response.getWriter().println(count);
response.getWriter().flush();
response.getWriter().close();
```

This example shows two major benefits:

- Modularization based on built-in CommonJS ('require' function on the first line)
- Native usage of the Java objects as API injected in the execution context (database, response)

Libraries (Modules)

You can create your own library modules in *.jslib files. Just do not forget to add the public parts in the *exports*.

```
exports.generateGuid = function() {
    var guid = uuid.randomUUID();
    return guid;
};
```

> The libraries are not directly exposed as services, hence they do not have accessible endpoints in the registry.

The reference of the library module from the service is done by using the standard function *require()* where the parameter is the location of the module constructed as follows: / Module path includes the full path to the module in the project structure without the predefined folder ScriptingServices and also without the extension *.jslib.

```
/sample_project
    /ScriptingServices
        /service1.js
        /library1.jslib

library1.jslib is refered in service1.js:
...
var library1 = require('sample_project/library1');
...
```

> Relative paths ('.', '..') are not supported. The project name must be explicitly defined.

Ruby

Language which also expanding its popularity in web development scenarios last years is Ruby. You can use also the standard modularization provided by the language as well as the injected context objects in the same way as in JavaScript. The execution engine used as runtime container is jRuby

Example service which has reference to a module can be generated from the Scripting Services wizard directly:

Service (sample.rb):

```
require "/sample_project/module1"
Module1.helloworld("Jim")
```

and Module (module1.rb):

```
module Module1
  def self.helloworld(name)
    puts "Hello, #{name}"
    $response.getWriter().println("Hello World!")
  end
end
```

> Note that in Ruby you have to put a dollar sign ('$') in the beginning of the API objects ($response) as they are global objects

Groovy

Groovy is yet another powerful language for web development nowadays with its static types, OOP abilities and many more.

Corresponding examples in Groovy:

Service (sample.groovy):

```
import sample_project.module1;



def object = new Module1();
object.hello(response);
```

Module (module1.groovy):

```
class Module1{
    void hello(def response){
        response.getWriter().println("Hello from Module1")
    }
}
```

Integration Services

Overview

Integration Services are the connection points between the application logic and the external services - 3-thd party cloud services, On-Premise services, public services, etc. There are support for in-bound as well as the out-bound services - consumption and provisioning. By utilizing one of the most mature and well known framework as underlying technology - Apache Camel, there are lots of ready-to-use integration patterns.

Routes

The term *Route* is directly taken from the Apache Camel's context and refers to *definition of routing rules*. The extension of the definition file is "*.routes"

Sample route descriptor looks like this:

```
< routes xmlns="http://camel.apache.org/schema/spring">
    < route id="simple_routing">
        < from uri="servlet:///simple_routing_endpoint" />
        < choice>
            < when>
                < header>name_parameter
                < transform>
                    < simple>Hello ${header.name_parameter} how are you?
                < /transform>
            < /when>
            < otherwise>
                < transform>
                    < constant>Add a name parameter to uri, eg ?name_parameter=foo
                < /transform>
            < /otherwise>
        < /choice>
    < /route>
< /routes>
```

The original source is here

Once you [activate|activation.wiki] or [publish|publishing.wiki] the project the route descriptor is read by the runtime agent and the route is enabled in the container. The end-point of such an integration service is exposed by the Camel container at the location constructed by the following pattern:

The pattern for the endpoint location of routes:

> *http //[host]:[port]/dirigible/camel/[servlet name - (from uri="servlet:///XXX")]*

e.g.

> *http //[host]:[port]/dirigible/camel/simpleroutingendpoint*

More information about the supported integration patterns can be found at the samples portal.

Web Content

Overview

Web Content includes all the static client-side resources such as HTML files, CSS and related theming ingredients as well as dynamic scripts (e.g. JavaScript) and images. In general, the web content adapter is playing a role of a tunnel, which takes the desired resource location from the request path, loads the corresponding content from the repository and send it back without any modification.

The default behavior of the adapter on a request to a collection (instead of particular resource) is to send back an error code to indicate that the listing of folders is forbidden.

> If the specific "application/json" *Accept* header is supplied with the request itself, then a JSON formatted array with the sub-folders and resources will be returned.

Templating

Common pattern in user interfaces of web based business applications is simplified templating - usually static header and footer. To support this we introduced a special handling of HTML pages:

- *header.html* is a special page, which is recognized as a static header, so that, if exists, it is rendered in the beginning of a requested regular page
- *footer.html* is a special page, which is recognized as a static footer, so that, if exists, it is rendered in the end of a requested regular page
- *index.html* is a special page, which is recognized as a welcome page, so that no header and footer are added to it
- *nohf* is a parameter, which can be added to the request URL to disable adding of header and footer

To boost the developer productivity in the most common cases, we provide a set of templates, which can help in user interface creation. There are set of templates, which can be used with entity services - list of entities, master-detail, input form, etc. More information can be found at samples area. The other templates can be used as utilities e.g. for creation application shell in index.html with main menu or as samples showing most common controls on different AJAX user interface frameworks - jQuery, Bootstrap, AngularJS, OpenUI5.

Wiki Content

Overview

An integral part of every application is the user documentation. For this purpose we introduced a special type of artifacts which are placed in a predefined sub-folder of a project. This type of artifacts follows the de-facto standard nowadays format for documenting behaviors and algorithms of applications as well as general information about the program itself - wiki. The supported markup language as of now is Confluence - well accepted by the community.

The wiki pages have to be placed under WikiContent folder of a project with *.wiki file extension. Once they are requested by GET request, underground transformation has been triggered which convert the confluence format to HTML and send the well formed web content back.

Sample of a wiki page in confluence format looks like as following:

```
h4. Confluence Markup

Ideally, the markup should be _readable_ and even *clearly understandable* when you are
editing it. Inserting formatting should require few keystrokes, and little thought.

After all, we want people to be concentrating on the words, not on where the angle-brackets
should go.

* Kinds of Markup
** Text Effects
** Headings
** Text Breaks
** Links
** Other
```

and after the rendering you will get:

Confluence Markup

Ideally, the markup should be *readable* and even *clearly understandable* when you are editing it. Inserting formatting should require few keystrokes, and little thought.

After all, we want people to be concentrating on the words, not on where the angle-brackets should go.

- Kinds of Markup
  - Text Effects
  - Headings
  - Text Breaks
  - Links
  - Other

Templating

Simple templating is also supported similar to web content:

- *header.html* is a special page, which is recognized as a static header, so that, if exists, it is rendered in the beginning of a requested regular page
- *footer.html* is a special page, which is recognized as a static footer, so that, if exists, it is rendered in the end of a requested regular page
- *nohf* is a parameter, which can be added to the request URL to disable adding of header and footer

Sample Pages

Sample header and footer as well as navigation page could look like:

- header.html
- footer.html
- navigation.wiki

```
* [Home|index.wiki]
* [Project|project.wiki]
...
```

and of course some custom css for the wiki content

- wiki.css

Batch of Wiki Pages

Sometimes it is helpful to combine several already existing pages to a single page. For this purpose you have to create a file with extension *.wikis and to list in it all the wiki pages that you want to merge.

File: *single.wikis*

```
part1.wiki
part2.wiki
```

Extension Definitions

Extensibility is important requirement for the business applications built to follow the custom processes in LoB areas. There are well known patterns already in the most popular languages and frameworks such as plugins in Eclipse, BAdIs in ABAP, services in Java, etc. In the toolkit it has been choosen the simplest yet most powerful way to define extensions. It provides just a generic description of the extension points and extensions, without explicitly define the contract.

h3. Extension Points The Extension Point is the place in the core module, where it is expected to be enhanced by the custom created modules. It is a simple JSON formated file with extension *.extensionpoint* placed in the project folder "Extension_Definitions".

```
{
  "extension-point":"/project1/extensionPoint1",
  "description":"description for the extension point 1"
}
```

h3. Extensions The Extension is the actual plugin in the custom module, which extends the core functionality It is also a simple JSON formated file with extension *.extension* in the same folder.

```
{
  "extension":"/project1/extension1",
  "extension-point":"/project1/extensionPoint1",
  "description":"description for the extension 1"
}
```

```
The *"extension"* parameter above should point to a valid [Scripting Service](scripting_services.html) in the same language.
```

Calling Extensions

Within the core module you can iterate over the defined extensions and call theirs functions:

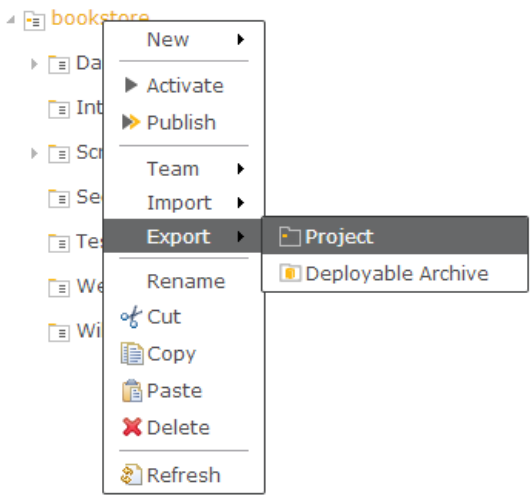
```
var extensions = extensionManager.getExtensions("/project1/extensionPoint1");
for (var i=0; i < extensions.length; i++) {
    var extension = require(extensions[i]);
    response.getWriter().println(extension.enhanceProcess());
}
```

In the code above the extension is a JavaScript Service Library (*extension1.jslib*) within the same project, which has exposed function *enhanceProcess()*

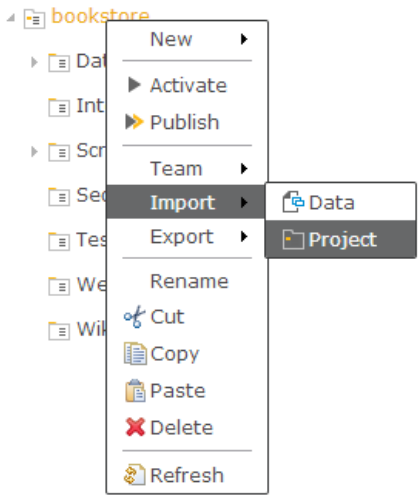
Backup - Project Import and Export

There is a possibility to export and import content of a given project or multiple projects as a zip archive for e.g. backup purposes. This is the simplest yet fastest way to transfer project content to Dirigible. The actions are available thru the main or pop-up menu

Export



and Import

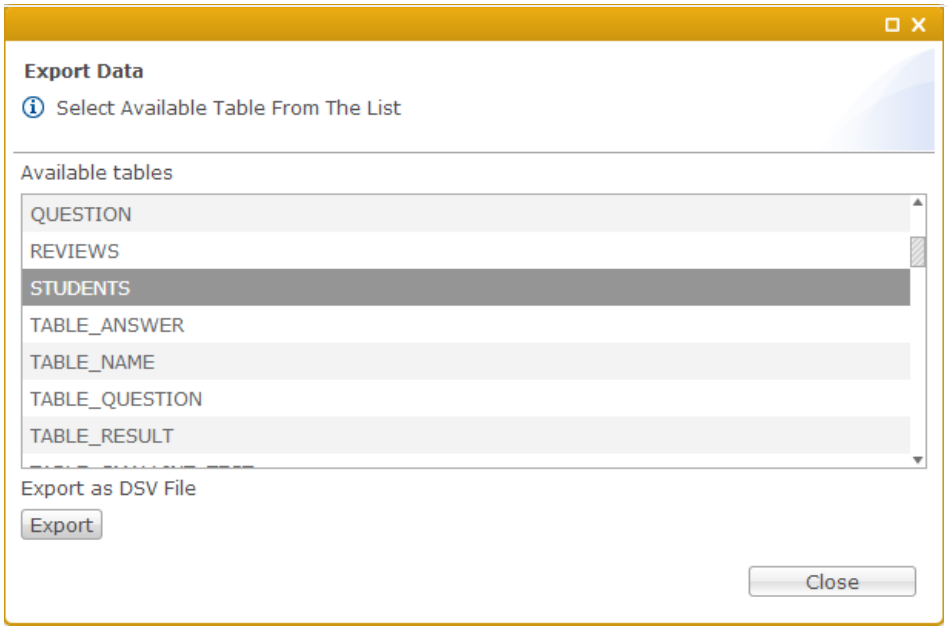


> You can use this functionality also in cases of mass import of project artifacts - e.g. images in the web content

This feature can be used also for import of sample projects from public sources or sharing of projects between accounts, although for real team source management the recommended approach is via Git.

Backup Data

Except creating backup of the project it is good practice to have backup of the content. This can be easily done by exporting tables data into *.dsv* files.


Export Data
Select Available Table From The List
Available tables
QUESTION
REVIEWS
STUDENTS
TABLE_ANSWER
TABLE_NAME
TABLE_QUESTION
TABLE_RESULT
Export as DSV File
Export
Close

Test Cases

Overview

Following the best practices unit tests are always integral part of the application code itself. For this reason in the cloud toolkit there is a predefined place for them and kind of deep integration with the scripting services. The language supported for test cases is the same as of the target scripting service you want to tests. Technically the unit tests are not different than the services, hence the separation in this regard is only semantical. You can code in the test cases everything as you can do also in scripting services. All the API objects and context as well as usage of libraries are supported transparently.

Writing a Test Case

You start by using the action from the pop-up menu while selecting a project. Choose the New->Test Case and select from the list of the predefined templates. You can use the following APIs which can help you to write more standardized yet comprehensible unit tests:

- *assertTrue*
  - message - the error message
  - condition - the condition usually containing the inspecting values
- *assertFalse*
  - message - the error message
  - condition - the condition usually containing the inspecting values
- *assertEquals*
  - message - the error message
  - o1 - the expecting object
  - o2 - the actual object
- *assertNull*
  - message - the error message
  - o - the inspecting object
- *assertNotNull*
  - message - the error message
  - o - the inspecting object
- *fail*
  - message - the error message

Some sample code should look like:

```
var assert = require('assert');
...
assert.assertNotNull('value is null', value);
...
```

For more comprehensive sample you can refer to sample test case.

Security

Security is quite broad topic, so that here will be described mostly the authentication and authorization concept of dynamic applications.

Being a standard Java web application, the Dirigible Design-Time as well as Runtime components rely entirely on the underlying Java web container/server for the authentication process. Usually it comes as well integrated JAAS service.

There are several predefined roles coming by default and can be used for dynamic applications:

- Administrator
- Manager
- PowerUser
- User
- ReadWrite
- ReadOnly
- Everyone

> Have in mind that the above roles are shared for all the projects/dynamic applications within the account More roles can be added only via custom build of the Dirigible's Runtime component.

As soon as the Roles definition are well standardized, User/Principals to Roles assignments are platform specific. For HANA Cloud Platfrom you can refer at SAP*HANA*Cloud.pdf section - 1.3.10.3.1

Once we have defined the Roles as well as the User-to-Roles assignments, it comes the definition of the protected resources. It is done by a simple JSON formatted *.access file under the SecurityConstraints project's sub-folder. There is a wizard which generates the sample main.access, which looks like:

```
[
  {
    "location":"/project1/secured",
    "roles":
      [
        {"role":"User"},
        {"role":"PowerUser"}
      ]
  },
  {
    "location":"/project1/confidential",
    "roles":
      [
        {"role":"Administrator"}
      ]
  }
]
```

After the publishing of the project you can try to access the protected resources with users with different roles assignments. The impact of protection of the resources is on the web and wiki content and also on scripting services' end-points.

> The location attribute of the protected resource is transitive i.e. the most specific location wins in case of multiple definitions with equal roots

There is a Security Manager view, opened by default in Workspace perspective, where you can see the currently protected resources as well as to disable the protection.

Git Integration

There is a Git connector for team development in the toolkit. The goal is to provided the simplest way to synchronize the sources with the remote source control repository and to leave the more complex operation (e.g. merge) for external tools.

Available commands: * *Clone* - clones remote Git repository to the toolkit as a project

> Constraint: Remote Git repository must contain only one project.

- *Push* - tries to push changes to the project's remote repository. If the changes are not conflicting with what is in remote "origin/master" branch, then the push is successful. After that the project content is synchronized with the state in the remote "origin/master" branch. If there are conflicts with the newly made changes then new remote branch is created and changes are pushed in it. The remote branch's name is "changes*branch*{dirigible's username}", e.g. "changes*branch*user1234".

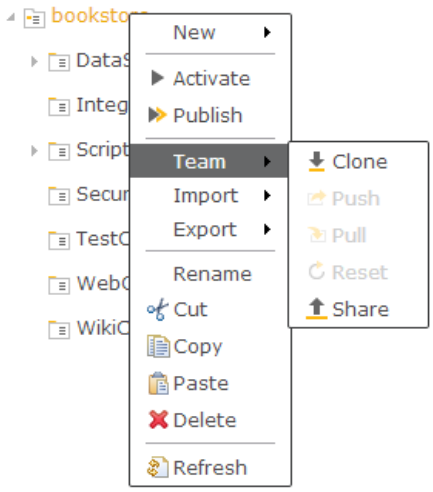
> Constraint: Merging of conflicting branches should be done via an external tool e.g. "GitBash", "eGit", etc.

- *Pull* - checkouts the remote changes from the "origin/master", if there are conflicts then an error will be raised.

> Constraint: If there are conflicting changes during Pull, then the recommendation is to [backup|backup.wiki] the project as zip, then to Reset it and manually apply your changes again.

- *Reset* - sets the local project to be as the latest state of the remote "origin/master" branch.
- *Share* - shares the selected project to remote repository.

All the commands are accessible from the project's pop-up menu



Concepts

There are several must-to-know concepts, which are implied in the toolkit and have to be understood before starting. Some of them are tightly related to the dynamic applications nature and behavior, others just follow the best practices from the services architecture reflected also in the cloud applications.

Isolated workspace for project management is important when we have to share a single server instance to many users. Another related feature is sand-boxing - the way every user to have its own private runtime space where to test his/her services during the development. Registry and the related publishing process are taken from the SOA (UDDI) and recent API Management to bring some of the strengths like discoverability, reusability, loose coupling, relevance, etc. To boost the development productivity at the initial phase, we introduced template based generation of application artifacts via wizards. The new Web 2.0 paradigm and the leveraged REST architectural style changed the way how the services should behave as well as how to be described. Although, there is a push for bilateral contracts only and free description of the services, we decided to introduce a bit more sophisticated kind of services for special purpose - Entity Services.

Workspace

The Workspce is the developer's place where he/she creates and manages the application artifacts. The first level entities are the projects them selves. Each project contains several system folders based on the type of the artifacts. Default ones are:

- DataStructures - containing the tables, views, etc. database related artifacts (more info here
- IntegrationServices - containing the definitions about the routes and related artifacts (more info here
- ScriptingServices - containing the JavaScript, Ruby, Groovy, etc. server-side services and related artifacts (more info here
- SecurityConstraints - containing the access control definitions artifacts (more info here
- TestCases - containing the unit tests for the scripting services and related artifacts (more info here
- WebContent - containing the static pages as well as the client-side scripting artifacts (more info here
- WikiContent - containing the confluence formatted wiki pages and related artifacts (more info here

Example layout of a project looks like this:

```
/db
  /dirigible
    /users
      /            (private space)
        /workspace
          /project1
            /DataStructures
              /data1.table
              /data1.dsv
            /IntegrationServices
              /connector1.routes
            /ScriptingServices
              /service1.js
            /SecurityConstraints
              /main.access
            /TestCases
              /service1_test.js
            /WebContent
              /index.html
              /default.css
            /WikiContent
              /project1.wiki
              /license.wiki
```

The project management can be done via the views and editors in the workspace perspective.

Generation

Template based generation of artifacts helps for developer productivity in the initial phase of building the application. There are several application components, which have similar behavior and often very similar implementation. A prominent example is Entity Service. It has several predefined methods based on REST concepts and HTTP - GET, POST, PUT, DELETE on an entity level as well as list of all entities. On the other hand, the most notable storage for the entity's data is the RDBMS provided by the platform. Hence to save the developer's effort to create such entity services from scratch we introduced a template, which can be used for generation from table to service. Another example is user interface templates based on patterns - list, master-detail, input form, etc. There can be provided also templates based on different frameworks for client-side interaction.

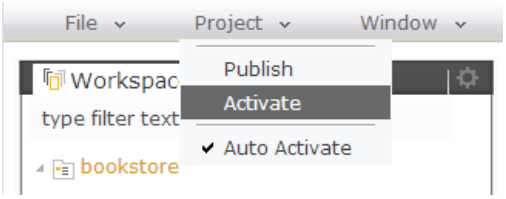
> The generation is one-time process. Once you have the generated artifact you can modify it based on your own requirements.

In contrast of the MDA, where you can expect to regenerate the PSMs every time you make changes on PIMs, we decided to choose the more pragmatic approach - single model. In general it is good to have an abstraction and technology agnostic languages and models, but in practice last years it turned out that it brings more problems than the benefits especially for support - where MDA claims to be good for.

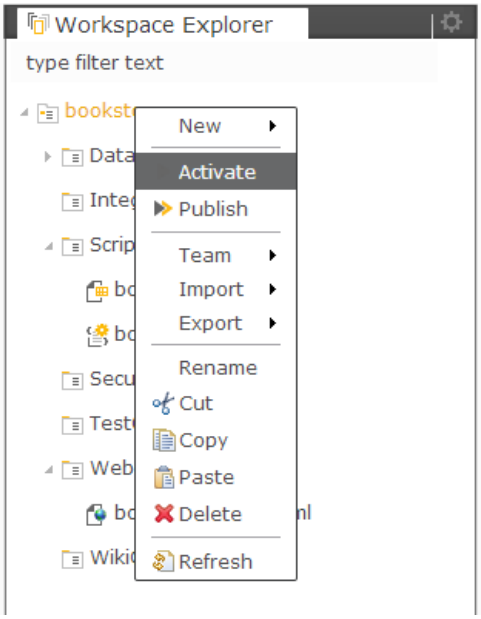
Activation

Activation is a concept related to development life-cycle of the application. The original sources are stored in the workspace of the user. All the changes reflect direclty the source artifacts there. When the source artifact is already in the state, which can be executed (i.e. tested) the developer has to perform *activation* on the project level. This will transfer (copy) the source artifacts from the workspace to the *sandbox*. This place is fully functional runtime container, isolated for the current user only. The difference between the *sandbox* and the *registry* space is the user isolation only.

Activation action is accessible from the main menu under the Project section or at the project's pop-up menu in the Workspace Explorer



or




```
/db
  /dirigible
    /sandbox
      /             (private space)
        /ScriptingServices
          /project1
            /service1.js
    /users
      /             (private space)
        /workspace
          /project1
            /ScriptingServices
              /service1.js
```

> The scripting services in the sandbox can access the services from the registry, but not vice versa

There is default *auto-activation* mechanism, which can perform the activation on save of the artifact. This can be switched on/off from the main menu -> Project (if you are in the Workspace perspective)

The auto-activation is enabled only for:

- Scripting Services
- Web Content
- Wiki Content

For:

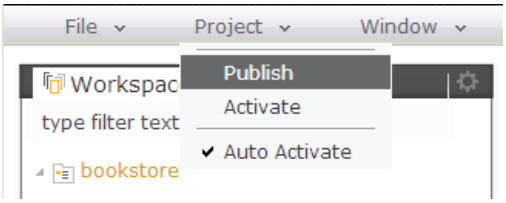
- Data Structures
- Security Constraints
- Integration Services
- Extension Definitions

there is no sandboxing supported as well as auto-activation. The activation process is equal to publication in this case.

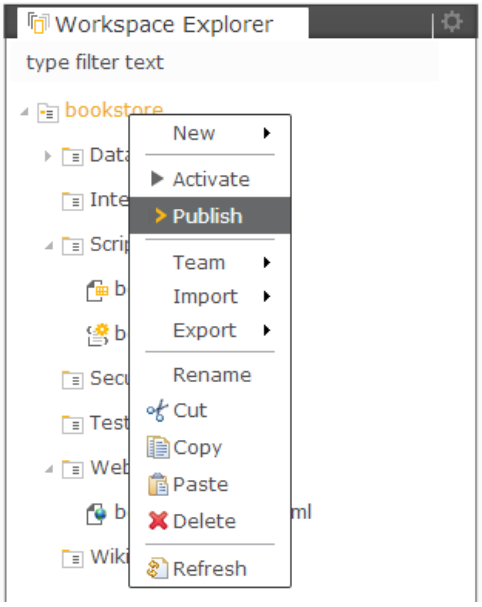
Publishing

There is a conceptual separation between design-time and runtime phases of the development life-cycle. During the design-time phase the source artifacts are created and managed within the isolated developer's area - workspace. When the developer is ready with a given feature, he/she have to publish the project, so that the application artifacts to become available for the users It depends of the type of the artifact the meaning "available". For the Scripting Services for instance it is the registration of a public end-point, for web and wiki content it is just allowed access to the artifacts them self, etc.

Publishing action is accessible from the main menu under the Project section or at the project's pop-up menu in the Workspace Explorer



or



The space within the Repository where all the public artifact are placed is called *Registry*

```
/db
  /dirigible
    /registry             (public space)
      /public             (placeholder)
        /ScriptingServices
          /project1
            /service1.js
    /users
      /             (private space)
        /workspace
          /project1
            /ScriptingServices
              /service1.js
```

To view the currently published artifacts you can go to Registry User Interface. There are separate section by the types of the artifacts.

Entity Service

In general, the Entity Service is fully capable RESTful service as it is defined by REST architecural style for performance, scalability, simplicity, etc. It exposes the CRUD operations of a given domain model object. Underneath it connects the database store as data transfer layer. Working following RESTful paradigm on business software components, soon it turns out that there is a service pattern, which is used very often - domain object management. Having in mind also the past experience from web services, we decided to enhance a bit the standard functionality of such a services without breaking REST principles. These enhancements are useful especially for generic utilities and user interface generation.

First of all let list what we have as a standard:

- *GET* method
  - if the requested path points directly to the service endpoint (no additional parameters) - it lists all the entities of this type (in this collection)
  - if the request contains an *id* parameter - then the service returns only the requested entity
- *POST* method - creates an entity getting the fields from the request body (JSON formatted) and auto-generated id
- *PUT* method - updates the entity getting the id from the request body (JSON formatted)
- *DELETE* method - deletes the entity by the provided id parameter which is mandatory

The enhancements we added to the standard functionality:

- on *GET* as parameters
  - *count* - returns the number of the entities collection size
  - *metadata* - returns the simplified descriptor of the entity in JSON (see below)
  - *sort* - indicate the order of the entities
  - *desc* - indicates reverse order, used with the above parameter
  - *limit* - used for paging, returns limited result set
  - *offset* - used for paging, result set starts from the offset value

Example metadata for an entity

```
{"name":"books","type":"object","properties":
  [
    {"name":"book_id","type":"integer","key":"true","required":"true"},
    {"name":"book_isbn","type":"string"},
    {"name":"book_title","type":"string"},
    {"name":"book_author","type":"string"},
    {"name":"book_editor","type":"string"},
    {"name":"book_publisher","type":"string"},
    {"name":"book_format","type":"string"},
    {"name":"book_publication_date","type":"date"},
    {"name":"book_price","type":"double"}
  ]
}
```

These enhancements we see as the minimal yet simplest valuable extension to the REST. Similar, but more complex specifications, which intentionally we do not included so far are OData and GData.

All these features of entity services are implied during the generation process. The template uses as input a database table and name of the entity service, which are entered in the corrsponding wizard. Just select the *.entity artifact in the Workspace Explorer and use the pop-up menu *Generate->User Interface for Entity Service*.

There are several limitation for the table to be entity service compliant:

- there should be one and only column as primary key, which will be used for its *identity*
- only a set of database column types, which are supported by default for generation (simple types only; clob, blob - not supported)

We do not generate also generic query methods, because on one hand it will cover only very simple cases with reasonable performance, which easiliy can be written anyway as additional methods (by parameters) and on the other hand for the complex queries there is no sense to introduce additional layer, which will not give the desired performance as well in comparison to the well analysed by the developer SQL script.

Entity services are generated used JavaScript language, hence the can be accessed right after the generation and publishing on:

```
*[protocol]://[host]:[port]/[dirigible's runtime application context]/js/[project]/[entity service path]*
e.g.

*https ://dirigibleide.hana.ondemand.com/dirigible/js/bookstore/books.js*
```

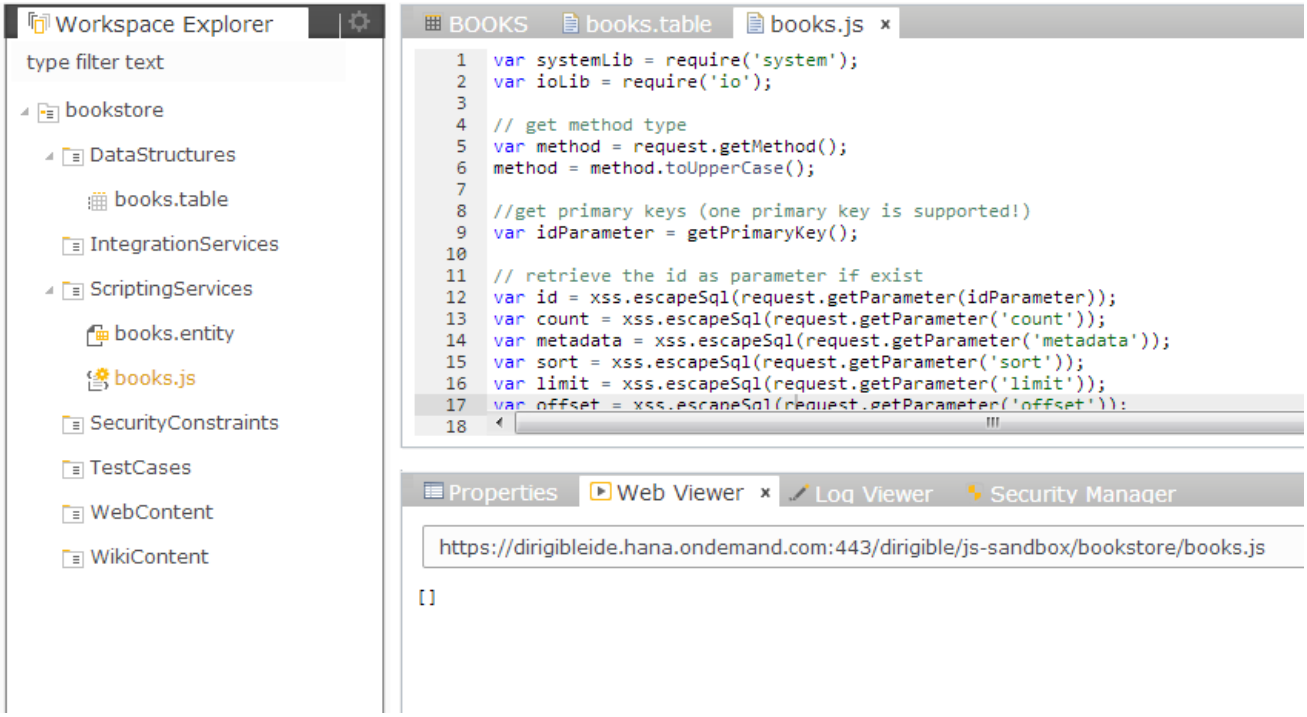
or just select them in Workspace Explorer and see the result in the Preview.

Tooling

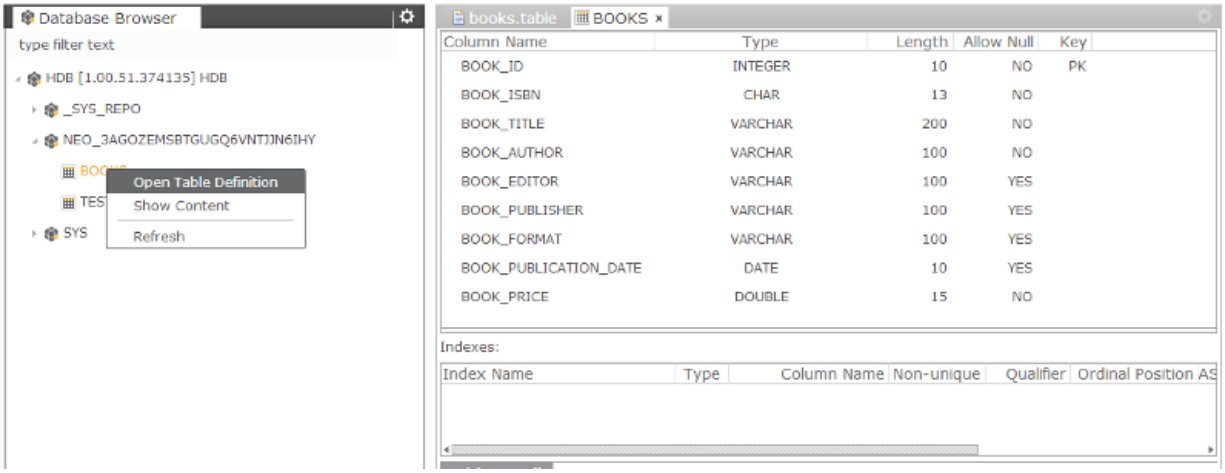
One of the biggest advantages of Dirigible project is its powerful design-time tooling. It claims to be Development Environment as a Service (DEaaS), means it is complete enough to cope with all the needs of a developer, building cloud based dynamic application only via a web browser. It is too ambitious statement and it realy is because it is based on the greatest yet mass-used so far IDE by the on-premise developers - Eclipse and its very innovative project RAP.

The different views and editors are separated by default to a few perspective depending on the purpose:

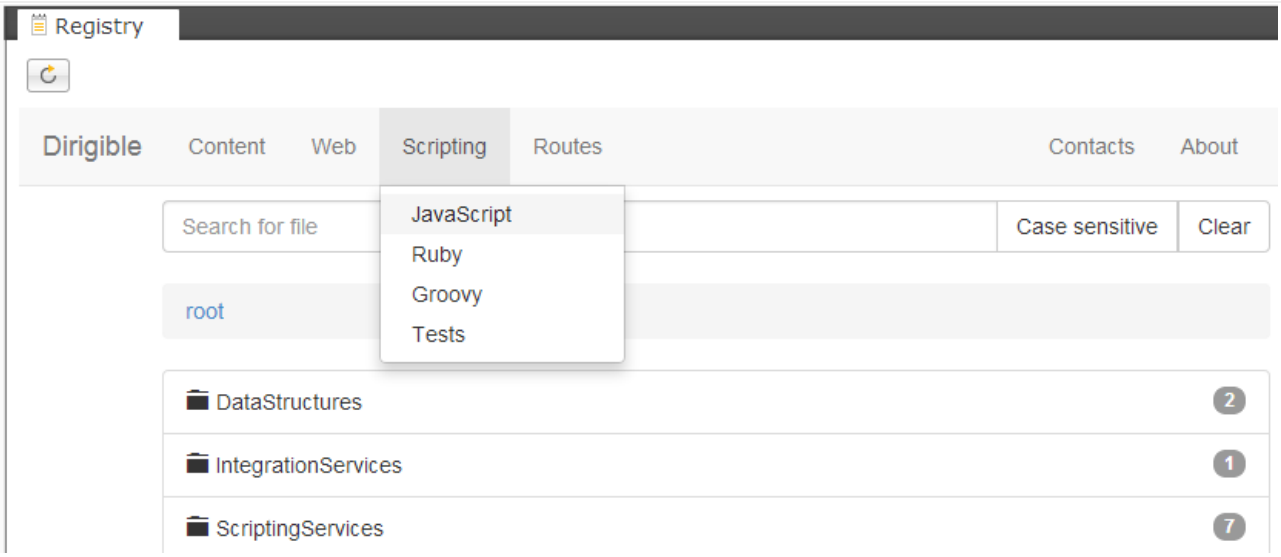
- Workspace Perspective - responsible to overall project management.



- Database Perspective - for inspecting low level raw content directly in the underlying database with the Database Browser and SQL Console



- Repository Perspective - looking into the current state as well as the history of the repository content
- Registry Perspective - an integrated view to the published resources and active service end-points in the runtime containers

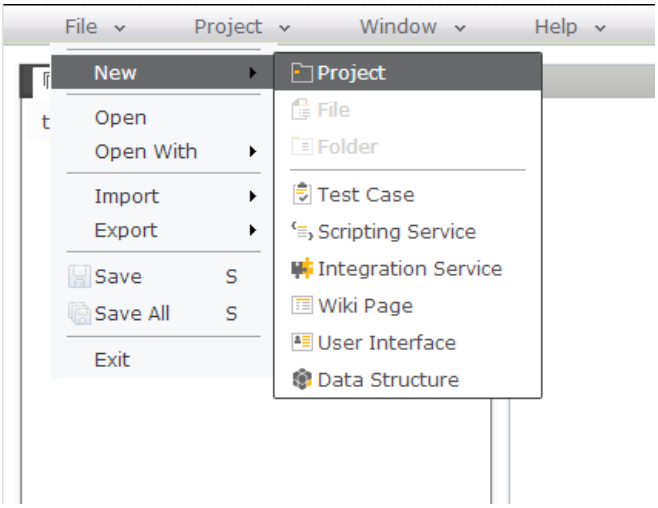

Registry
Dirigible
Content
Web
Scripting
Routes
Contacts
About
JavaScript
Ruby
Groovy
Tests
Search for file
Case sensitive
Clear
root
DataStructures
2
IntegrationServices
1
ScriptingServices
7

Workspace Perspective

This is the place where you actually develop your dynamic applications. This perspective contains all the views and editors that help you in overall implementation from domain models via services to the user interface.

Main views which are opened by default in this perspective are:

- *Workspace Explorer* - it is a standard view on the projects in your workspace. It shows the folder structure as well as the contained resources. There is a pop-up menu assigned to the project node:



via which you can create new artifacts:

- Project
- Data Structure
- Scripting Service
- User Interface
- Wiki Page
- Integration Service
- Test Case

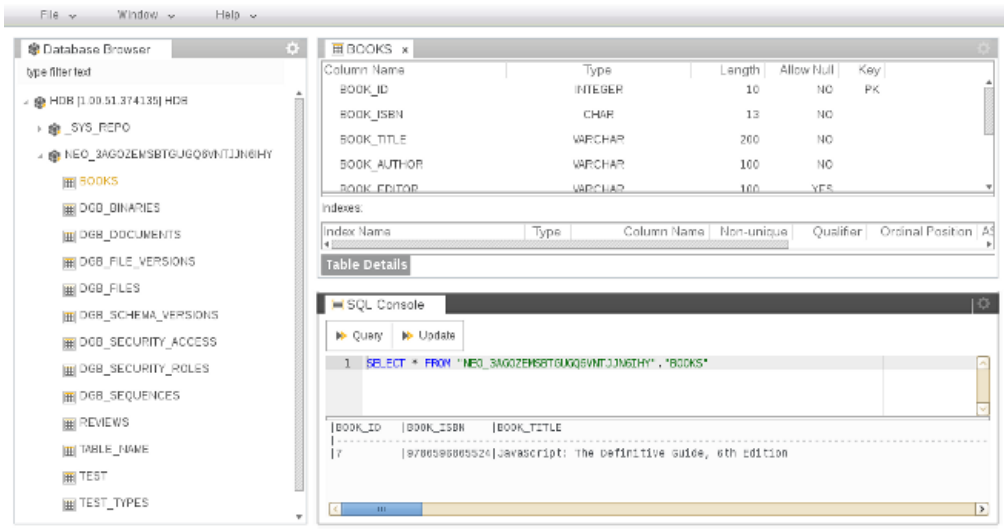
or just regular:

- File and
- Folder

While selecting an artifact you can use Open or Open With actions to load its content in the corresponding editor e.g. Source Editor.

Database Perspective

Database Perspective contains the tools for inspection and manipulation of the artifacts within the underlying relational database. You have direct access to the default target schema assigned to your account in the cloud platform. From the Database Browser you can expand the schema item and to see the list of all the tables and views created either via the models or directly via SQL script.



The actions which can be performed on a table or view are:

- Open Definition - presents the columns layout
- Show Content - generates a select statement in the SQL Console and executes it
- Export Data - exports the data from the table or view in *.dsv format
- Delete - removes physically the table from the database

Another tool in the Database Perspective is very generic and in the same time very powerful one - SQL Console.

Debug

The toolkit offers Debugging functionality. The goal is to easy the developers in the hunt of server side bugs.

Debug Perspective

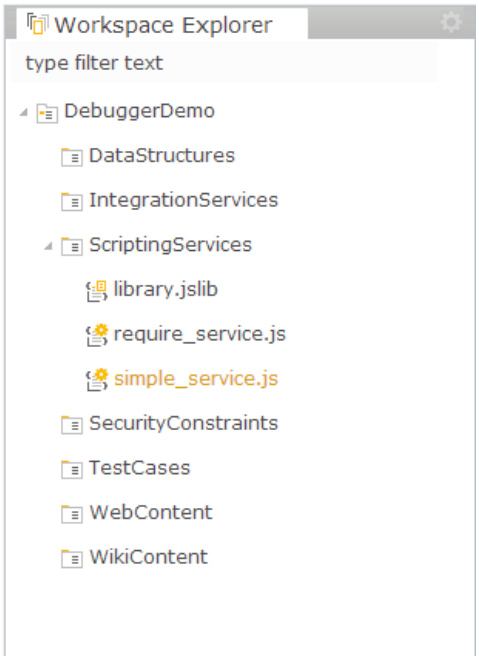
* *Sessions* - contains all debug execution sessions. * *Variables/Values* - contains variables and their values, available in the current scope of execution. * *File/Row/Source* - contains inforamation about in witch file and witch row, a *Breakpoint* is set.

Available Commands

- Refresh
- Step Into
- Step Over
- Continue
- Skip all breakpoints

Debugging Example

Step 0 - Project Structure



simple_service.js

```
main();


function main(){
    var message = createMessage();
    var students = createStudents();
    response.getWriter().println(message);
    response.getWriter().println(JSON.stringify(students));
}


function createMessage(){
    var initialValue = 1;
    var endValue = startCounter(initialValue);
    var message = 'Initial value was '+initialValue+', end value is '+endValue;
    return message;
}


function startCounter(value) {
    for(var i = 0; i < 5; i++){
        value ++;
    }
    return value;
}


function createStudents(){
    var students = [];
    students.push(createStudent('Desi', 18));
    students.push(createStudent('Jordan', 21));
    students.push(createStudent('Martin', 22));
    return students;
}


function createStudent(name, age){
    var student = {};
    student.name = name;
    student.age = age;
    return student;
}
```

library_jslib

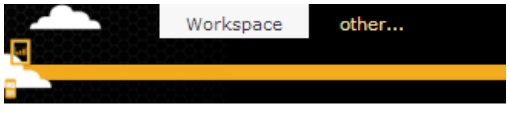
```
exports.generateGuid = function() {
    var guid = ''+uuid.randomUUID();
    return guid;
};
```

require_service.js

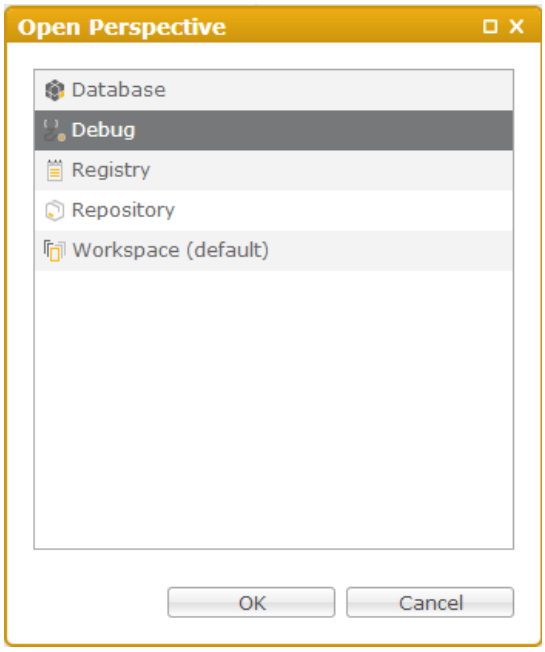
```
var guidGen = require('/DebuggerDemo/library');
var user = 'Test User';
var id = guidGen.generateGuid();


response.getWriter().println(user+", id "+id);
response.getWriter().flush();
response.getWriter().close();
```

Step 1 - Open Debug Perspective

Click *other...* to list available perspectives



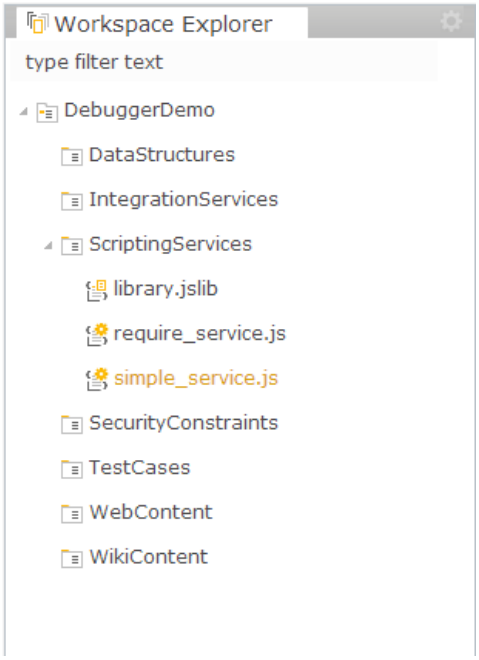
From the list select *Debug* perspective



Now *Debug* perspective is opened

Step 2 - Start Debugging

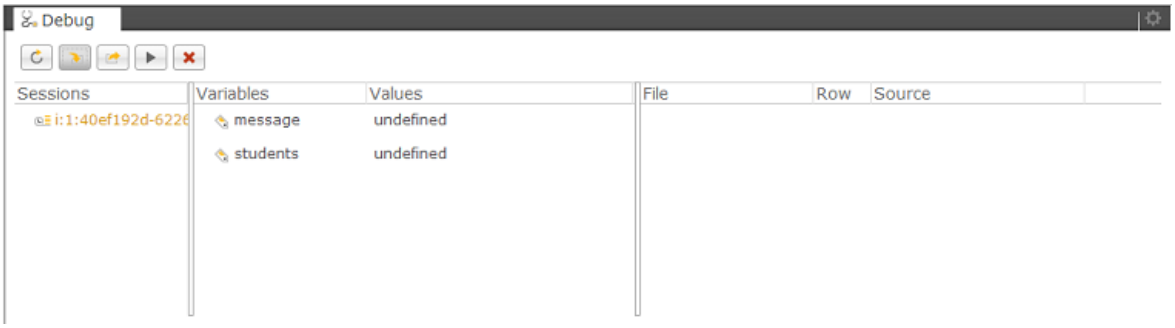
Select *simple_service.js* from *Workspace Explorer*



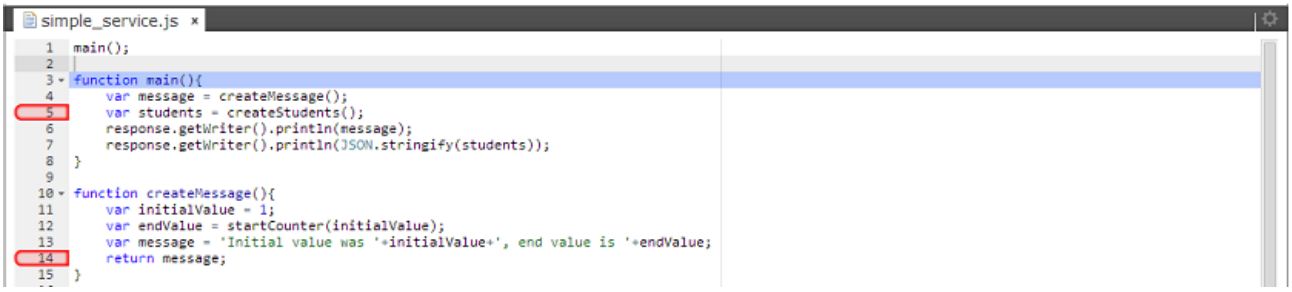
Debugger was started and waits for user interaction

In the *Debug* view press *Refresh* button to list available *Debug Sessions* and select one.

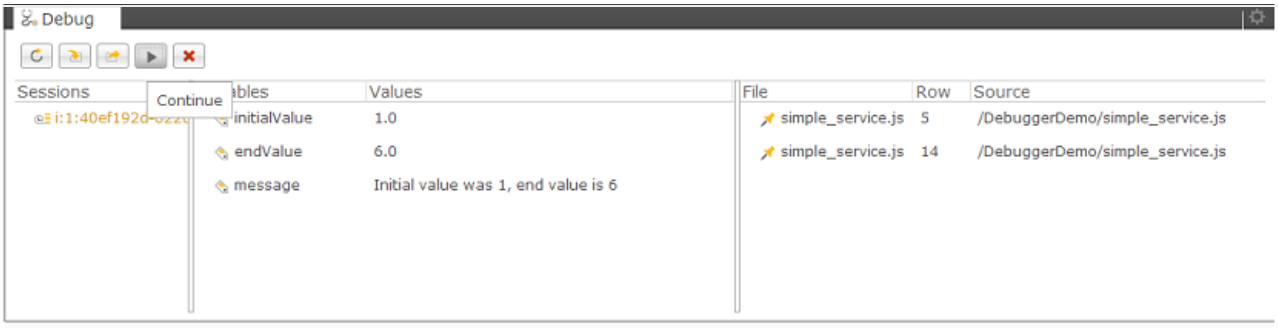
Press *Step Into* button to continue script's execution.

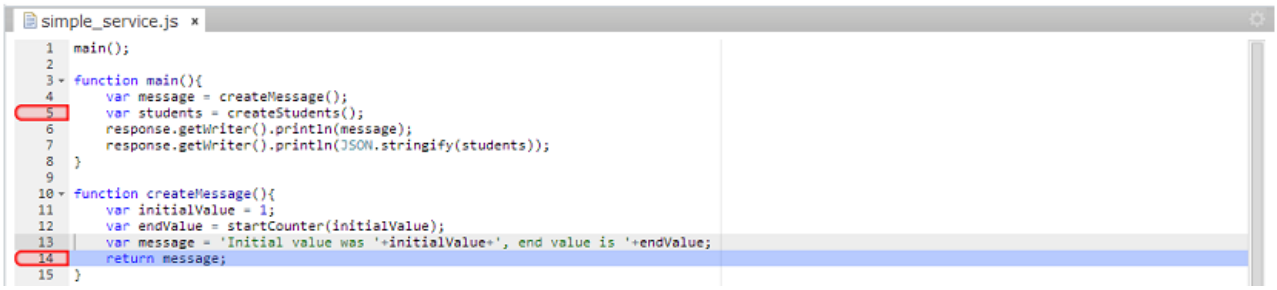


Let's set some *Breakpoints*. Click on the line numbers on the left of the opened editor. Press *Refresh* button to see *Breakpoints* that were set.

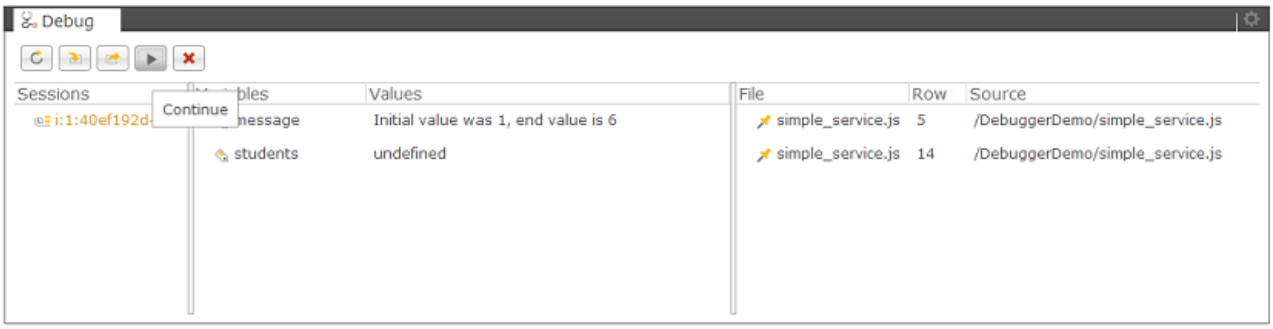


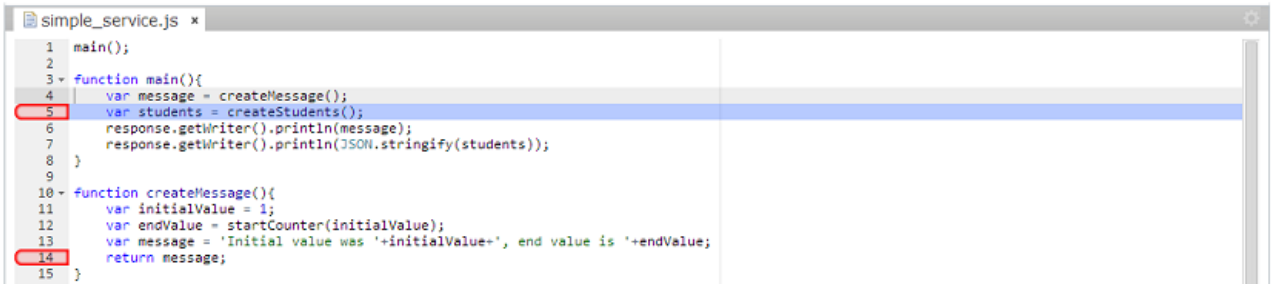
Press *Continue* button to resume script's execution to the next *Breakpoint*.





Press *Continue* ▶ button again.

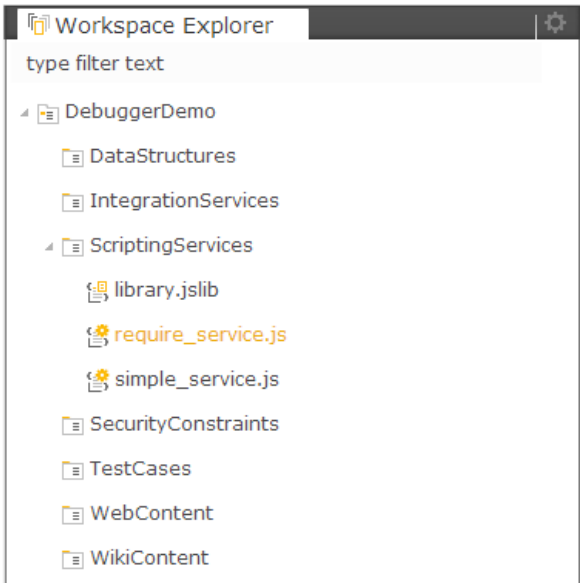


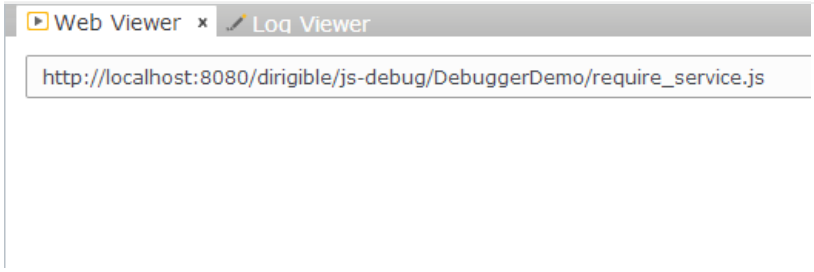


To exit *Debug Session* press *Skip all breakpoints* ✖ button or continue pressing *Step Over* or *Step Into* buttons until script's execution finish.

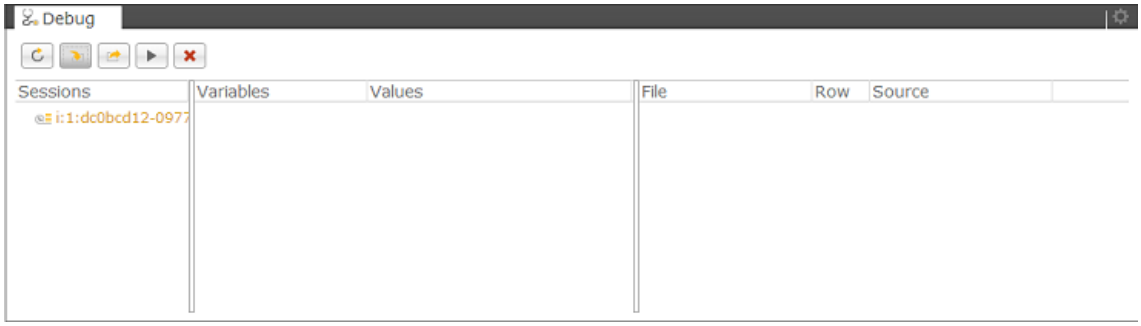
Step 3 - Debugging Scripts Requiring Libraries

Select *require_service.js* from *Workspace Explorer*

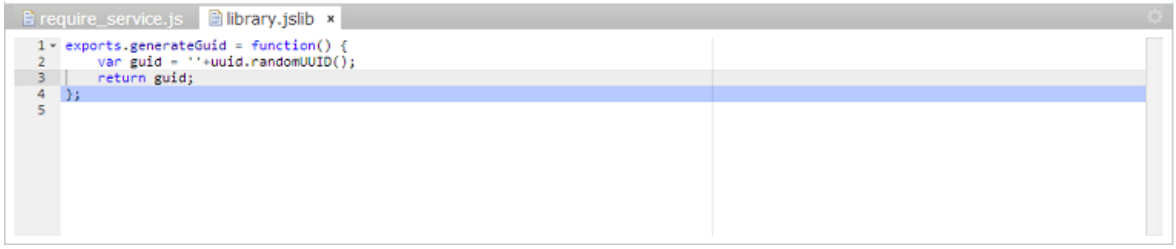




A new *Debug Session* was started. Press *Refresh* button, select session and press *Step Into* button.

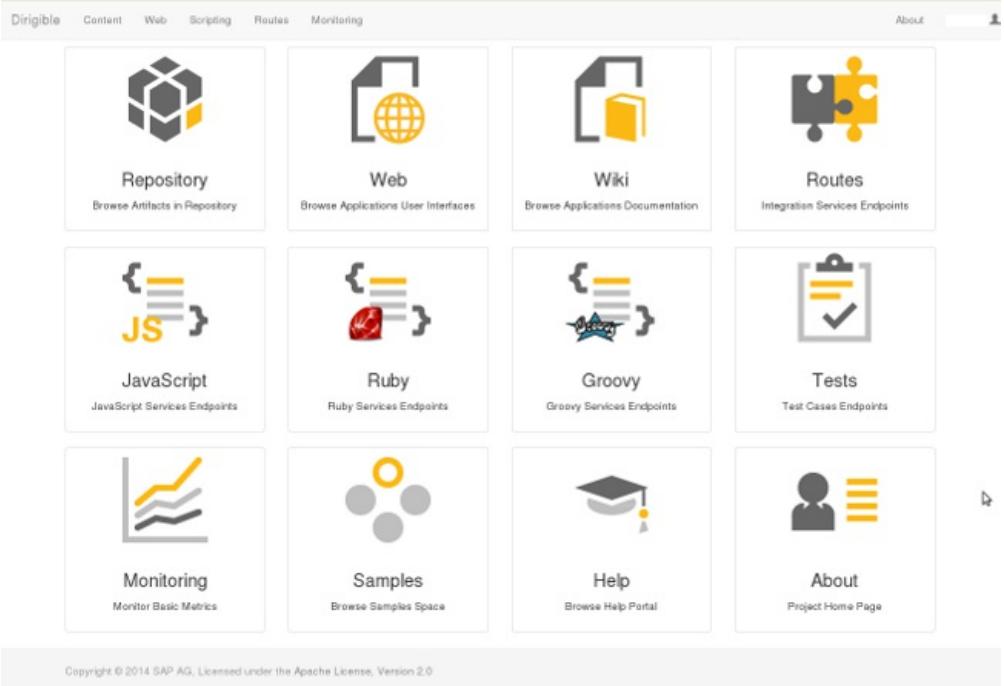


Continue debugging.



Registry

The entry point of the searching and browsing of the service end-points as well as monitoring and administration at runtime phase is the Registry. Technically it is a space within the Repository where all the published artifacts are placed.



To access the user interface you can point to the runtime context - default one is "dirigible"

> *http //[host]:[port]/dirigible*

End-points

From the index page of the Registry, you can navigate to the corresponding sub-pages for browsing the raw content of the Repository, published user interfaces (html, css, client-side javascript, etc.), [documentation[(wiki_content.html) of the applications, lookup the end-points of the scripting services as well as the end-points of the integration services.

Monitoring Tools

The last phase of the applications life-cycle includes administration and monitoring.

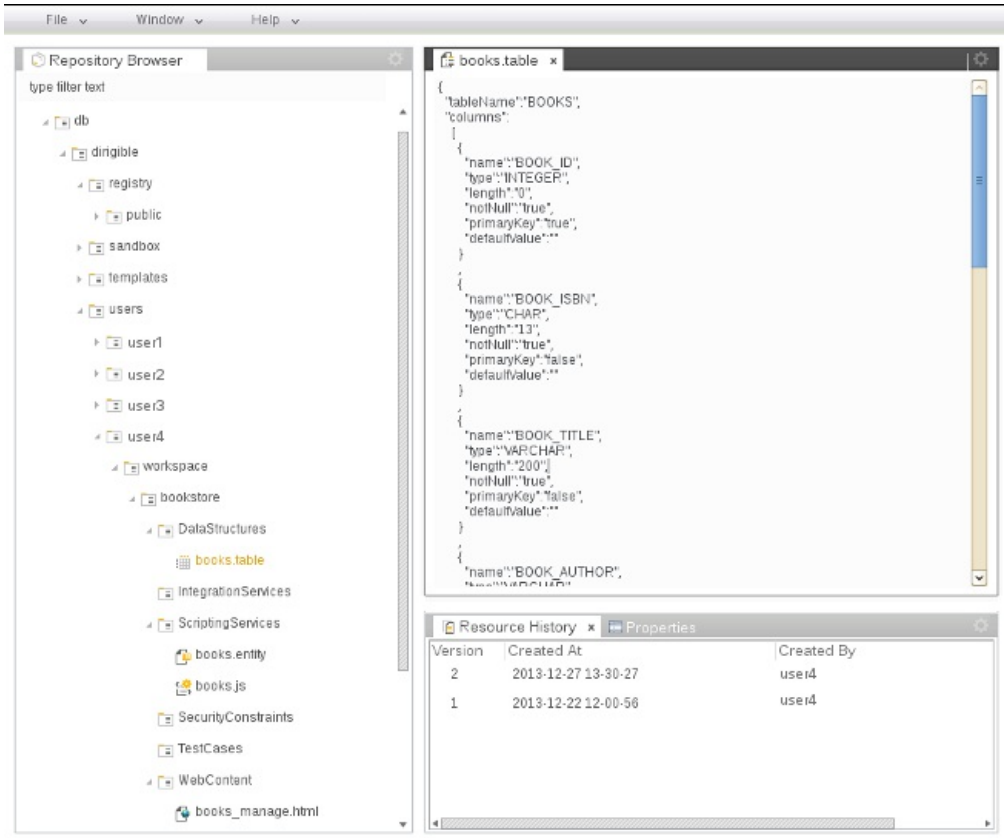
Via the Registry interface you can navigate to the monitoring tools including:

- Hit Count Statistics
- Response Time Statistics
- Memory Allocations
- Log Traces

The URIs on which you want to collect information about the request parameters have to be registered in the Manage Access Locations section.

Repository Perspective

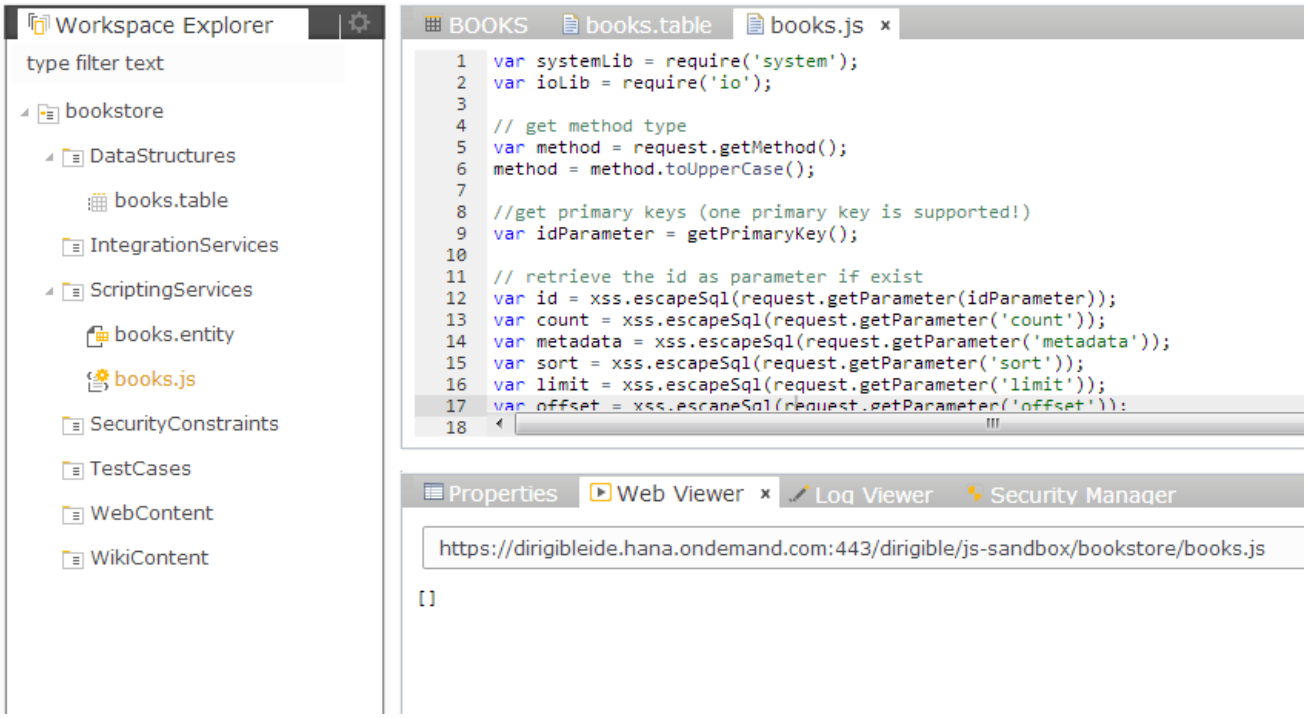
Repository Perspective gives the access to the raw structure of the underlying Repository content. There one could inspect at low level the project and folder structure as well as the artifacts content in the read-only plain editor.



Resource History View lists the most recent versions of the main artifact. It plays a role as local file history. As soon as the Repository component doesn't target the version control system by itself, you should rely on the Git integration for secure resource management. The history of the Repository is automatically cleared up after a given period of time.

Source Editor

Technically Source Editor provided by the toolkit is embedded version of well known ACE Editor in RAP environment. It supports syntax highlighting of mostly used development languages and data formats such as JavaScript, Ruby, Groovy, JSON, XML, HTML and many more.

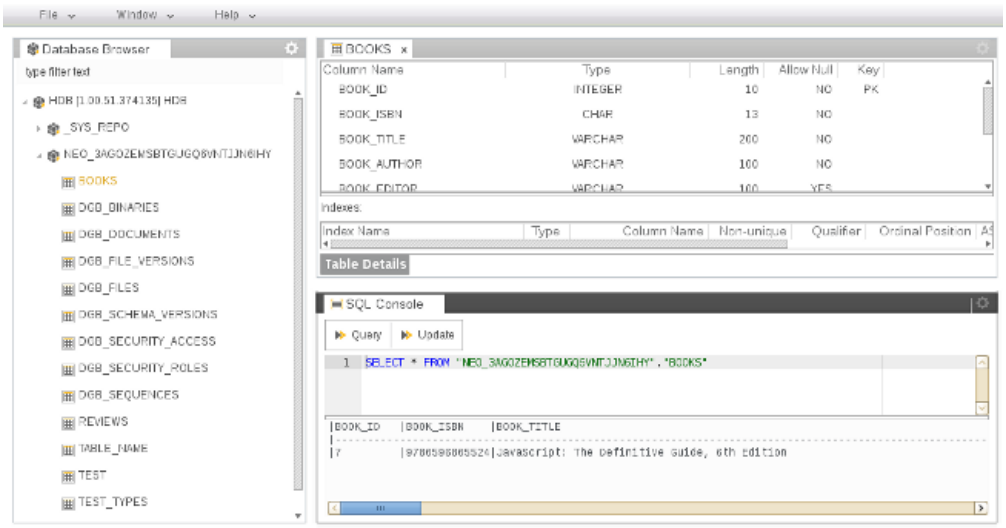


Cut, Copy, Paste as well as Undo/Redo actions are built-in. Key bindings aligned to the standard text editors also contribute to the increasing of development productivity.

SQL Console

Generic and in the same time most powerful and most dangerous tool for database management - SQL Console.

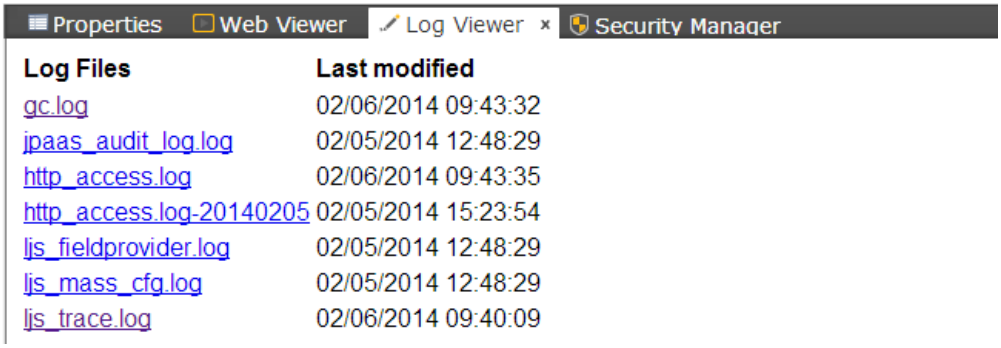
There are two separated areas - at the top you can enter the SQL script compliant to the underlying database system. At the bottom - you get the result of the execution in fixed text format.



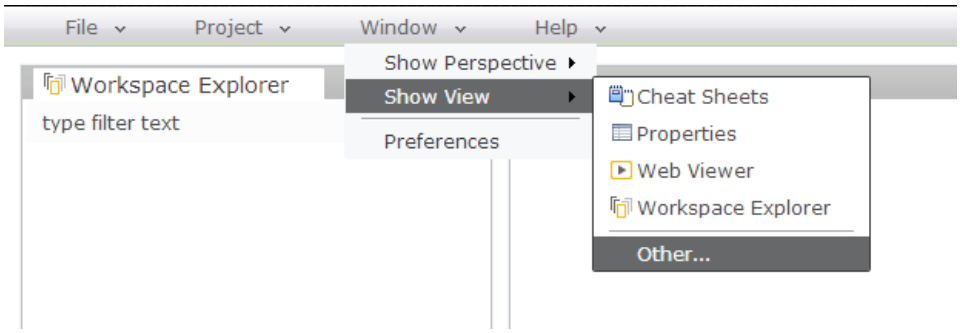
There is also separation between the *Query* and *Update* statements, which the user can execute depending on his/her roles.

Log Viewer

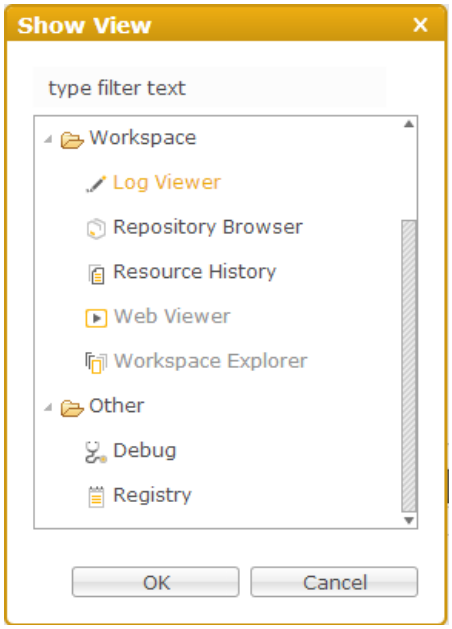
List all available log files



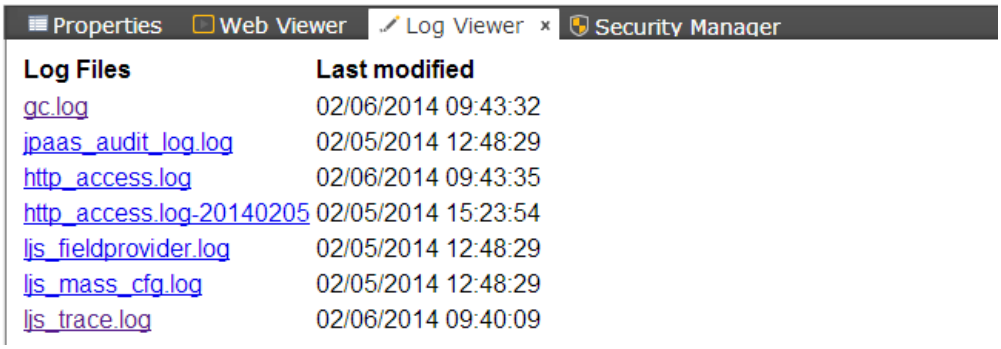
If the *Log Viewer* is not available, navigate to *Window->Show View->Other...*



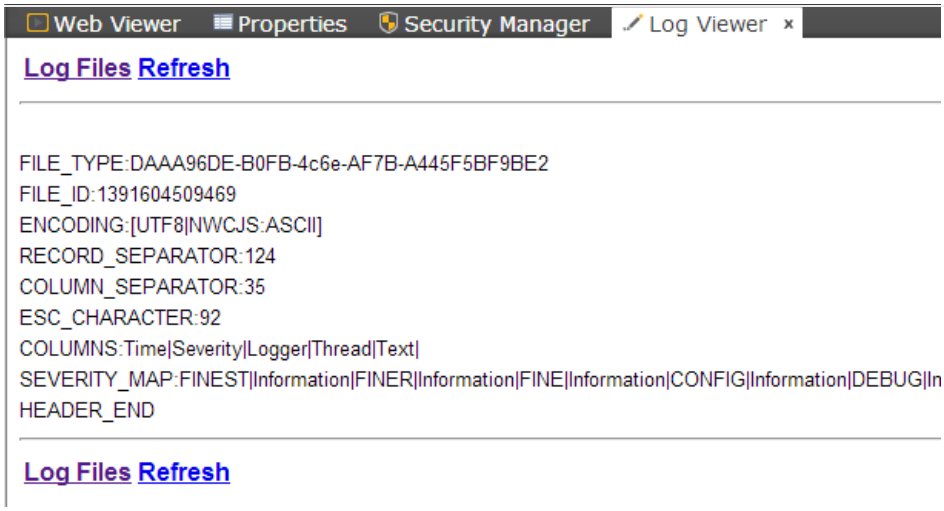
Then select from *Workspace* folder *Log Viewer*



Now the *Log Viewer* is available



Open any log file, for example *ljsmasscfg.log*



Log Files - go back to the list of available log files *Refresh* - refresh the content of the opened log file (needs to be pressed if new logging event occurred)

More about the server side logs and configurations can be found at: [logging service[(service_logging.html) section.

Runtime Services

There are several REST services available at runtime, which can give you another communication channel with Dirigible containers.

Search Service

Search Service exposes the Apache Lucene memory index over the Repository content.

- Main use case - search by query string

The endpoint is: */search*

> Parameter: *q*
>
> http //[host]:[port]/dirigible/ *search?q=[search_query]*
>
> Explicit search for *name* or *path* fields e.g.:
>
> http //[host]:[port]/dirigible/ *search?q=path:*

More info about the query syntax of Apache Lucene can be found here.

- Hard reindex the memory store

> Parameter: *reindex*
>
> http //[host]:[port]/dirigible/ *search?reindex*

Access Log Service

Via the Access Logs Service one can manage the locations to be filtered and registered as well as to receive the comprehensive information about the accessed ones for the latest time period.

> The endpoint is: */acclog*

For Management of Locations:

```
GET: http //[host]:[port]/dirigible/ *acclog*

POST: http //[host]:[port]/dirigible/ *acclog* //

DELETE: http //[host]:[port]/dirigible/ *acclog* //

DELETE: http //[host]:[port]/dirigible/ *acclog* /all

GET: http //[host]:[port]/dirigible/ *acclog* /locations
```

For chart compliant data:

Parameter *hitsPerPattern* - hits count calculated grouped by the locations above GET: http //[host]:[port]/dirigible/ *acclog* ? *hitsPerPattern*

Parameter *hitsPerProject* - hits count calculated grouped by the project names GET: http //[host]:[port]/dirigible/ *acclog* ? *hitsPerProject*

Parameter *hitsPerURI* - hits count calculated grouped by the actual requested URI GET: http //[host]:[port]/dirigible/ *acclog* ? *hitsPerURI*

Parameter *hitsByURI* - hits count calculated grouped hierarchically GET: http //[host]:[port]/dirigible/ *acclog* ? *hitsByURI*

Logging Service

Logging Service exposes the list of log files as well as theirs content.

> The endpoint is: */logging*

- To list all the available log files use:

> http //:/dirigible/ */logging*

- To retrieve the content of a log file (e.g. ljs_trace.log) use:

> Parameter: */log*
>
> http //:/dirigible/ */logging?log=ljs_trace.log*

The user interface in the IDE for accessing the logs is Log Viewer.

Operational Service

Operational Service exposes some utility functions.

> The end-point is: */op*

- To get the current logged-in user name:

> Parameter: *user*
>
> http //[host]:[port]/dirigible/ *op?user*

- To log-out from the the current user session:

> Parameter: */logout*
>
> http //[host]:[port]/dirigible/ *op?logout*

Repository Service

Repository Service gives full access to the Dirigible Repository API.

> The endpoint is: */repository*

To be able to use the service:

> User must be assigned to Role: *Repository Basic Authentication* headers must be provided Header *Accept* must be provided with value *application/json*

- To get the catalog of the full content:

> http //[host]:[port]/dirigible/ *repository*

- To get the index of a given collection:

> http //[host]:[port]/dirigible/ *repository/db/dirigible*

```
{
    "name" : "root",
    "path" : "/",
    "files" : [ {
        "name" : "registry",
        "path" : "/dirigible/repository/db/dirigible/registry/",
        "folder" : true
    }, {
        "name" : "sandbox",
        "path" : "/dirigible/repository/db/dirigible/sandbox/",
        "folder" : true
    }, {
        "name" : "templates",
        "path" : "/dirigible/repository/db/dirigible/templates/",
        "folder" : true
    }, {
        "name" : "users",
        "path" : "/dirigible/repository/db/dirigible/users/",
        "folder" : true
    } ]
}
```

- To get the content of a given artifact:

> http //[host]:[port]/dirigible/ *repository/db/dirigible/registry/WebContent/[mywebproject]/index.html*

Memory Service

Memory Service dumps the current information from the Runtime.

> The endpoint is: */memory*

The result is in JSON format, e.g.:

```
{
  "totalMemory":126353408,
  "availableProcessors":4,
  "maxMemory":1877475328,
  "freeMemory":109053320
}
```

To retrieve the chart compliant data:

> Parameter: *log* The endpoint is: */memory?log*

License

The Dirigible project source code base is provided under the Apache License 2.0

Apache License

Version 2.0, January 2004

http://www.apache.org/licenses/

TERMS AND CONDITIONS FOR USE, REPRODUCTION, AND DISTRIBUTION

1. Definitions.

"License" shall mean the terms and conditions for use, reproduction, and distribution as defined by Sections 1 through 9 of this document.

"Licensor" shall mean the copyright owner or entity authorized by the copyright owner that is granting the License.

"Legal Entity" shall mean the union of the acting entity and all other entities that control, are controlled by, or are under common control with that entity. For the purposes of this definition, "control" means (i) the power, direct or indirect, to cause the direction or management of such entity, whether by contract or otherwise, or (ii) ownership of fifty percent (50%) or more of the outstanding shares, or (iii) beneficial ownership of such entity.

"You" (or "Your") shall mean an individual or Legal Entity exercising permissions granted by this License.

"Source" form shall mean the preferred form for making modifications, including but not limited to software source code, documentation source, and configuration files.

"Object" form shall mean any form resulting from mechanical transformation or translation of a Source form, including but not limited to compiled object code, generated documentation, and conversions to other media types.

"Work" shall mean the work of authorship, whether in Source or Object form, made available under the License, as indicated by a copyright notice that is included in or attached to the work (an example is provided in the Appendix below).

"Derivative Works" shall mean any work, whether in Source or Object form, that is based on (or derived from) the Work and for which the editorial revisions, annotations, elaborations, or other modifications represent, as a whole, an original work of authorship. For the purposes of this License, Derivative Works shall not include works that remain separable from, or merely link (or bind by name) to the interfaces of, the Work and Derivative Works thereof.

"Contribution" shall mean any work of authorship, including the original version of the Work and any modifications or additions to that Work or Derivative Works thereof, that is intentionally submitted to Licensor for inclusion in the Work by the copyright owner or by an individual or Legal Entity authorized to submit on behalf of the copyright owner. For the purposes of this definition, "submitted" means any form of electronic, verbal, or written communication sent to the Licensor or its representatives, including but not limited to communication on electronic mailing lists, source code control systems, and issue tracking systems that are managed by, or on behalf of, the Licensor for the purpose of discussing and improving the Work, but excluding communication that is conspicuously marked or otherwise designated in writing by the copyright owner as "Not a Contribution."

"Contributor" shall mean Licensor and any individual or Legal Entity on behalf of whom a Contribution has been received by Licensor and subsequently incorporated within the Work.

1. Grant of Copyright License. Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable copyright license to reproduce, prepare Derivative Works of, publicly display, publicly perform, sublicense, and distribute the Work and such Derivative Works in Source or Object form.
2. Grant of Patent License. Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable (except as stated in this section) patent license to make, have made, use, offer to sell, sell, import, and otherwise transfer the Work, where such license applies only to those patent claims licensable by such Contributor that are necessarily infringed by their Contribution(s) alone or by combination of their Contribution(s) with the Work to which such Contribution(s) was submitted. If You institute patent litigation against any entity (including a cross-claim or counterclaim in a lawsuit) alleging that the Work or a Contribution incorporated within the Work constitutes direct or contributory patent infringement, then any patent licenses granted to You under this License for that Work shall terminate as of the date such litigation is filed.
3. Redistribution. You may reproduce and distribute copies of the Work or Derivative Works thereof in any medium, with or without modifications, and in Source or Object form, provided that You meet the following conditions:

You must give any other recipients of the Work or Derivative Works a copy of this License; and You must cause any modified files to carry prominent notices stating that You changed the files; and You must retain, in the Source form of any Derivative Works that You distribute, all copyright, patent, trademark, and attribution notices from the Source form of the Work, excluding those notices that do not pertain to any part of the Derivative Works; and If the Work includes a "NOTICE" text file as part of its distribution, then any Derivative Works that You distribute must include a readable copy of the attribution notices contained within such NOTICE file, excluding those notices that do not pertain to any part of the Derivative Works, in at least one of the following places: within a NOTICE text file distributed as part of the Derivative Works; within the Source form or documentation, if provided along with the Derivative Works; or, within a display generated by the Derivative Works, if and wherever such third-party notices normally appear. The contents of the NOTICE file are for informational purposes only and do not modify the License. You may add Your own attribution notices within Derivative Works that You distribute, alongside or as an addendum to the NOTICE text from the Work, provided that such additional attribution notices cannot be construed as modifying the License.

You may add Your own copyright statement to Your modifications and may provide additional or different license terms and conditions for use, reproduction, or distribution of Your modifications, or for any such Derivative Works as a whole, provided Your use, reproduction, and distribution of the Work otherwise complies with the conditions stated in this License. 5. Submission of Contributions. Unless You explicitly state otherwise, any Contribution intentionally submitted for inclusion in the Work by You to the Licensor shall be under the terms and conditions of this License, without any additional terms or conditions. Notwithstanding the above, nothing herein shall supersede or modify the terms of any separate license

agreement you may have executed with Licensor regarding such Contributions.

1. Trademarks. This License does not grant permission to use the trade names, trademarks, service marks, or product names of the Licensor, except as required for reasonable and customary use in describing the origin of the Work and reproducing the content of the NOTICE file.
2. Disclaimer of Warranty. Unless required by applicable law or agreed to in writing, Licensor provides the Work (and each Contributor provides its Contributions) on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied, including, without limitation, any warranties or conditions of TITLE, NON-INFRINGEMENT, MERCHANTABILITY, or FITNESS FOR A PARTICULAR PURPOSE. You are solely responsible for determining the appropriateness of using or redistributing the Work and assume any risks associated with Your exercise of permissions under this License.
3. Limitation of Liability. In no event and under no legal theory, whether in tort (including negligence), contract, or otherwise, unless required by applicable law (such as deliberate and grossly negligent acts) or agreed to in writing, shall any Contributor be liable to You for damages, including any direct, indirect, special, incidental, or consequential damages of any character arising as a result of this License or out of the use or inability to use the Work (including but not limited to damages for loss of goodwill, work stoppage, computer failure or malfunction, or any and all other commercial damages or losses), even if such Contributor has been advised of the possibility of such damages.
4. Accepting Warranty or Additional Liability. While redistributing the Work or Derivative Works thereof, You may choose to offer, and charge a fee for, acceptance of support, warranty, indemnity, or other liability obligations and/or rights consistent with this License. However, in accepting such obligations, You may act only on Your own behalf and on Your sole responsibility, not on behalf of any other Contributor, and only if You agree to indemnify, defend, and hold each Contributor harmless for any liability incurred by, or claims asserted against, such Contributor by reason of your accepting any such warranty or additional liability.

END OF TERMS AND CONDITIONS

APPENDIX: How to apply the Apache License to your work To apply the Apache License to your work, attach the following boilerplate notice, with the fields enclosed by brackets "[]" replaced with your own identifying information. (Don't include the brackets!) The text should be enclosed in the appropriate comment syntax for the file format. We also recommend that a file or class name and description of purpose be included on the same "printed page" as the copyright notice for easier identification within third-party archives.

```
Copyright [yyyy] [name of copyright owner]


Licensed under the Apache License, Version 2.0 (the "License");
you may not use this file except in compliance with the License.
You may obtain a copy of the License at

    http://www.apache.org/licenses/LICENSE-2.0

Unless required by applicable law or agreed to in writing, software
distributed under the License is distributed on an "AS IS" BASIS,
WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
See the License for the specific language governing permissions and
limitations under the License.
```

Credits and Special Thanks

We would like to say big *THANK YOU* to all the open source projects we use as platfrom components:

- Rhino
- Eclipse Equinox
- Eclipse OSGi
- Remote Application Platfrom
- Mylyn
- Camel
- CXF
- Ant
- Derby
- Commons
- Geronimo
- HttpClient
- Xerces
- Xalan
- WS
- Felix
- Log4j
- Batik
- Avalon
- Velocity
- Quartz
- Spring Framework
- StaX
- Woodstox
- Jettison
- Groovy
- CyberNeko HTML
- Gson
- EZMorph
- ACE Editor
- JCraft
- JLine
- ASM
- Antlr
- Hamcrest
- JUnit
- jRuby
- wsdl4j
- Slf4j
- jsoap
- ICU
- Mockito
- JAF
- AOP Alliance
- jQuery
- Bootstrap
- AngularJS

Dirigible Samples

Dirigible Samples collect various applications created to demonstrate the main usages and strengths of the toolkit. They are built on scenarios with different complexity level from simple samples targeting demoing a single feature to a complete end-to-end solutions

Simple Samples

- Data Model - detailed description of the supported data structure descriptors and the corresponding database artifacts
- Entity Service - create a domain model object and generate RESTful service on top
- Entity User Interface - generate pattern-based user interface on top of na entity service
- JavaScript Service & Library - implement your own custom algorithms using the most popular dynamic language - JavaScript. Along with the fairly good number of out-of-the-box utilities implied in the context to speed-up the developer productivity, it is also shown how you can create your own set of functions and pack them as a libraries for later reuse
- User Interface - to complete the stack of capabilities for user interaction use-cases, you can go through the sample showing how to create User Interface based on jQuery, Bootstrap and the other related well proved AJAX libraries
- Mail Service - send email with only 5 lines of code
- REST Calls - example of how to send REpresentational State Transfer calls to other services
- REST Call with Authorization Header - shows how to add Authorization Header for a REST call
- Simple Routing - more advanced topics follows concerning integration of your application with external services. The first one shows the most simple integration service you can ever have - restful service with simple state based implementation
- Shielding JavaScript Service - the second one shows how to model your own web service and expose it, by connecting it to a JavaScript service as implementation.
- Scheduling Job on JavaScript Service - next sample shows how to schedule a job which triggers the execution of a JavaScript Service.
- Using simple data storage - how to put, get, clear, delete binary content
- XML to JSON and vice-versa conversions
- Documentation - how to add the documentation of your project using some templating technics

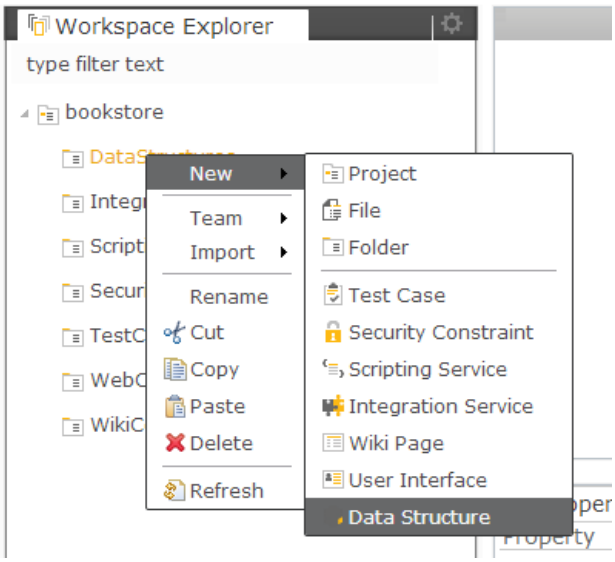
Solution Samples

- BookStore - comprehensive example that can be used as a step-by-step tutorial for creation of an online shop application. It combines most of the features above and show them in a holistic manner - from data models, thru the entity services, user interfaces, till the integration services and documentation.

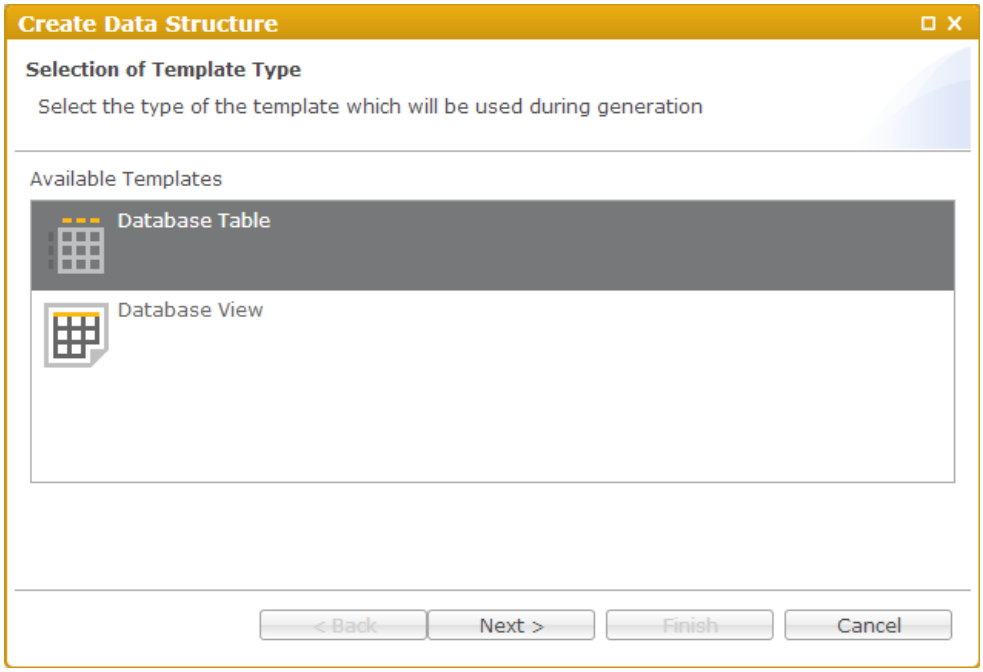
Data Model

Data models a.k.a domain models is the set of the entities of your application and also their relations. In Dirigible we use also the term data structures, which is more related to the actual artifact - the data descriptor. Let create the first model entity of the BookStore sample - the books table.

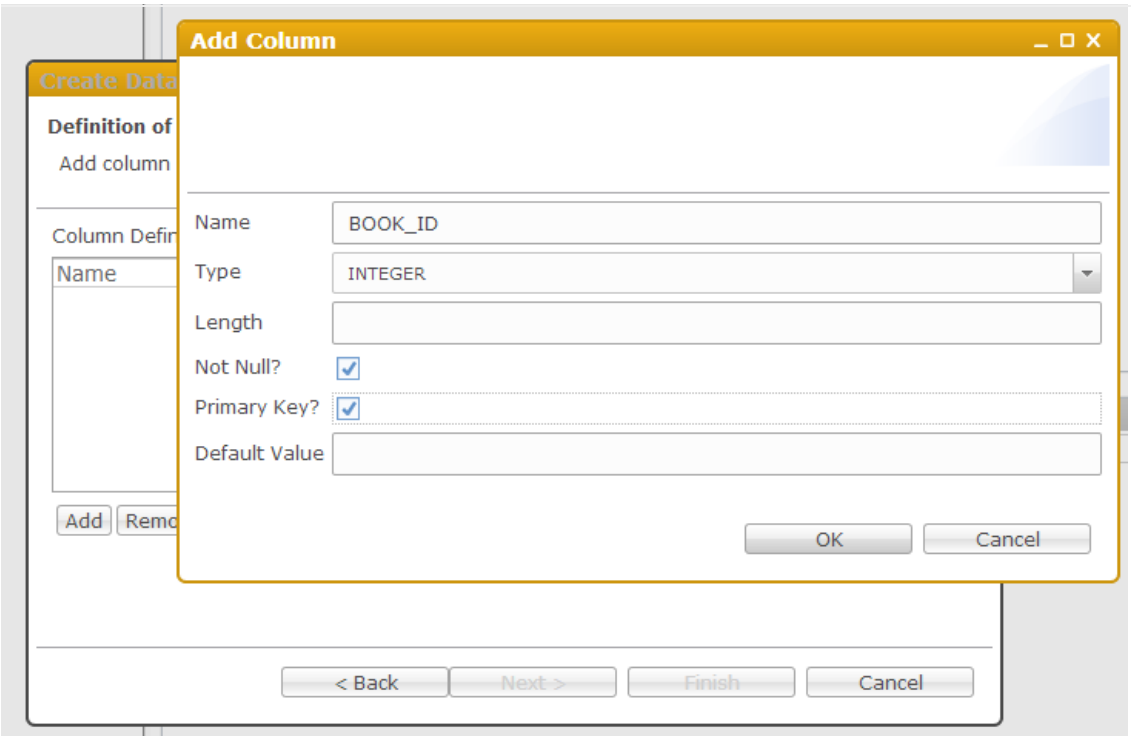
Select the DataStructures sub-folder of a project and open the pop-up menu (right-click). From the menu go to *New->Data Structure*



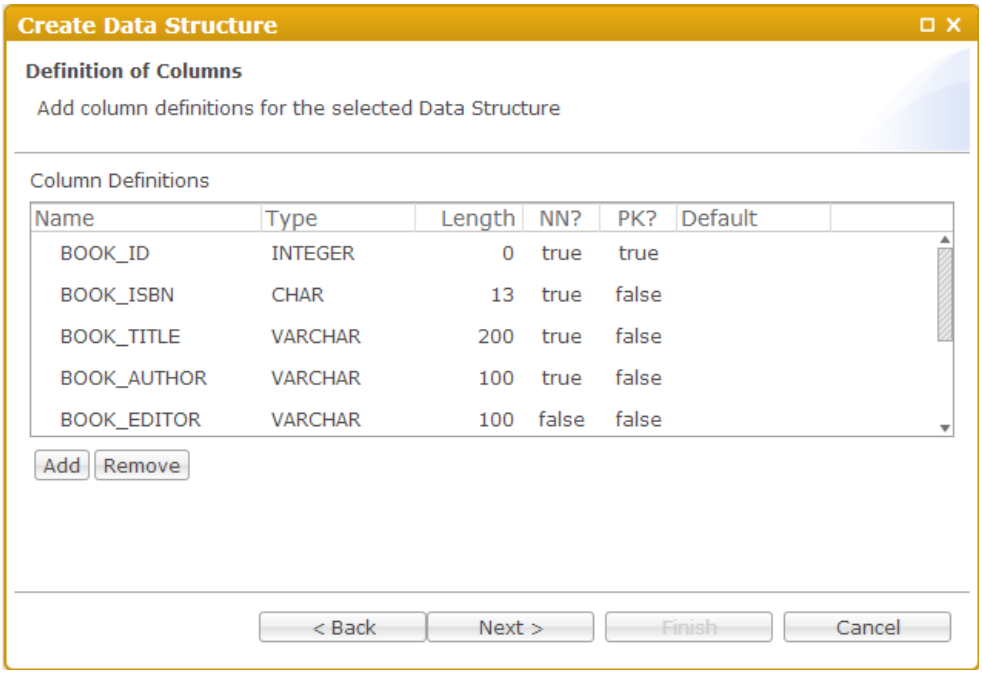
The first page of this wizard let you choose from the several artifacts related to the domain model. In this case we need a table where to store the books metadata like ISBN, Title, Author, etc. Choose "Database Table" and click Next



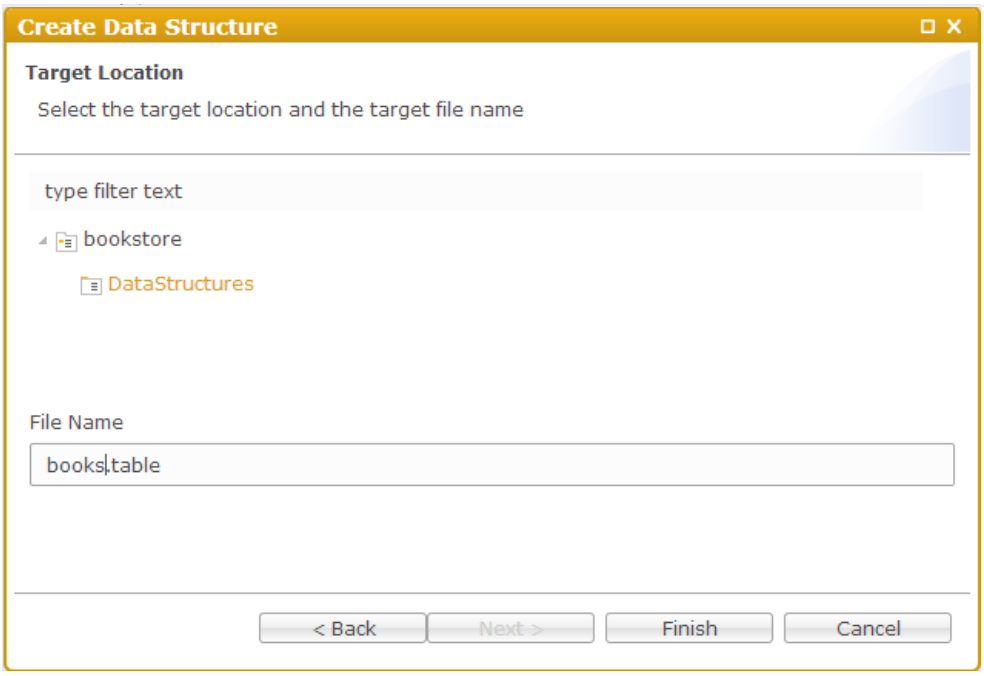
Use Add/Remove buttons to create the actual layout of the table




```
BOOK_ID                 INTEGER         not null    primaryKey
BOOK_ISBN               CHAR        13  not null
BOOK_TITLE              VARCHAR     200 not null
BOOK_AUTHOR             VARCHAR     100 not null
BOOK_EDITOR             VARCHAR     100
BOOK_PUBLISHER          VARCHAR     100
BOOK_FORMAT             VARCHAR     100
BOOK_PUBLICATION_DATE   DATE
BOOK_PRICE              DOUBLE          not null
```



Give a name and click finish



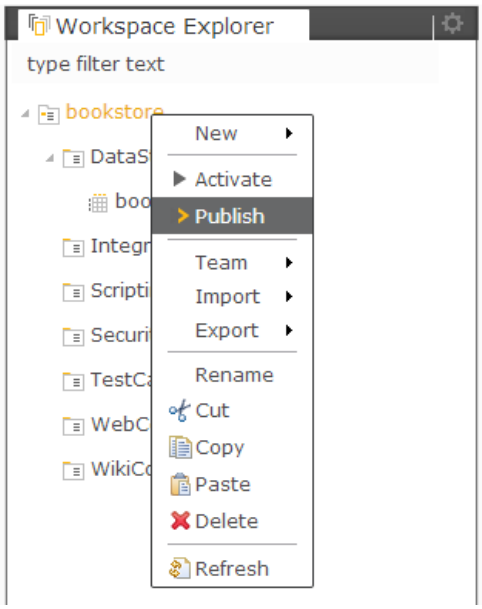
The table descriptor should be generated, based on your input and the file itself should be opened in the editors area

![New DataStructures Content]bookstore/11*books*new*ds*content.png!

```
{
  "tableName":"BOOKS",
  "columns":
    [
      {
        "name":"BOOK_ID",
        "type":"INTEGER",
        "length":"0",
        "notNull":"true",
        "primaryKey":"true",
        "defaultValue":""
      }
      ,
      {
        "name":"BOOK_ISBN",
        "type":"CHAR",
        "length":"13",
        "notNull":"true",
        "primaryKey":"false",
        "defaultValue":""
      }
      ,
      {
        "name":"BOOK_TITLE",
        "type":"VARCHAR",
        "length":"200",
        "notNull":"true",
        "primaryKey":"false",
        "defaultValue":""
      }
      ,
      {
        "name":"BOOK_AUTHOR",
        "type":"VARCHAR",
        "length":"100",
        "notNull":"true",
        "primaryKey":"false",
        "defaultValue":""
      }
      ,
      {
        "name":"BOOK_EDITOR",
        "type":"VARCHAR",
        "length":"100",
        "notNull":"false",
        "primaryKey":"false",
        "defaultValue":""
      }
      ,
      {
```

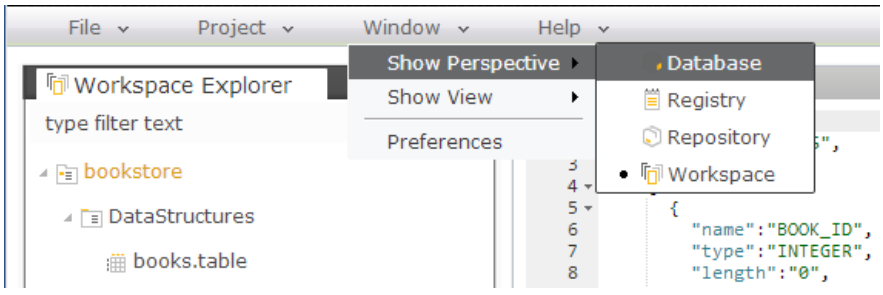
```
        "name":"BOOK_PUBLISHER",
        "type":"VARCHAR",
        "length":"100",
        "notNull":"false",
        "primaryKey":"false",
        "defaultValue":""
      }
      ,
      {
        "name":"BOOK_FORMAT",
        "type":"VARCHAR",
        "length":"100",
        "notNull":"false",
        "primaryKey":"false",
        "defaultValue":""
      }
      ,
      {
        "name":"BOOK_PUBLICATION_DATE",
        "type":"DATE",
        "length":"0",
        "notNull":"false",
        "primaryKey":"false",
        "defaultValue":""
      }
      ,
      {
        "name":"BOOK_PRICE",
        "type":"DOUBLE",
        "length":"0",
        "notNull":"true",
        "primaryKey":"false",
        "defaultValue":""
      }
    ]
}
```

More about the data structure types and their descriptors can be found here Now we have to create the real database artifact in the underlying database. This can be done via publish action from the project's popup menu.

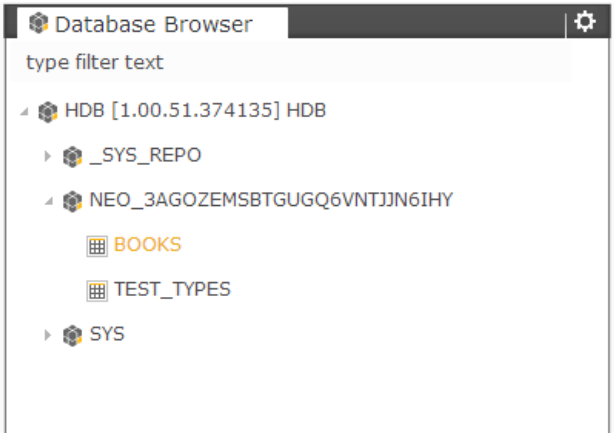


In this case (data structures artifacts) you can do the same also with activation action. Follow the links for more information about the differences between activation and publishing.

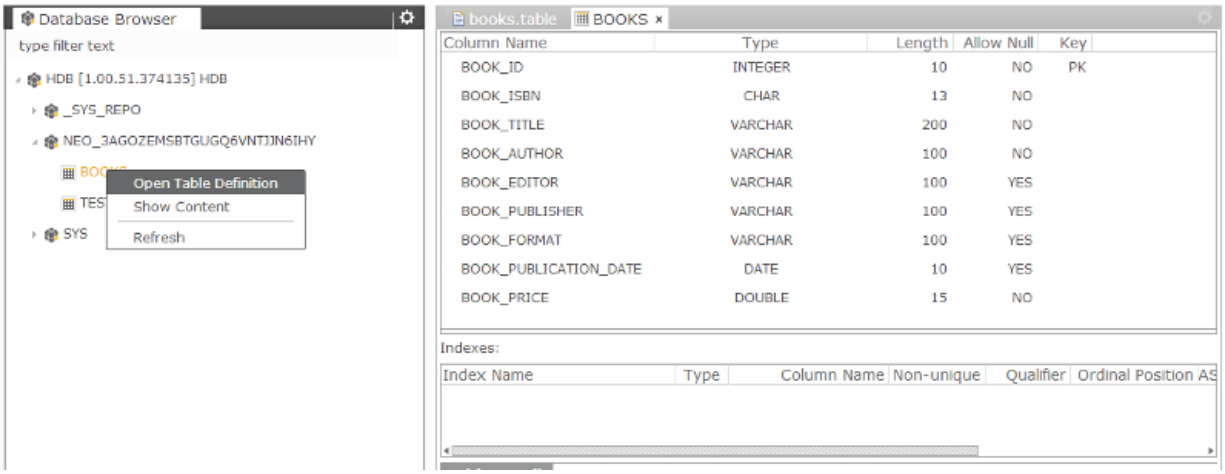
Once we have published successfully the project, we can go to Database perspective to double-check the table definition. From main menu go to Window->Show Perspective->Database



Open the database schema node and find the BOOKS table. Right-click and choose "Open Table Definition"



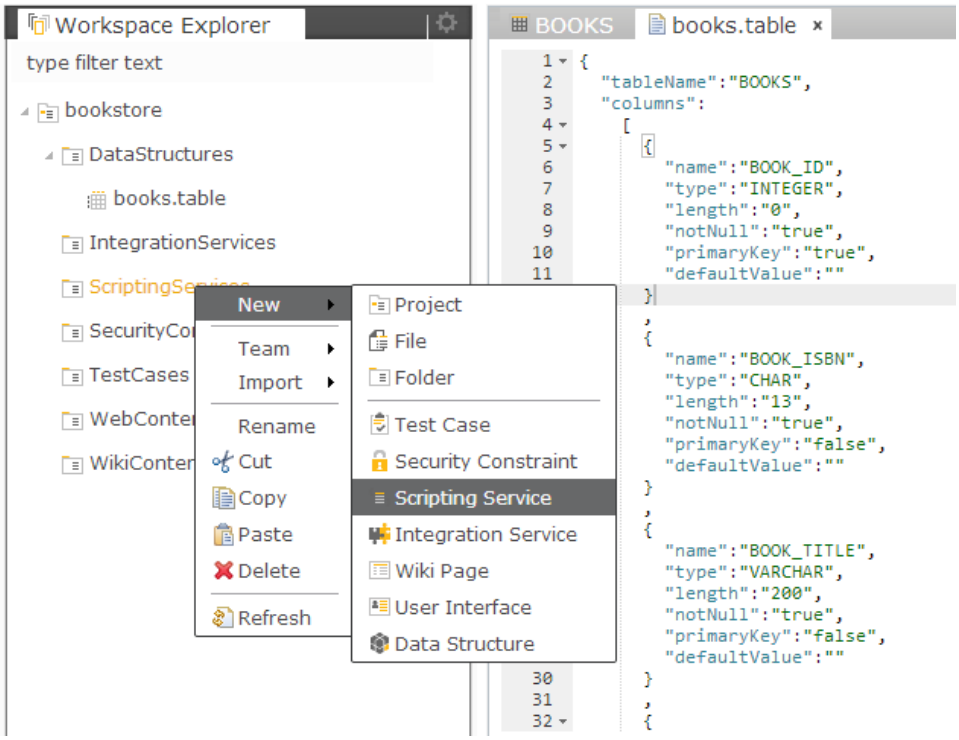
This will open the Table Definition Viewer



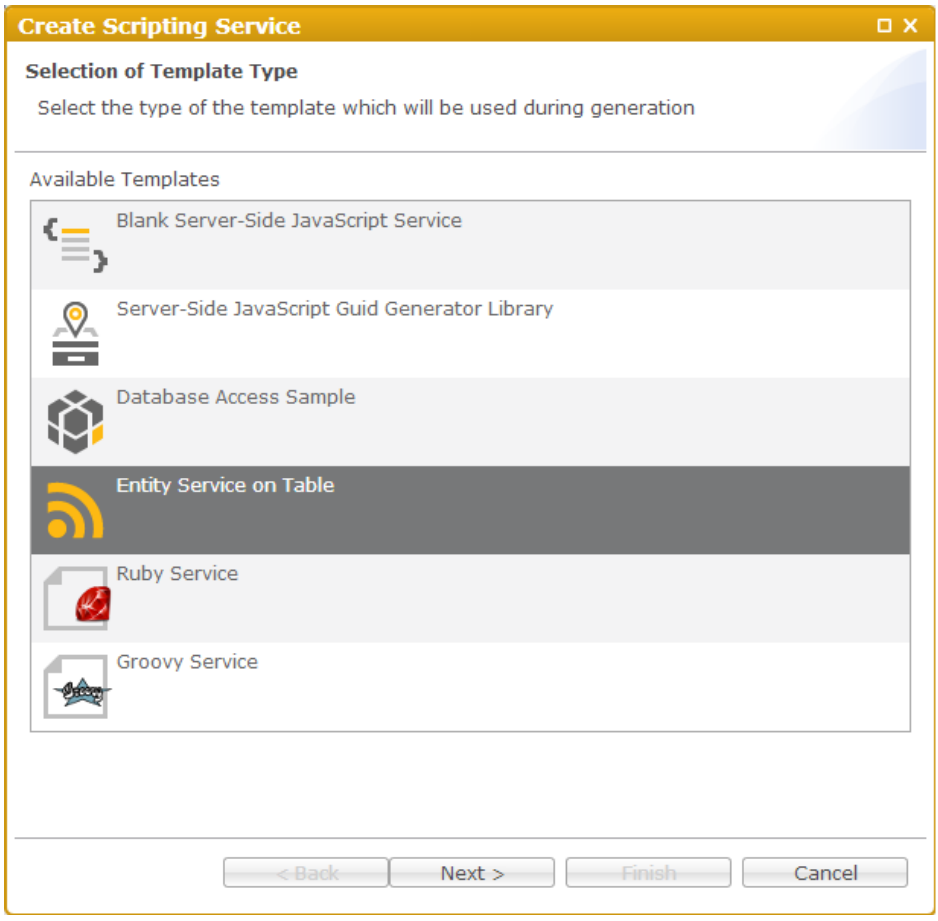
Entity Service

Entity Service meaning in terms of Dirigible is a RESTful service, which exposes the [CRUD](http://en.wikipedia.org/wiki/Create,*read*,update*and*delete] methods on top of the database table. The following steps shows how to generate such an entity service on top of existing table.

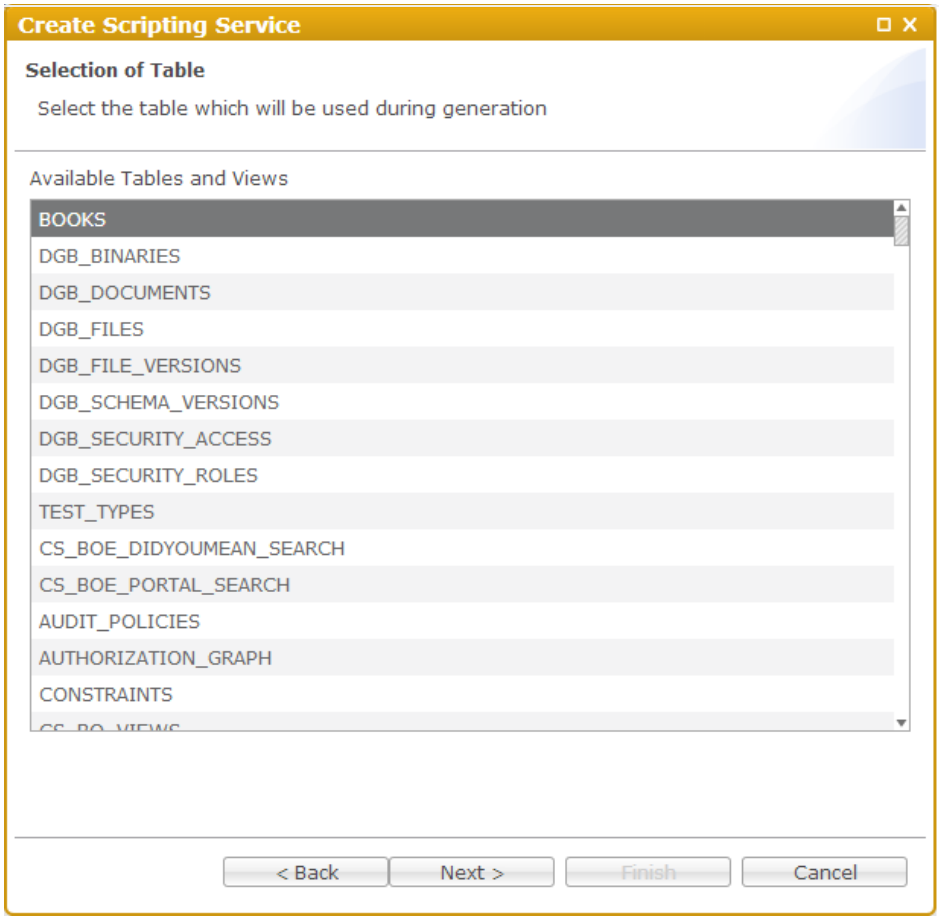
Select the ScriptingServices sub-folder of the project and open the pop-up menu From the menu go to *New->Scripting Service*



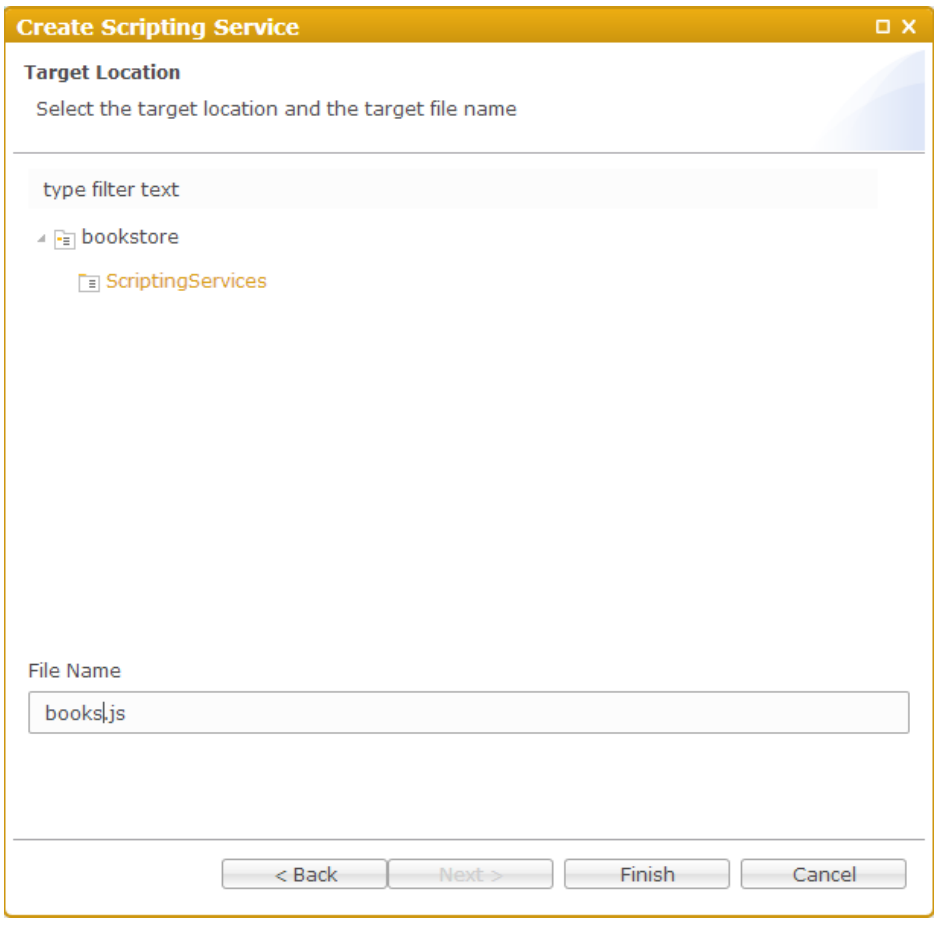
From the opened wizard select the "Entity Service" from the list of available templates.



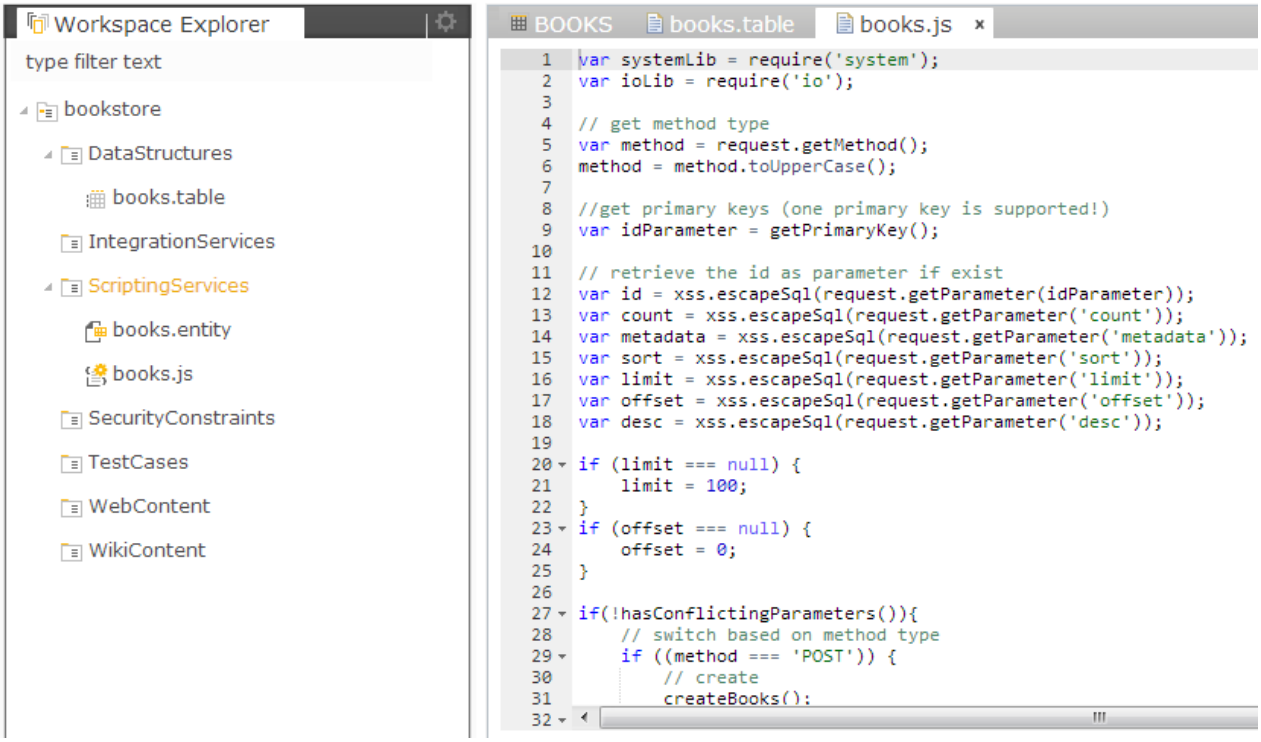
Once you have selected the "Entity Service" template and click "Next", the page which will be shown is specific one - lists all the available tables from the current database schema.



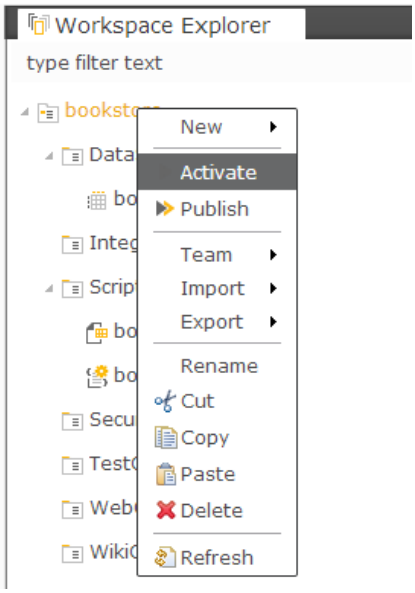
Give the name of your entity service on the next page and click finish



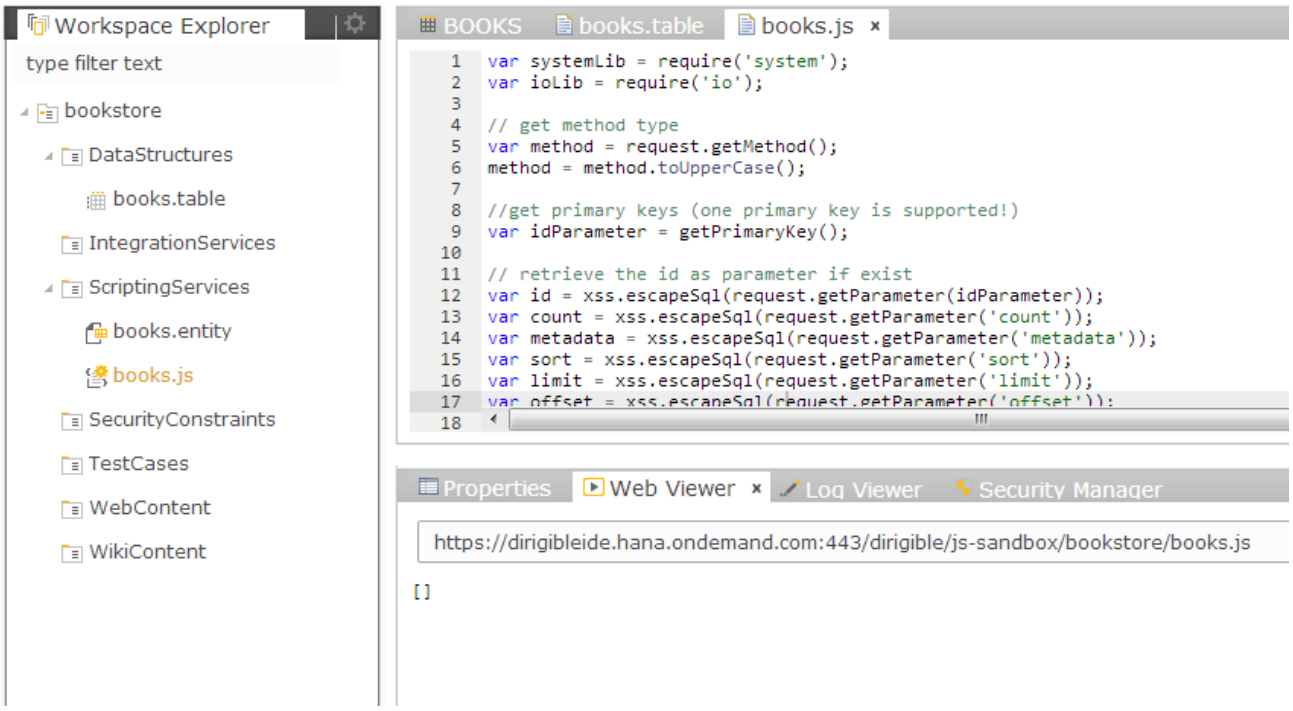
The generated service should be opened in the editors area.



Now we can use activation action from the project's pop-up menu to enable the service

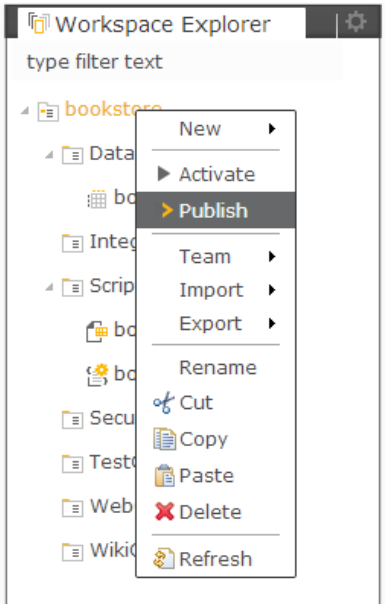


During the activation the artifact goes to the sandbox of the logged-in user. We can see the result of calling the service right away in the Preview (sould be opened by default in the Workspace Perspective), so find it (next to Properties view) and select it. Now go to the Workspace Explorer where the project is managed and select the service artifact (books.js). This will trigger the construction of the right URL of the service endpoint in the sandbox, hence you will see the result in the Preview.

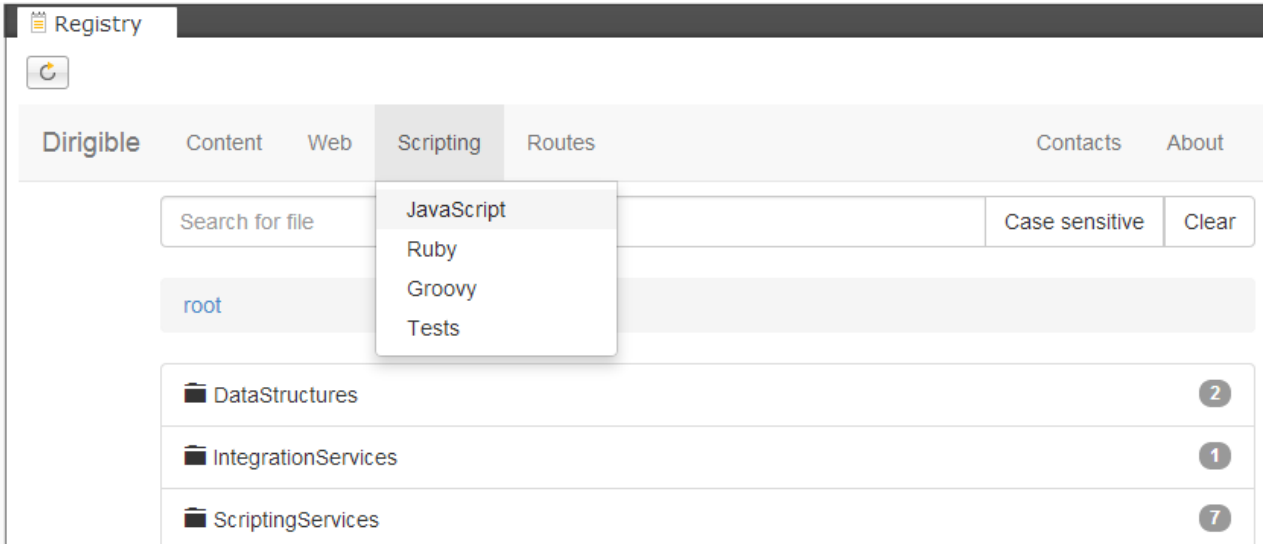


in this case just an empty JSON array.

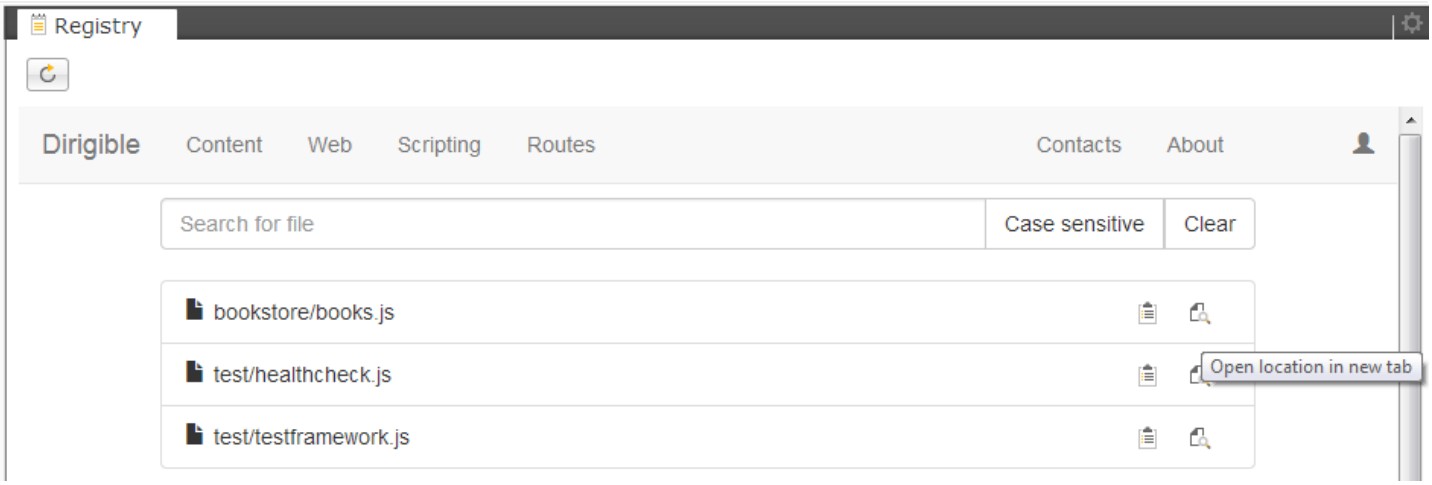
If you like the result in the sandbox you can publish the service, so that it become accessible by the other users.



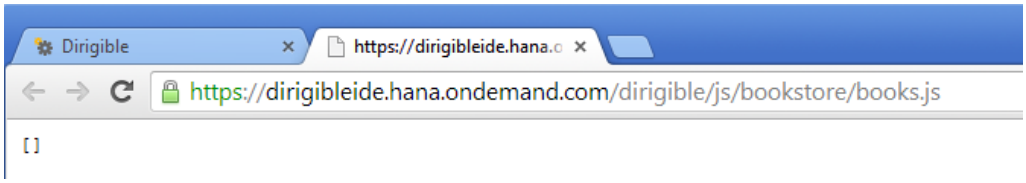
To find the URL you can go to the Registry Perspective. From the main menu go to Window->Show Pespective->Registry The Registry Perspective is representing a view to the enabled runtime content. From its menu choose Scripting->JavaScript to open the currently available server-side JavaScript service endpoints.



You can see the list of the available end-points, where you can find yours by naming convention .



The link to the service can be copied to the clipboard via the first image at the right side of the row or can be directly opened by clicking on the second image.



The naming convension for the service' endpoints URLs is as follows:

> *[protocol]://[host]:[port]/[dirigible's runtime application context]/[scripting container mapping]/[project]/[service path]*

e.g.

> *https ://dirigibleide.hana.ondemand.com/dirigible/js/bookstore/books.js*

The scripting containers mappings are:

- JavaScript
  - /js
  - /js-secured
- Ruby
  - /rb
  - /rb-secured
- Groovy
  - /groovy
  - /groovy-secured
- Test
  - /test

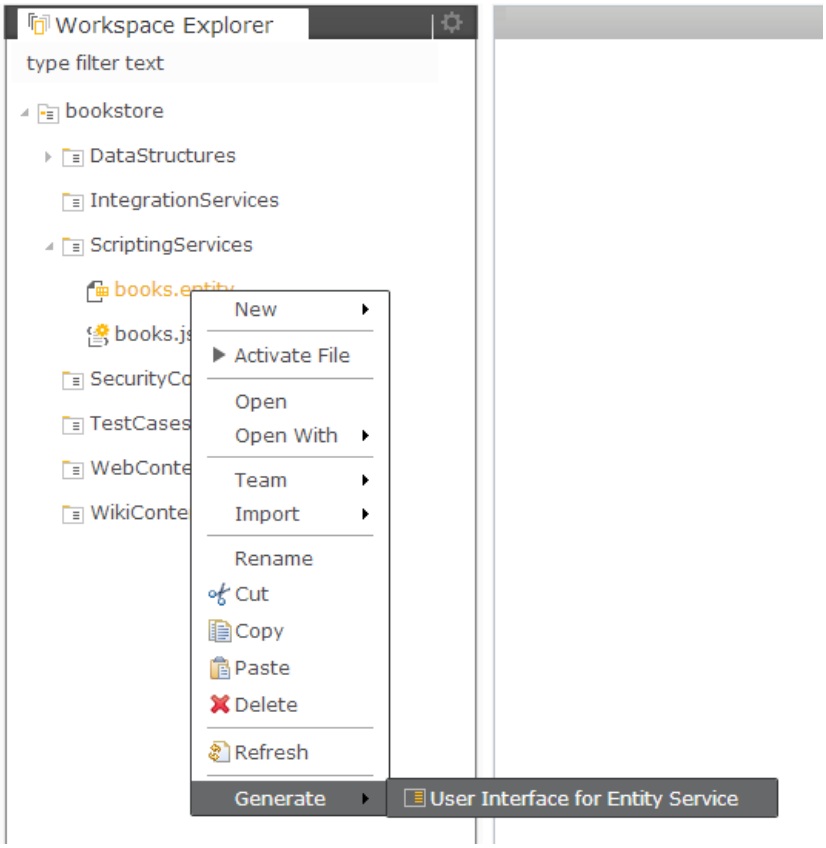
The suffix "-secured" above shows the access point for the secured end-points. More info can be found here.

There are some specific requirement for the table to be able to be exposed as entity service (e.g. primary key have to be defined, it should be a single column, etc.). Also the entity service itself support a bit more operations than the standard ones defined by HTTP. More about the entity services can be found here.

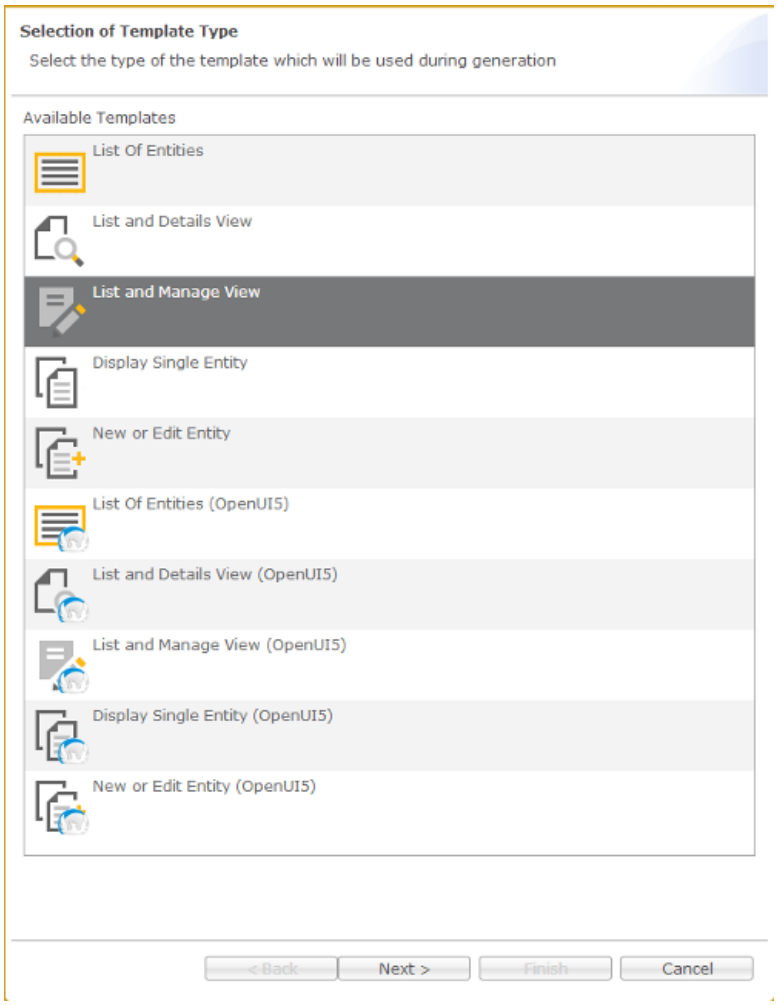
Entity User Interface

After the creation of the data model and the entity service, now we will going to generate an user interface for entity management (list, new, edit, delete...)

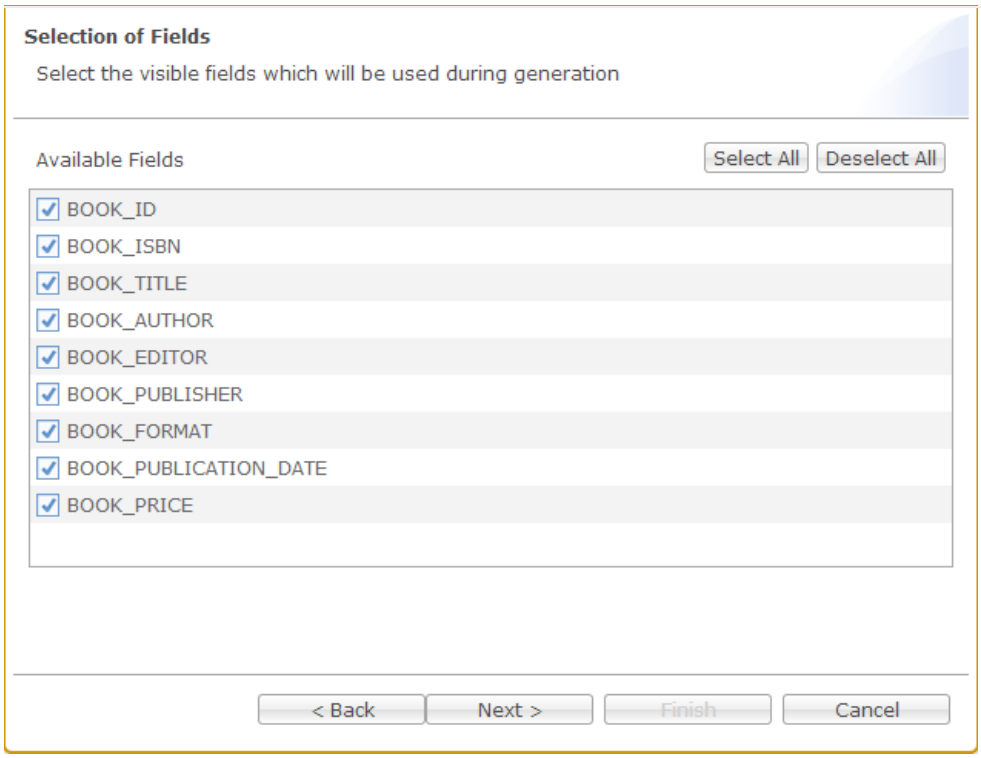
Select the *books.entity* and open the pop-up menu. Choose *Generate->User Interface for Entity Service*



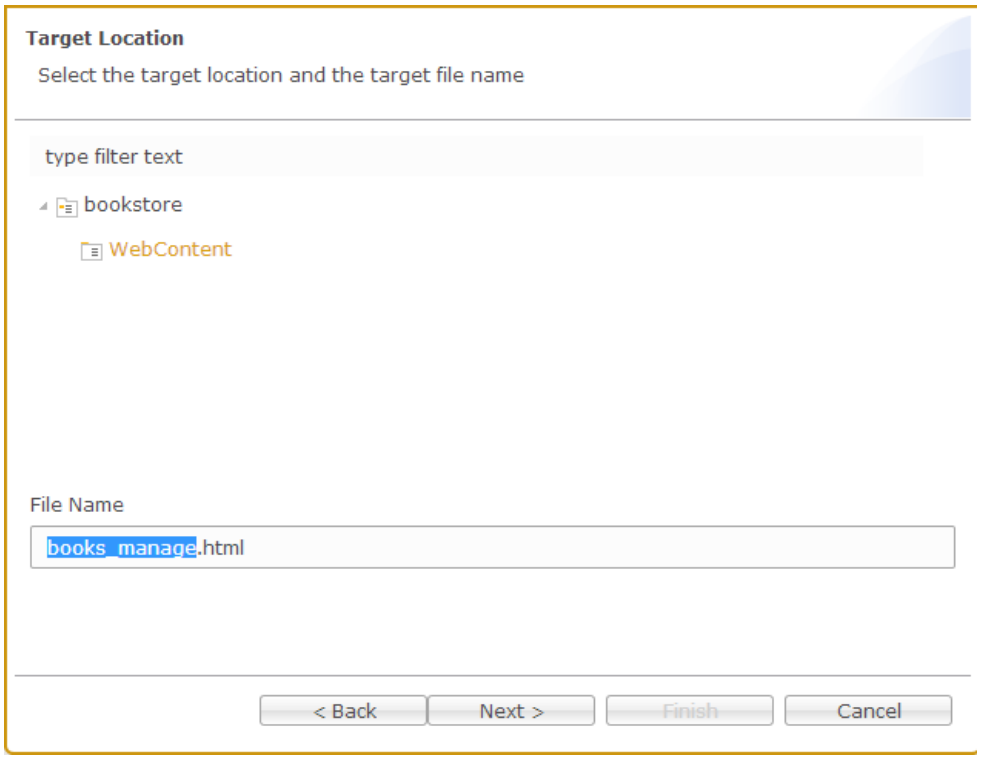
From the wizard select the template "List and Manage View"



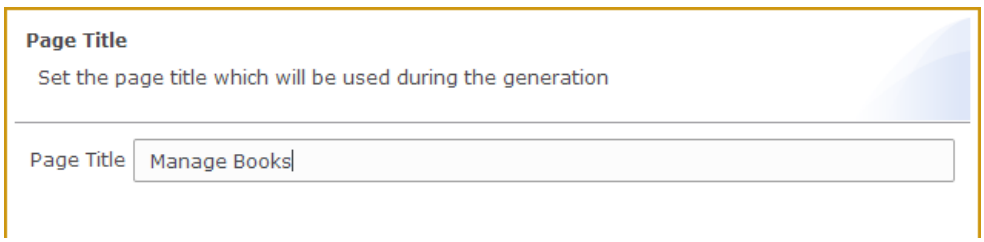
Click Next and select all the columns from the list. You can use "Select All" button



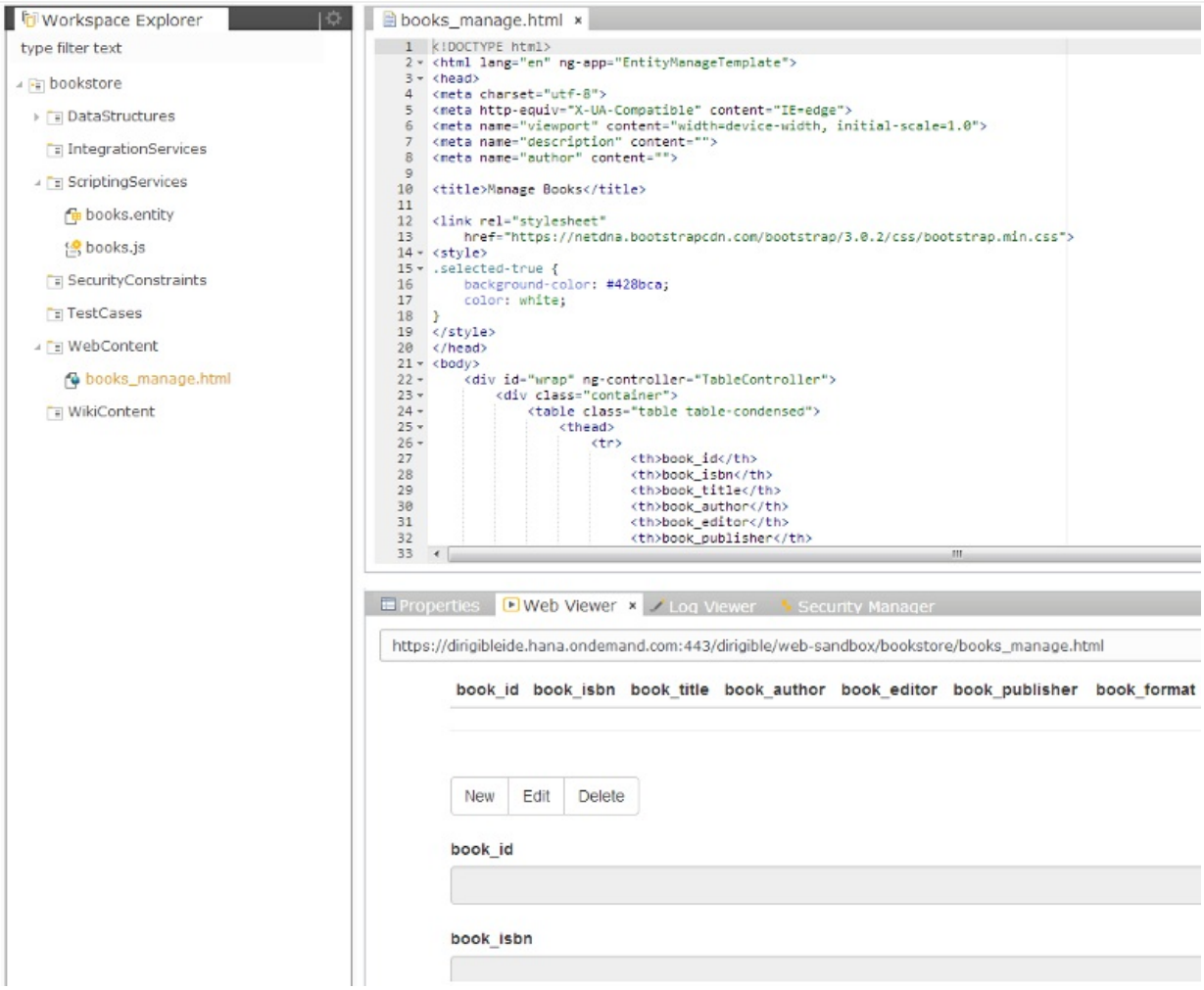
On the next page enter the name of the page *books_manage.html*



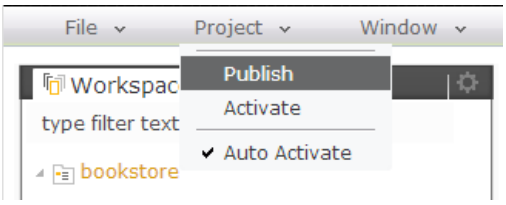
For the Title on the next page you can enter *Manage Books*



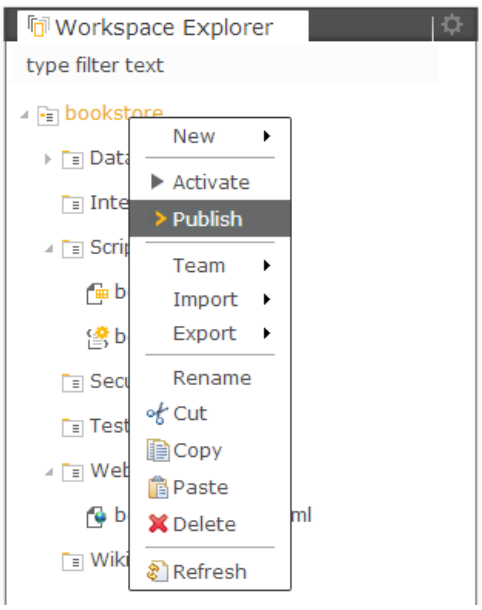
After clicking Finish button the generation is triggered. You can see the result under the WebContent folder When you select the file with active Preview you shall see the resulted running page.



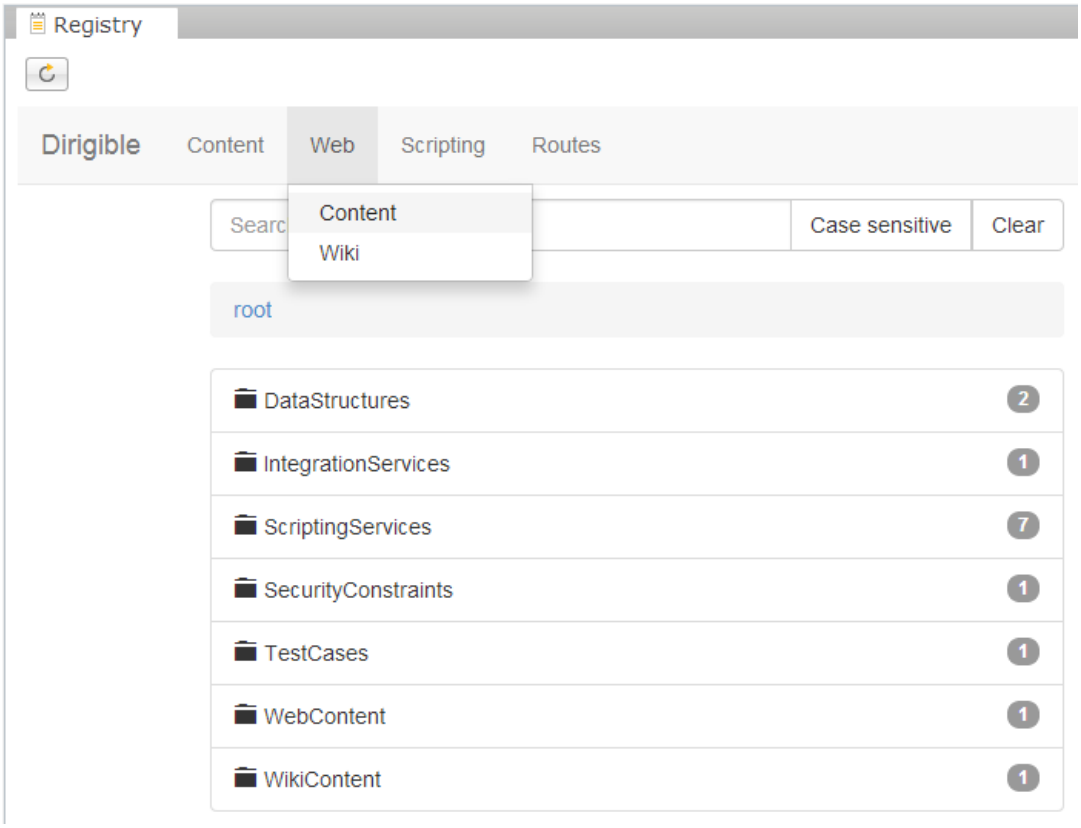
For the real test of the web page and the entity service you can Publish the project



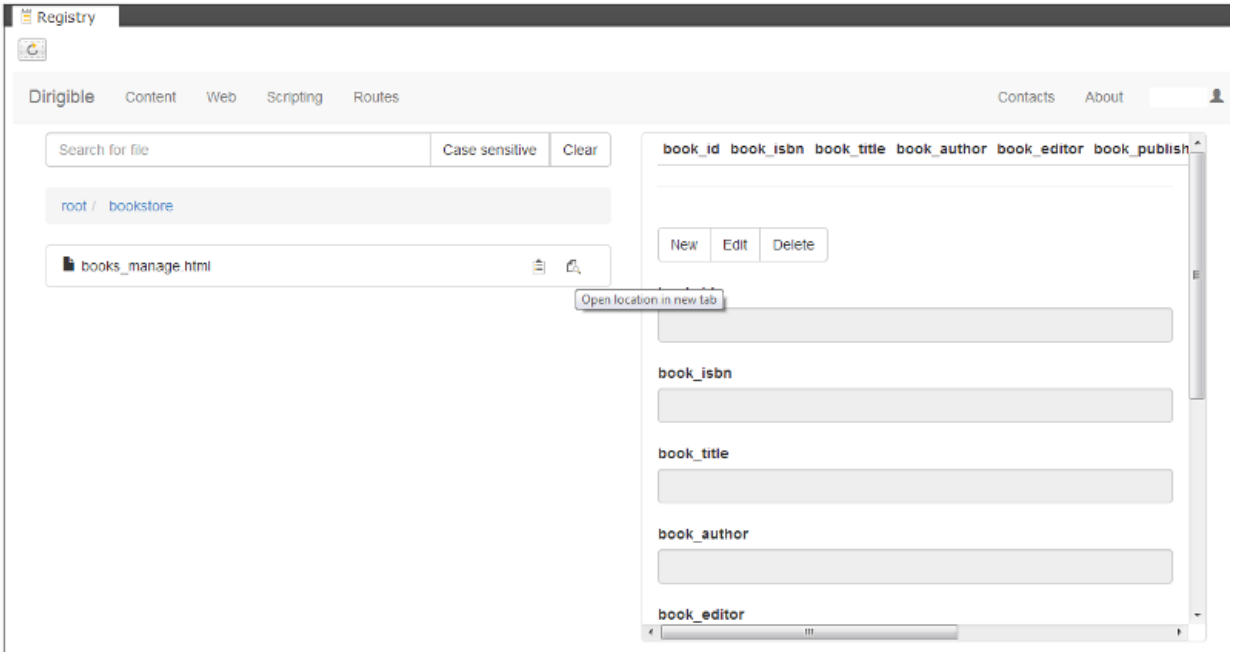
or



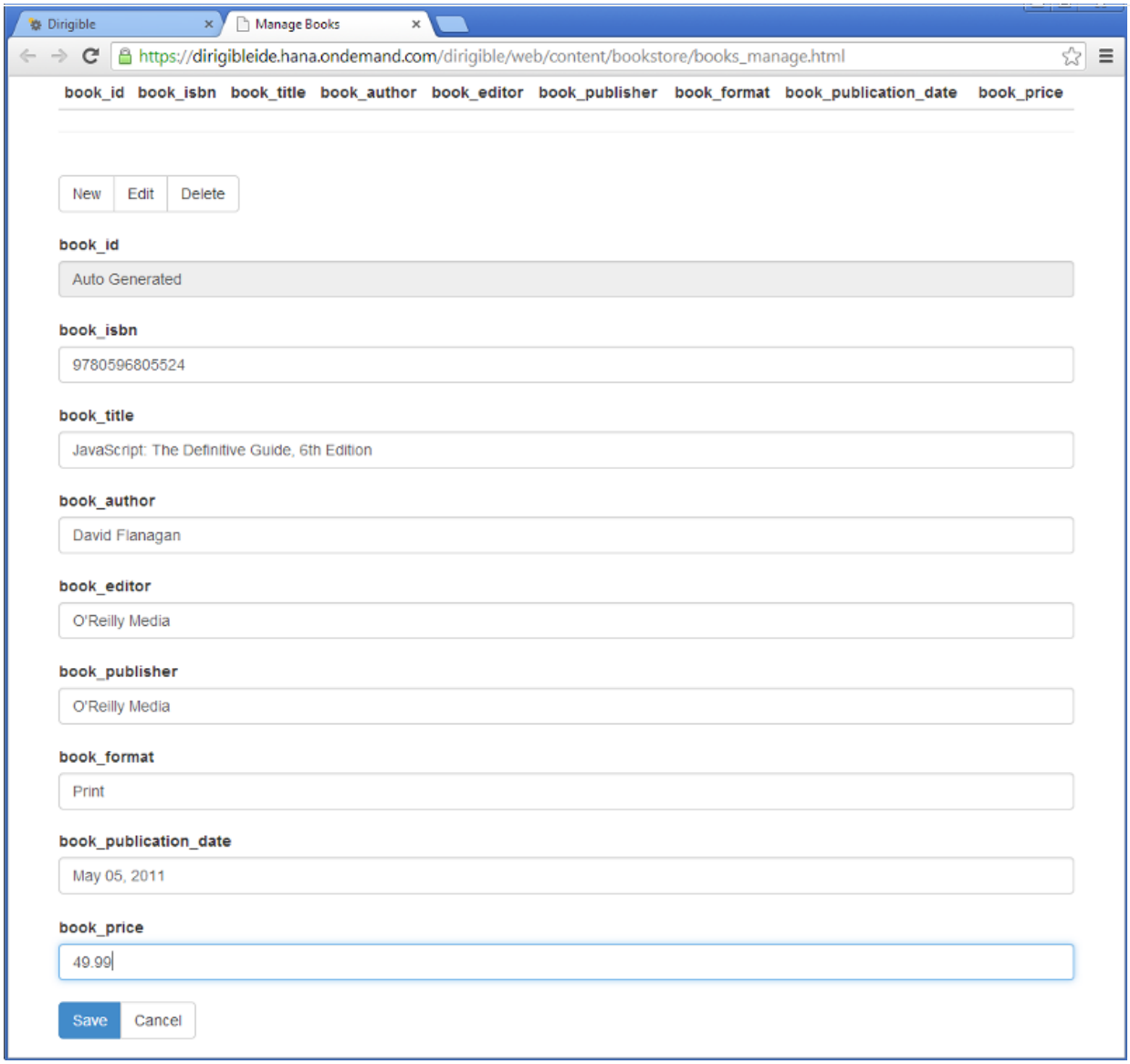
Now fo to the Registry perspective to find the link to the page, so that we can open it in an external browser. From the Registry embedded page menu choose Web->Content



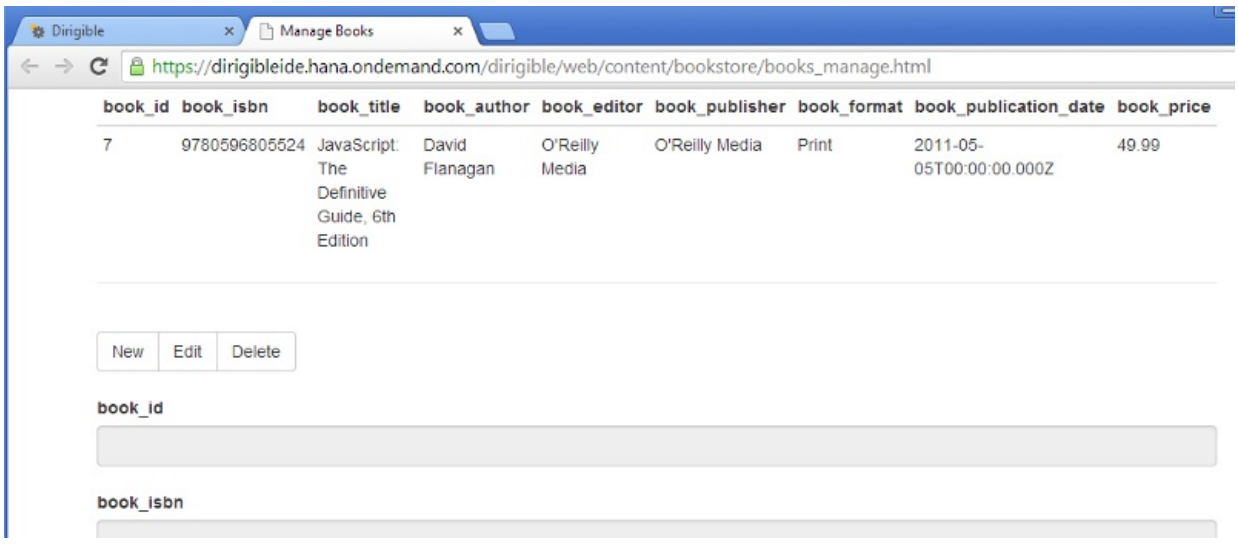
Drill-down in the bookstore project folder and click on the page which is listed. To open the page in a new tab click on the icon on the right side



Click on "Edit" button and input the information about the first book you want to have in your store.



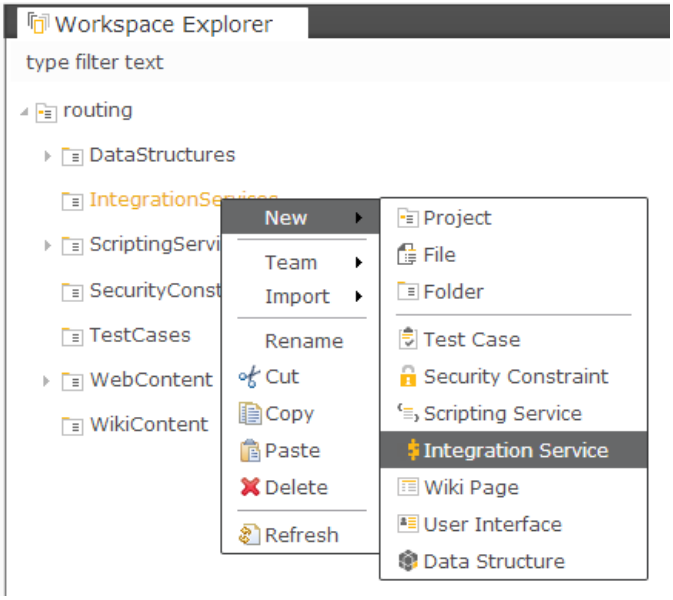
Click Save button and see the inserted record in the table above



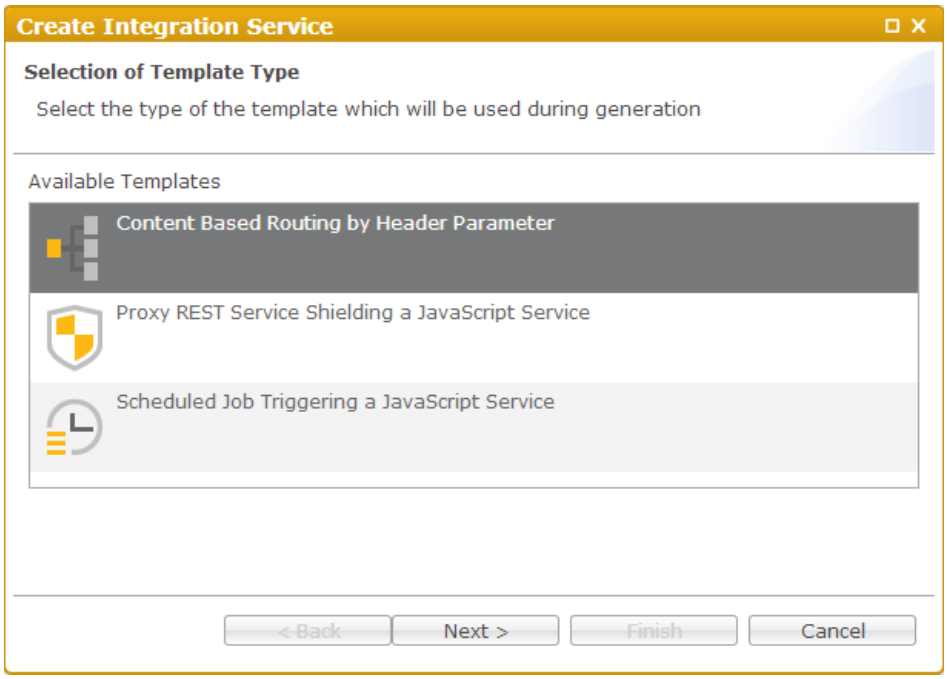
Simple Routing

The simplest *Integration Service* you can have is the one using the built in conditioning DSL of [Apache Camel|http://camel.apache.org]. General steps are valid for all the other template types for the different integration patterns.

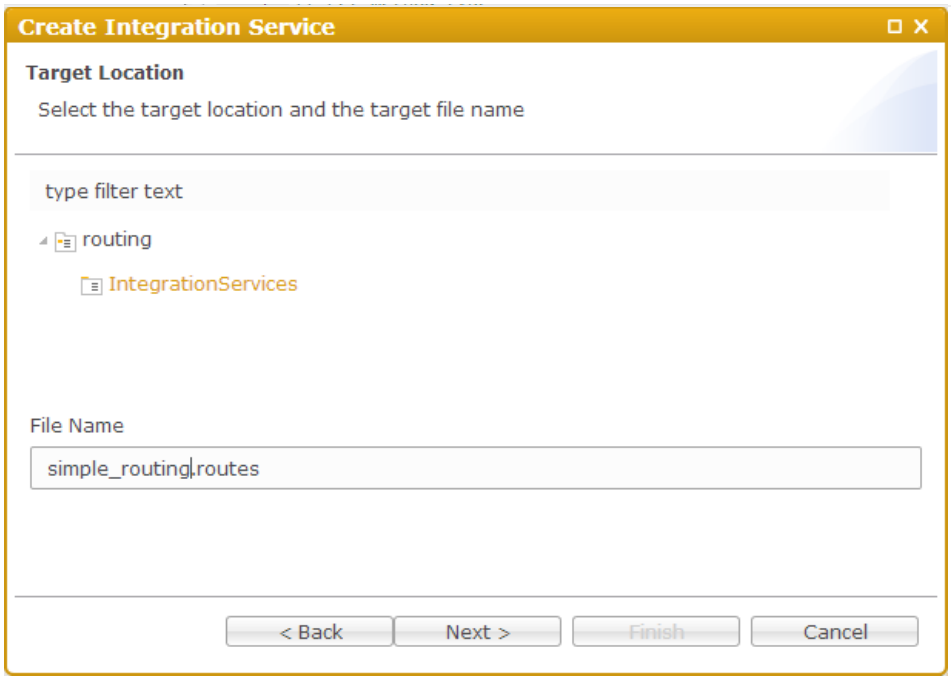
Create a simple project named *routing* (from main menu New->Project) Select just created project in the Workspace Explorer. From the pop-up menu select *New->Integration Service*



From the template list in the wizard which is opened, chose "Content Based Routing by Header Parameter"

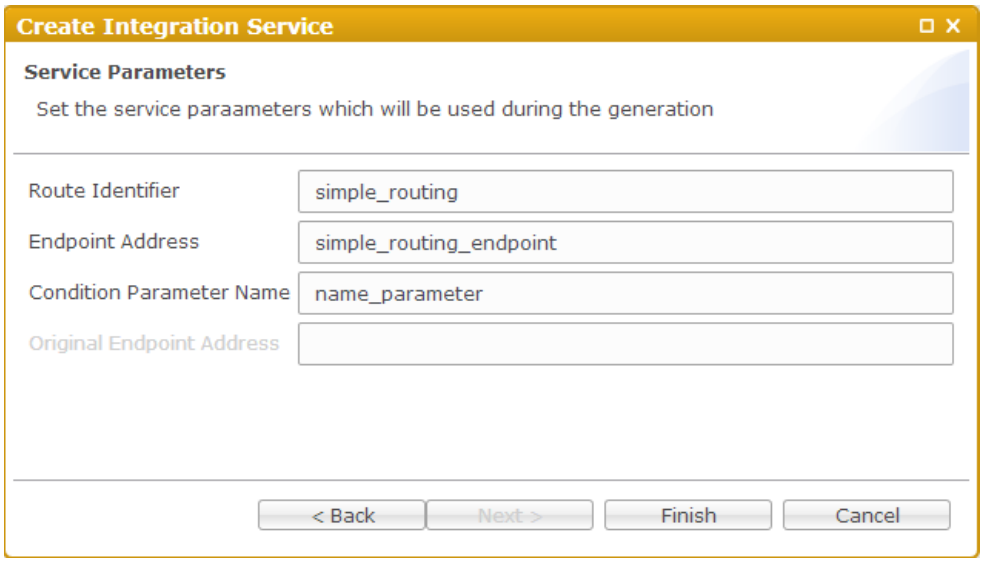


Click Next and give a name of the routes artifact - *simple_routing.routes*



On the next page enter the required parameters for this particular template:

> Route Identifier: *simple_routing* Endpoint Address: *simpleroutingendpoint* Condition Parameter Name: *name_parameter*



Click finish and inspect the generated artifact, which should look like this:

```
< routes xmlns="http://camel.apache.org/schema/spring">
    < route id="simple_routing">
        < from uri="servlet:///simple_routing_endpoint" />
        < choice>
            < when>
                < header>name_parameter
                < transform>
                    < simple>Hello ${header.name_parameter} how are you?
                < /transform>
            < /when>
            < otherwise>
                < transform>
                    < constant>Add a name parameter to uri, eg ?name_parameter=foo
                < /transform>
            < /otherwise>
        < /choice>
    < /route>
< /routes>
```

Activate or publish the project and it is done.

Go to location *http //[host]:[port]/dirigible/camel/simpleroutingendpoint* and try it.

The pattern which is used for generating the location URL can be found here

Shielding JavaScript Service

Often by security reasons the original end-point of a service should not be exposed directly. To make a simple redirect from public end-point to an internal end-point of a scripting service we can use the routing functionality in Integration Services section.

Create a project *routing* and a JavaScript service *service_impl.js* with the following content:

```
var systemLib = require('system');

systemLib.println("Hello World!");

response.getWriter().println("Hello World!");
response.getWriter().flush();
response.getWriter().close();
```

Using the menu create *New->Integration Service* and choose *Proxy REST Service Shielding a JavaScript Service* Enter the parameters:

> Route Identifier: *route_js* Endpoint Address: *routejsendpoint* Original Endpoint: *http //localhost:9001/dirigible/js/routing/service_impl.js*

Click finish. The generated routes artifact should looks like:

```
< routes xmlns="http://camel.apache.org/schema/spring">
    < route id="route_js">
        < from uri="servlet:///route_js_endpoint" />
        < to uri="http://localhost:9001/dirigible/js/routing/service_impl.js?bridgeEndpoint=true"/>
    < /route>
< /routes>
```

Activate or publish the project and check the result at:

> *http //[host]:[port]/dirigible/camel/routejsendpoint*

In general you can bridge any local service using this approach

Scheduled Job

In the real-world applications there is a need of some tasks to be executed in the background, without user interaction. For this purpose we use the well accepted term Job and this example shows how to schedule it to be executed periodically.

Create a project *routing* and a JavaScript service *service_job.js* with the following content:

```
var systemLib = require('system');

systemLib.println('Scheduled Job Triggered at: '  + new Date());
```

Using the menu create *New->Integration Service* and choose *Scheduling Job Triggering a JavaScript Service* Enter the parameters:

> Route Identifier: *route_js* Endpoint Address: *http //localhost:9001/dirigible/js/routing/service_job.js*

Click finish. The generated routes artifact should looks like:

```
< routes xmlns="http://camel.apache.org/schema/spring">
    < route id="route_job">
        < from uri="timer://route_job?period=10000&repeatCount=10&fixedRate=true" />
        < to uri="http://localhost:9001/dirigible/js/routing/service_job.js"/>
    < /route>
< /routes>
```

Scheduled Job with Quartz

In the integrated Camel engine you can use the most popular job scheduling library - Quartz. A simple job which is triggering at midnight every day and prints in the default output some message looks like following:

```
< routes xmlns="http://camel.apache.org/schema/spring">
    < route id="jobId">
        < from uri="quartz://groupName/timerName?cron=0+0+0+*+*+?" />
        < transform>
            < simple>Hello from Quartz< /simple>
        < /transform>
        < to uri="stream:out"/>
    < /route>
< /routes>
```

REST Call

Create a *Project RESTCall* and a *JavaScript* service *rest_call.js* with the following content:

GET Call

```
var ioLib = require('io');



var url = 'http://rest.call/example';



var getRequest = http.createGet(url);
var httpClient = http.createHttpClient();
var httpResponse = httpClient.execute(getRequest);
var entity = httpResponse.getEntity();
var content = entity.getContent();



var input = ioLib.read(content);
http.consume(entity);



response.getWriter().println(input);
```

Parsing *input* to JSON

```
var json = JSON.parse(input);
```

POST Call

```
var postRequest = http.creatPost(url);
```

PUT Call

```
var putRequest = http.createPut(url);
```

DELETE Call

```
var deleteRequest = http.createDelete(url);
```

For more information check the *API* documentation

REST Call with Authorization Header

Most of the RESTfull services playing role as remote APIs require some kind of authentication. In the following example we use *Basic* scheme with authorization header as it specified at RFC 2617. The target service could be OData service, from which we explicitly ask for JSON format as well.

Create a *Project Authorization* and a *JavaScript* service *restcallauthorization.js* with the following content:

GET Call

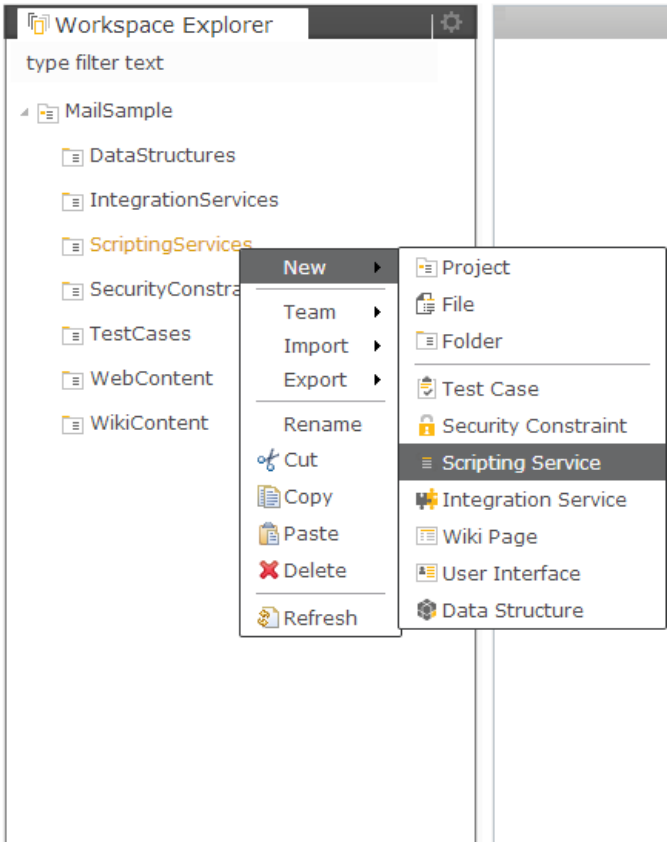
```
var ioLib = require('io');



var url = 'http://rest.call/example';
var user = 'user1';
var password = 'secret1';



var getRequest = http.createGet(url);
var httpClient = http.createHttpClient(true);
var credentials = http.createUsernamePasswordCredentials(user, password);

var scheme = http.createBasicScheme();
var authorizationHeader = scheme.authenticate(credentials, getRequest);
getRequest.addHeader(http.createBasicHeader("Accept", "application/json"));
getRequest.addHeader(authorizationHeader);



var httpResponse = httpClient.execute(getRequest);



var entity = httpResponse.getEntity();
var content = entity.getContent();



var input = ioLib.read(content);
http.consume(entity);



response.getWriter().println(input);
```

For more information check the *API* documentation

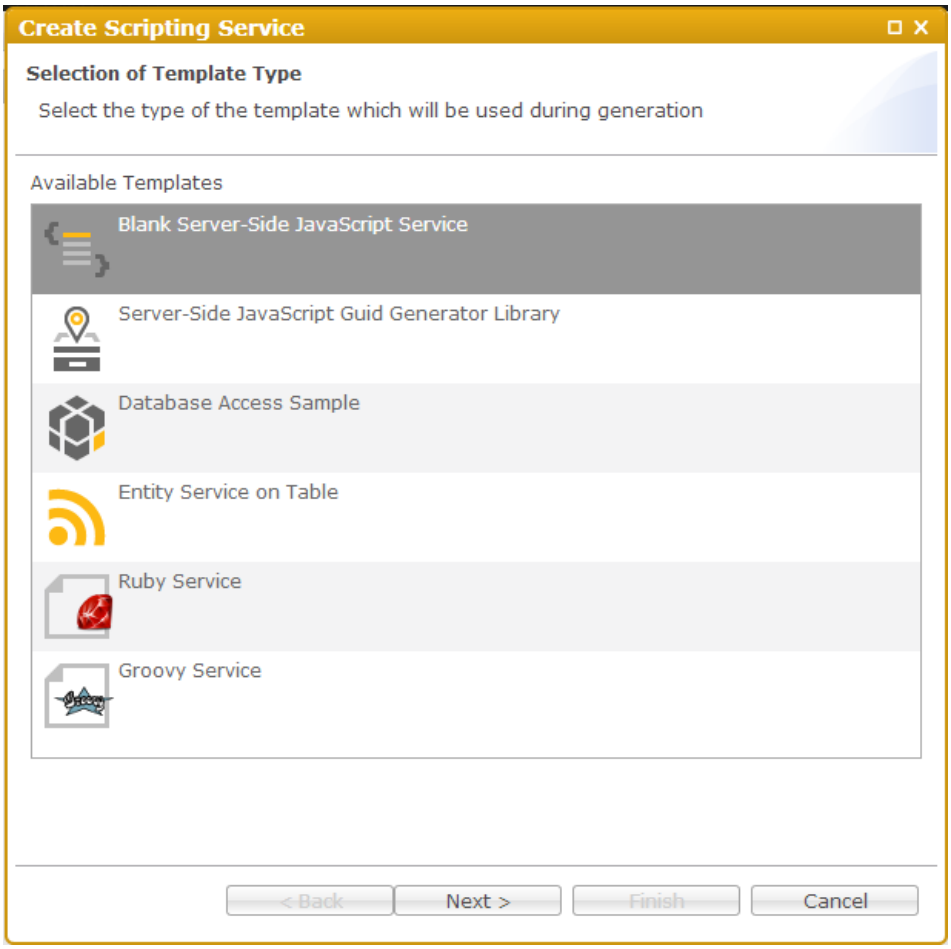
Mail Service

Create new project or use existing one.

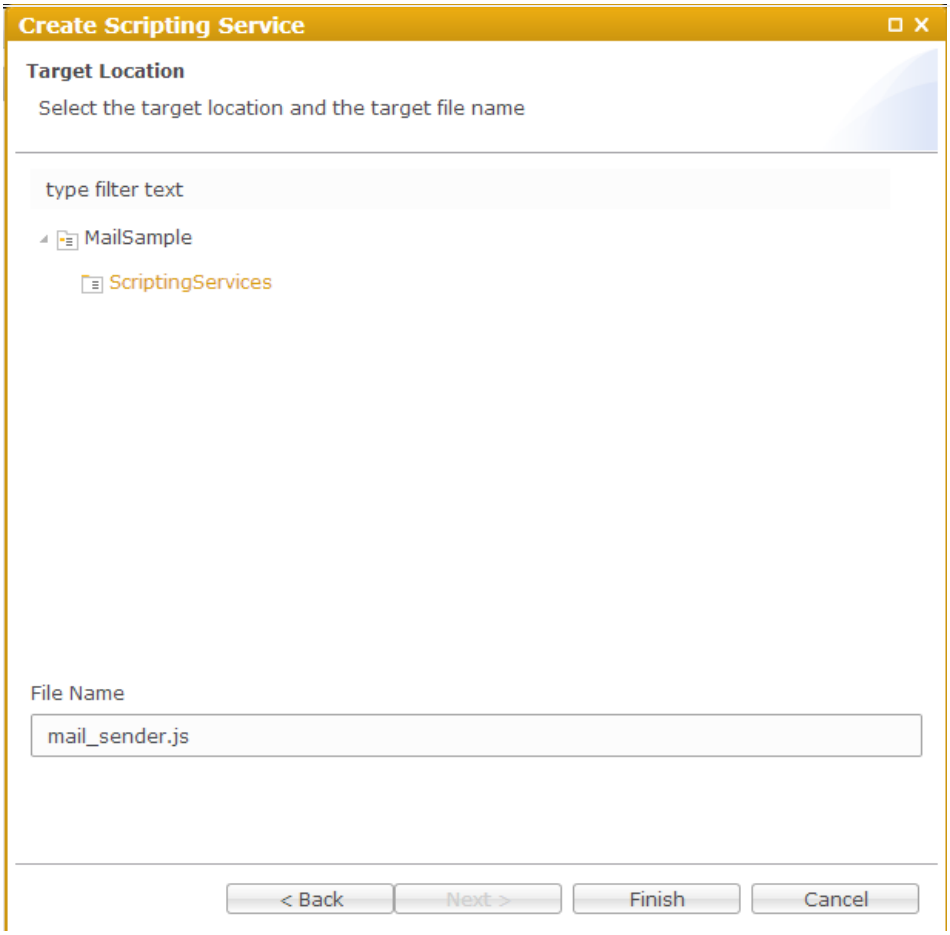
Create new *Scripting Service*



Choose *Blank Server-Side JavaScript Service* from the list of available templates



Give it some meaningful name (e.g *mail_sender.js*)

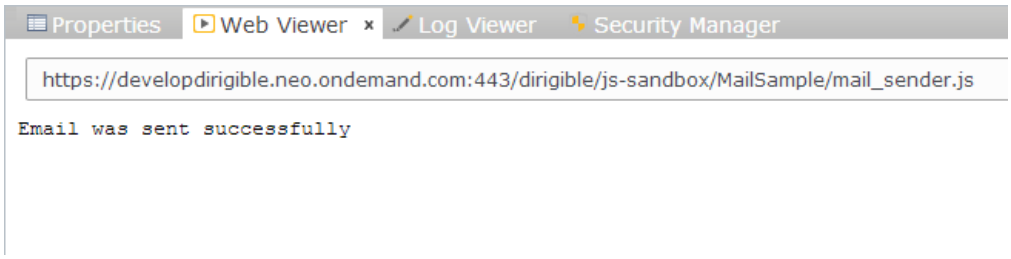


Now the project structure should look like this

Replace the generated code in *mail_sender.js* with the following:

```
var from = 'employee@your.company.com';
var to = 'boss@your.company.com';
var title = 'Test Email Service';
var body = 'Hello Boss! The mail service is up and running!';
mail.sendMail(from, to, title, body);
response.getWriter().println('Email was sent successfully');
```

Select *Preview* tab. Click on *mail_sender.js* from the *Workspace Explorer*. Accessing the scripting service will send the email.

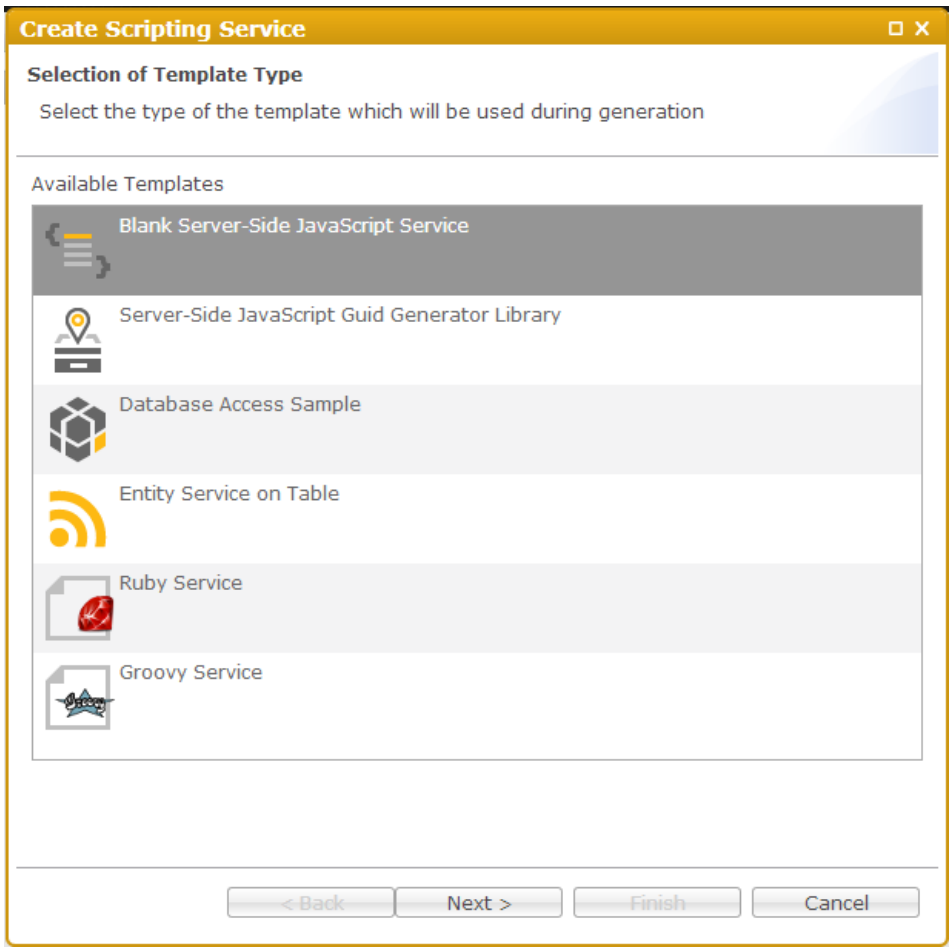


Config Sample

Create new project or use existing one.

Create new *Scripting Service*

Choose *Blank Server-Side JavaScript Service* from the list of available templates



Give it some meaningful name (e.g *config.js*)

Replace the generated code in *filestorageupload.js* with the following:

```
var ioLib = require("io");

var method = request.getMethod();
if (method == "POST") {
    var input = ioLib.read(request.getReader());
    var message = JSON.parse(input);

    if(message.path && message.key && message.value){
        config.putProperty(message.path, message.key, message.value);
    }
} else if (method == "GET") {
    var list = xss.escapeSql(request.getParameter("list"));
    var path = xss.escapeSql(request.getParameter("path"));
    var key = xss.escapeSql(request.getParameter("key"));

    if(list && path){
        var properties = config.getProperties(path);
        if (properties) {
            properties.list(response.getWriter());
        } else {
            response.getWriter().println("No configs found on path '" + path +"'");
        }
    } else if(path && key){
        response.getWriter().println("" + config.getProperty(path, key));
    }
} else if (method == "DELETE") {
    config.clear();
    response.getWriter().println("Config cleared!");
}

response.getWriter().flush();
response.getWriter().close();
```

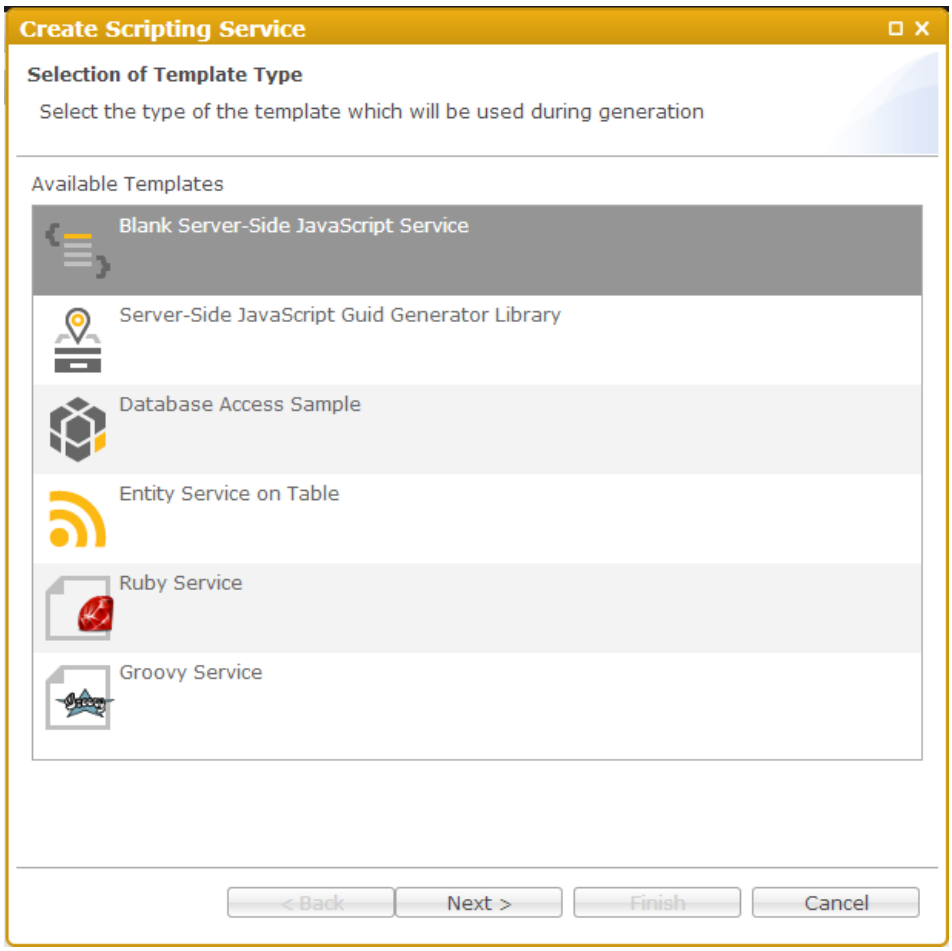
> getProperty(path, key) - get property value by given path
>
> getProperties(path) - retrieves properties by given path
>
> putProperty(path, key, value) - add property at given path
>
> putProperties(path, properties) - add properties at given path
>
> delete(path) - removes properties by given path
>
> clear() - removes all properties from the storage

Storage Sample

Create new project or use existing one.

Create new *Scripting Service*

Choose *Blank Server-Side JavaScript Service* from the list of available templates



Give it some meaningful name (e.g *storage_usage.js*)

Replace the generated code in *storage_usage.js* with the following:

```
function toString(bytes) {
    var s = "";
    for(var i=0, l=bytes.length; i < l; i++) {
        s += String.fromCharCode(bytes[i]);
    }
    return s;
}

var byteArray = [49,50,51]; // string "123"
storage.put("/a/b/c", byteArray);
var retrievedData = storage.get("/a/b/c");
var result = toString(retrievedData);

response.getWriter().println(result);

response.setContentType("text/html");
response.getWriter().flush();
response.getWriter().close();
```

Select *Preview* tab. Click on *storage_usage.js* from the *Workspace Explorer* and check the result.

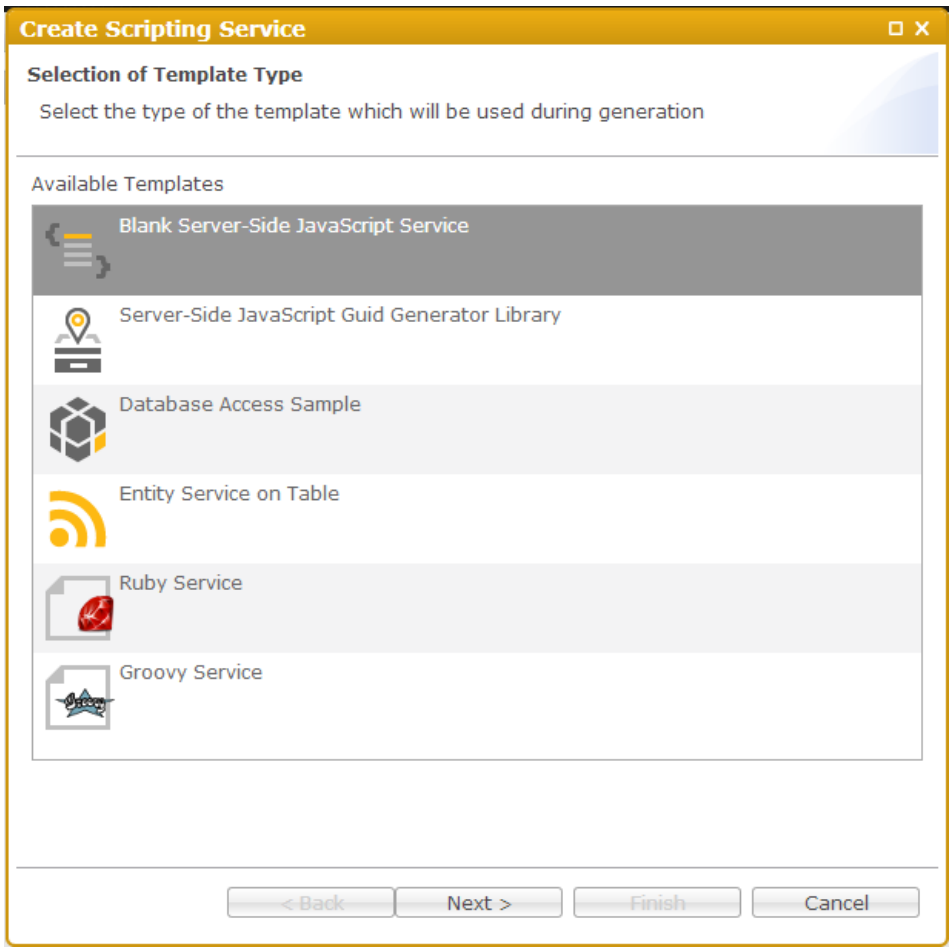
> put(path, data) - add binary data at given path
>
> get(path) - retrieves binary data by given path
>
> delete(path) - removes binary data by given path
>
> clear() - removes all the data from the storage

File Storage Sample

Create new project or use existing one.

Create new *Scripting Service*

Choose *Blank Server-Side JavaScript Service* from the list of available templates



Give it some meaningful name (e.g *file_upload.js*)

Replace the generated code in *file_upload.js* with the following:

```
var uploadLib = require("upload");
if(request.getMethod()=="POST"){
    var files = uploadLib.consumeFiles(request);
    for(var i = 0 ; i < files.length; i++){
        var file = files[i];
        fileStorage.put(file.name, file.data, file.contentType);
    }
}
```

Create new *Scripting Service*

Choose *Blank Server-Side JavaScript Service* from the list of available templates

Give it some meaningful name (e.g *file_download.js*)

Replace the generated code in *file_download.js* with the following:

```
if(request.getMethod()=="GET"){
    var fileName = xss.escapeSql(request.getParameter("fileName"));
    var file = fileStorage.get(fileName);

    if(file){
        response.setHeader("content-disposition", "inline");
        response.setHeader("content-disposition", "attachment; filename="+fileName);
        response.setContentType(file.contentType);
        response.setContentLength(file.data.length);
        io.write(file.data, response.getOutputStream());
    }else{
        response.getWriter().println("No file with name '" + fileName + "' found");
        response.setContentType("text/html");
    }
}
response.getWriter().flush();
response.getWriter().close();
```

> put(path, data, contentType) - add file at given path
>
> get(path) - retrieves file by given path
>
> delete(path) - removes file by given path
>
> clear() - removes all files from the storage

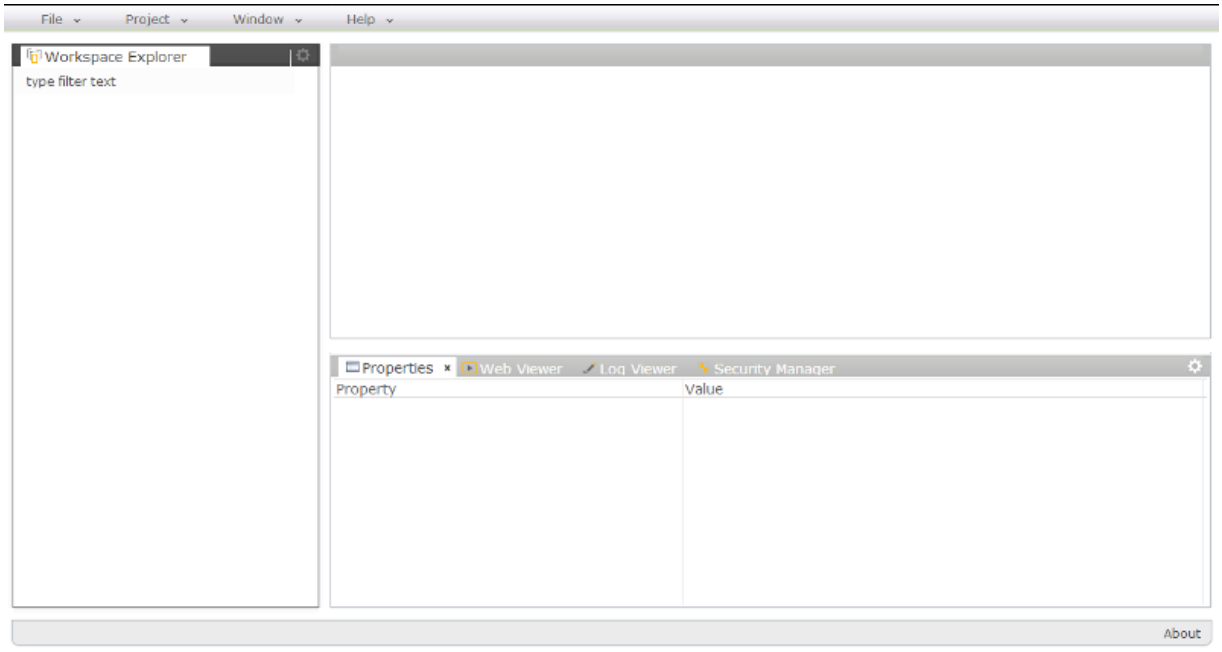
BookStore Sample

Overview

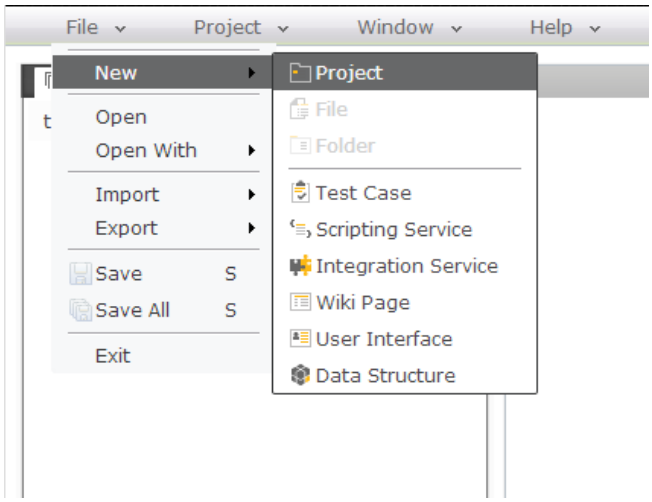
BookStore sample follows the steps required to build a simple online shop. It will show you how to create the domain model in the target database, how to generate the RESTful services on top of the modeled entities and finally how to generate simple management user interface as a basis for the actual web design later.

Project Creation

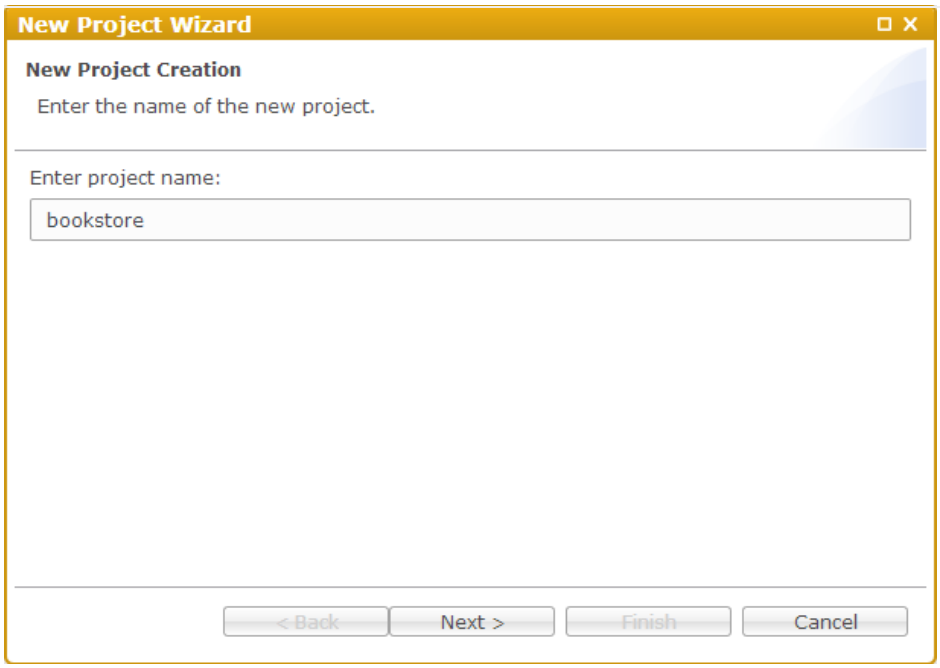
First you need to create a project in your private workspace. Workspace perspective is the default one.



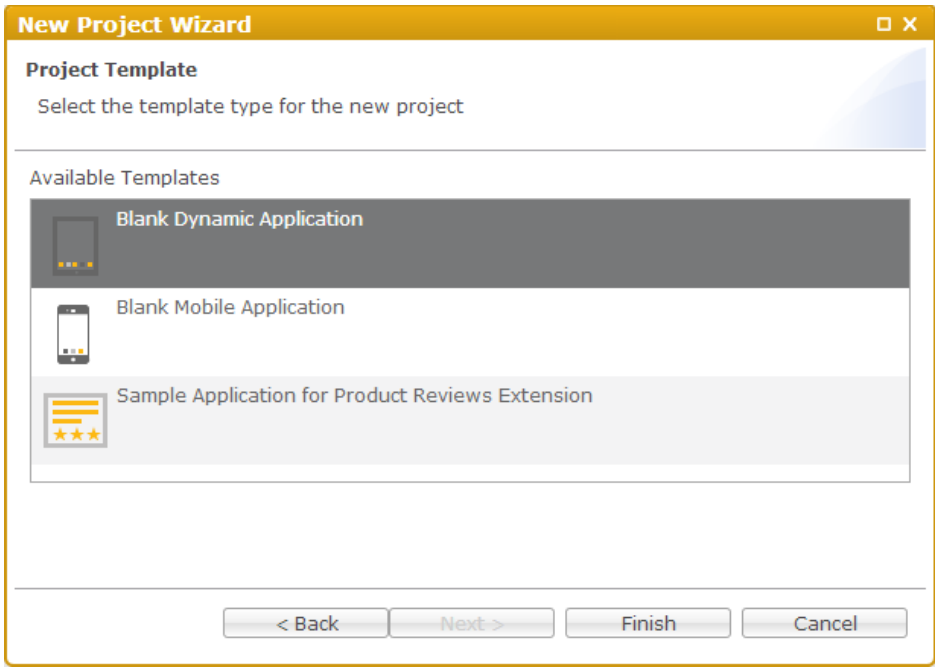
Go to the main menu *File->New->Project*



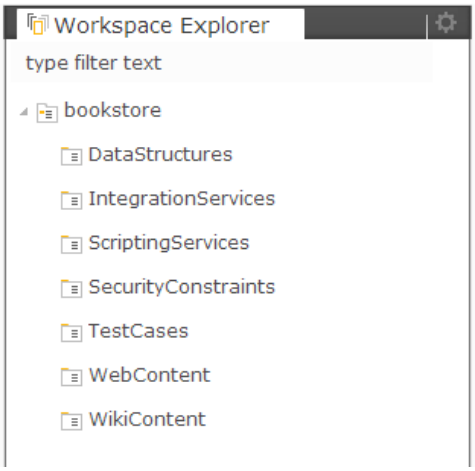
In the just opened wizard enter the project name - "bookstore" and click Next



From the list you can choose from several predefined project templates. In this case just go to the first one - "Blank Dynamic Application"



Click Finish and open the project in the workspace to see the folder layout

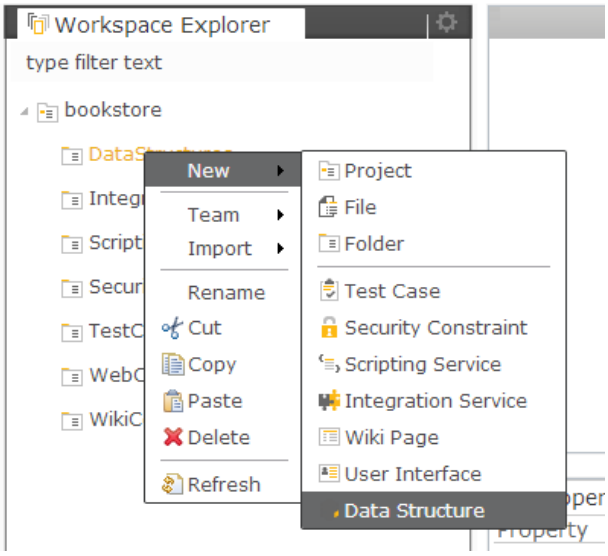


There are several predefined folders which are tightly related to the types of the artifacts which can be placed there as well as the corresponding activities you can make on them.

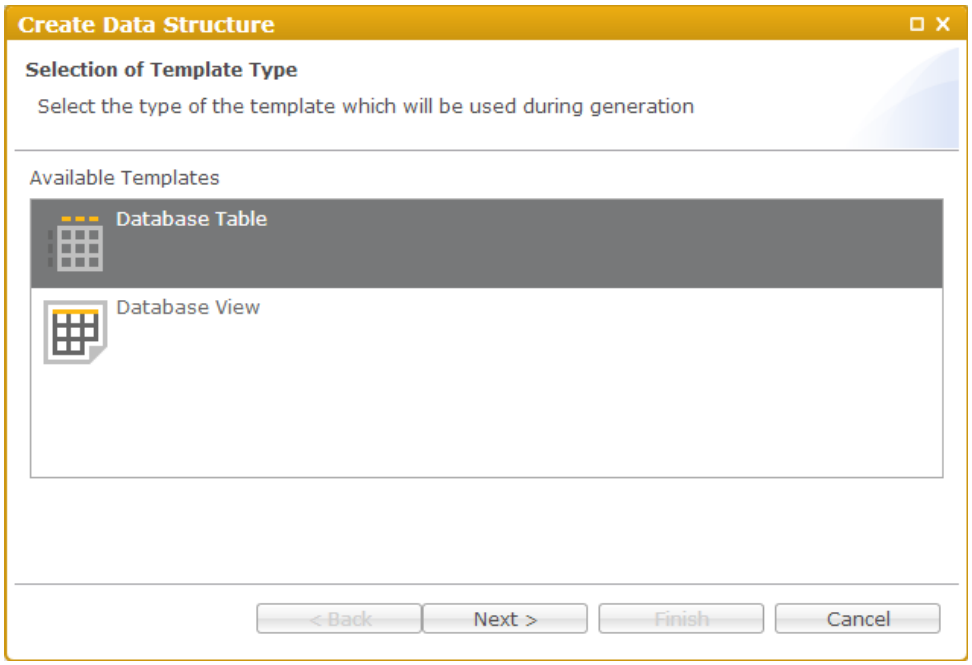
Data Model

Data models a.k.a domain models is the set of the entities of your application and also their relations. In Dirigible we use also the term data structures, which is more related to the actual artifact - the data descriptor. Let create the first model entity of the BookStore sample - the books table.

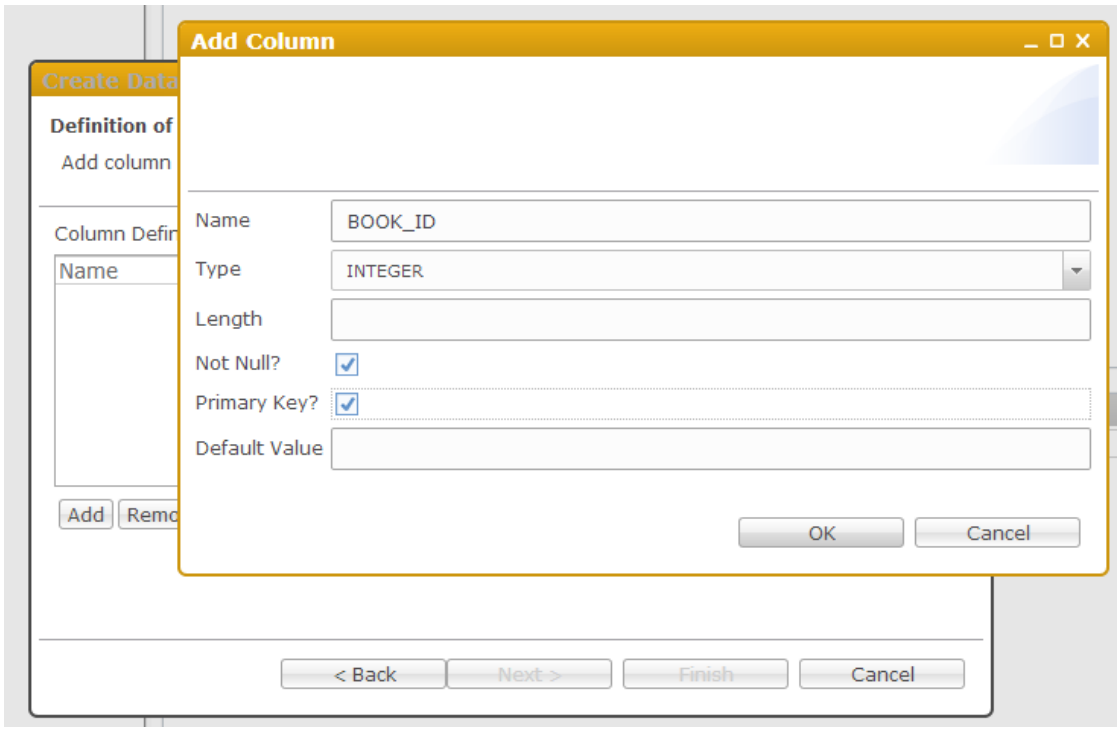
Select the DataStructures sub-folder of a project and open the pop-up menu (right-click). From the menu go to *New->Data Structure*



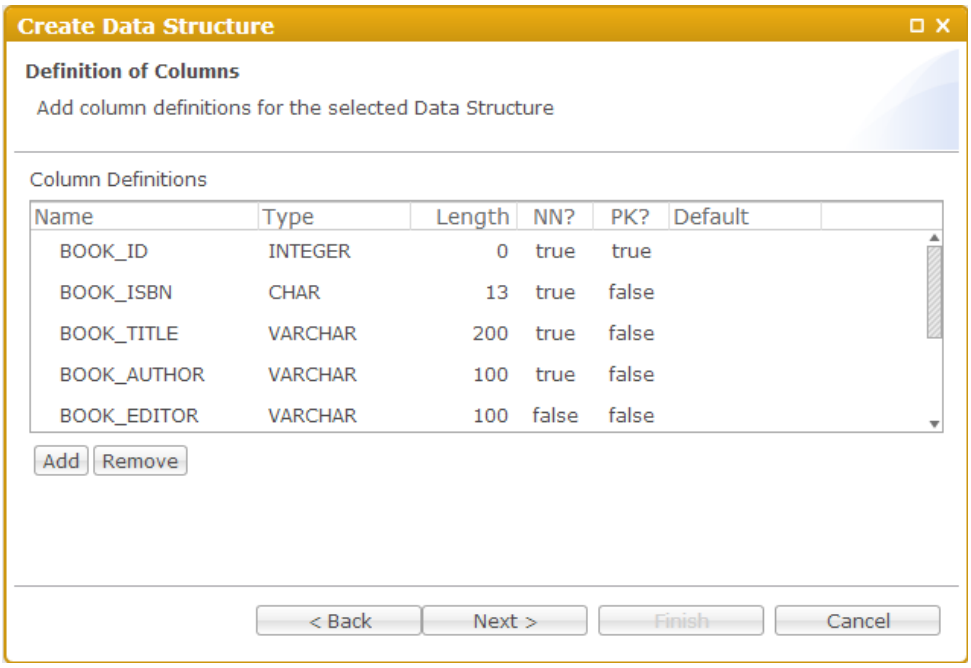
The first page of this wizard let you choose from the several artifacts related to the domain model. In this case we need a table where to store the books metadata like ISBN, Title, Author, etc. Choose "Database Table" and click Next



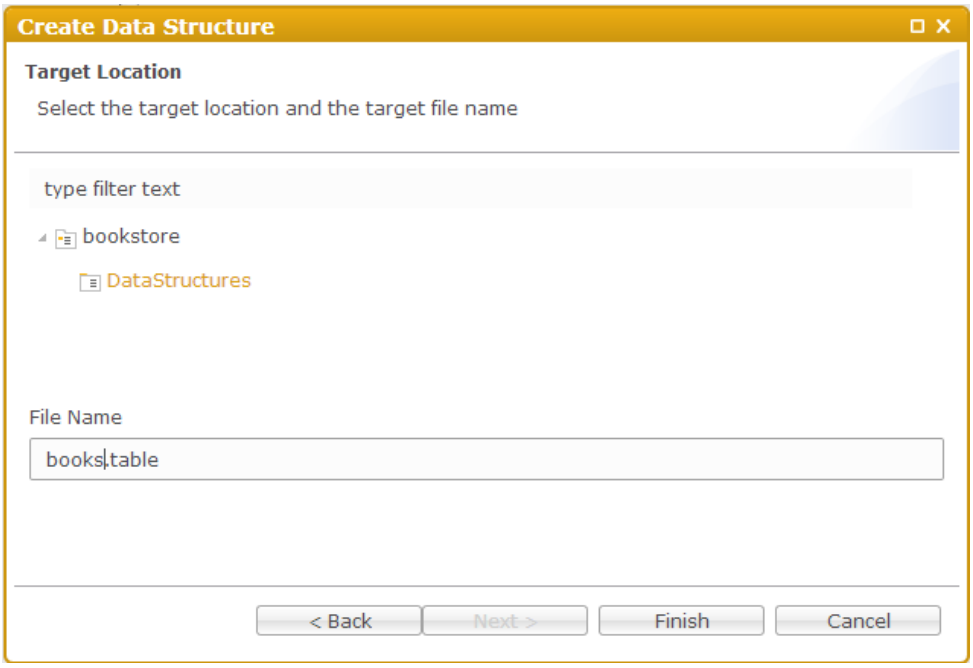
Use Add/Remove buttons to create the actual layout of the table



```
BOOK_ID                 INTEGER         not null    primaryKey
BOOK_ISBN               CHAR        13  not null
BOOK_TITLE              VARCHAR     200 not null
BOOK_AUTHOR             VARCHAR     100 not null
BOOK_EDITOR             VARCHAR     100
BOOK_PUBLISHER          VARCHAR     100
BOOK_FORMAT             VARCHAR     100
BOOK_PUBLICATION_DATE   DATE
BOOK_PRICE              DOUBLE          not null
```



Give a name and click finish



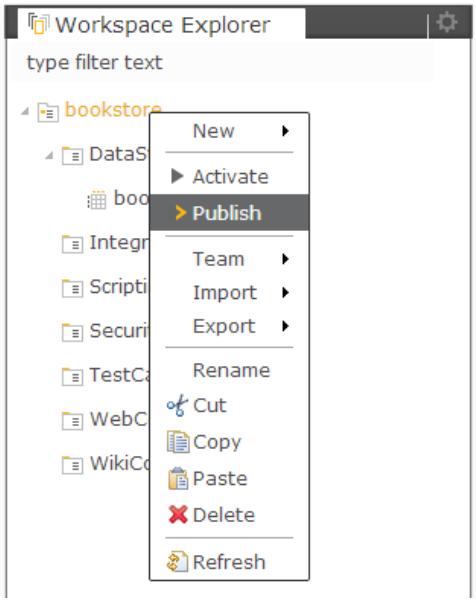
The table descriptor should be generated, based on your input and the file itself should be opened in the editors area

![New DataStructures Content]bookstore/11*books*new*ds*content.png!

```
{
  "tableName":"BOOKS",
  "columns":
    [
      {
        "name":"BOOK_ID",
        "type":"INTEGER",
        "length":"0",
        "notNull":"true",
        "primaryKey":"true",
        "defaultValue":""
      }
      ,
      {
```

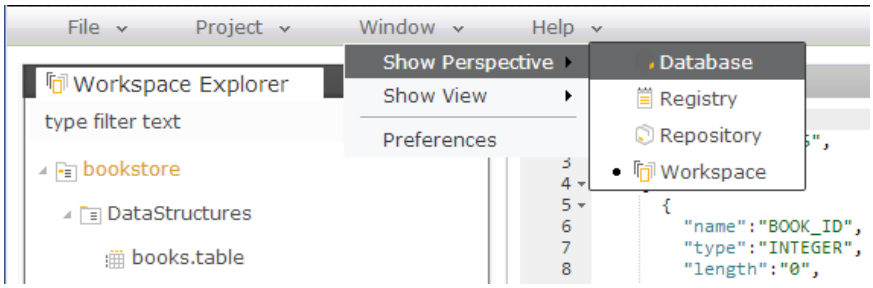
```
        "name":"BOOK_ISBN",
        "type":"CHAR",
        "length":"13",
        "notNull":"true",
        "primaryKey":"false",
        "defaultValue":""
      }
      ,
      {
        "name":"BOOK_TITLE",
        "type":"VARCHAR",
        "length":"200",
        "notNull":"true",
        "primaryKey":"false",
        "defaultValue":""
      }
      ,
      {
        "name":"BOOK_AUTHOR",
        "type":"VARCHAR",
        "length":"100",
        "notNull":"true",
        "primaryKey":"false",
        "defaultValue":""
      }
      ,
      {
        "name":"BOOK_EDITOR",
        "type":"VARCHAR",
        "length":"100",
        "notNull":"false",
        "primaryKey":"false",
        "defaultValue":""
      }
      ,
      {
        "name":"BOOK_PUBLISHER",
        "type":"VARCHAR",
        "length":"100",
        "notNull":"false",
        "primaryKey":"false",
        "defaultValue":""
      }
      ,
      {
        "name":"BOOK_FORMAT",
        "type":"VARCHAR",
        "length":"100",
        "notNull":"false",
        "primaryKey":"false",
        "defaultValue":""
      }
      ,
      {
        "name":"BOOK_PUBLICATION_DATE",
        "type":"DATE",
        "length":"0",
        "notNull":"false",
        "primaryKey":"false",
        "defaultValue":""
      }
      ,
      {
        "name":"BOOK_PRICE",
        "type":"DOUBLE",
        "length":"0",
        "notNull":"true",
        "primaryKey":"false",
        "defaultValue":""
      }
    ]
}
```

More about the data structure types and their descriptors can be found here Now we have to create the real database artifact in the underlying database. This can be done via publish action from the project's popup menu.

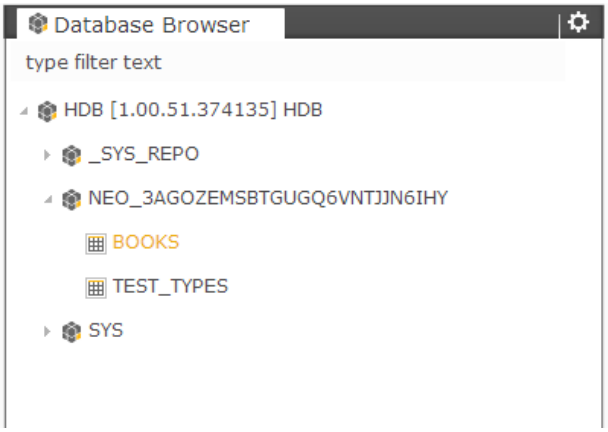


In this case (data structures artifacts) you can do the same also with activation action. Follow the links for more information about the differences between activation and publishing.

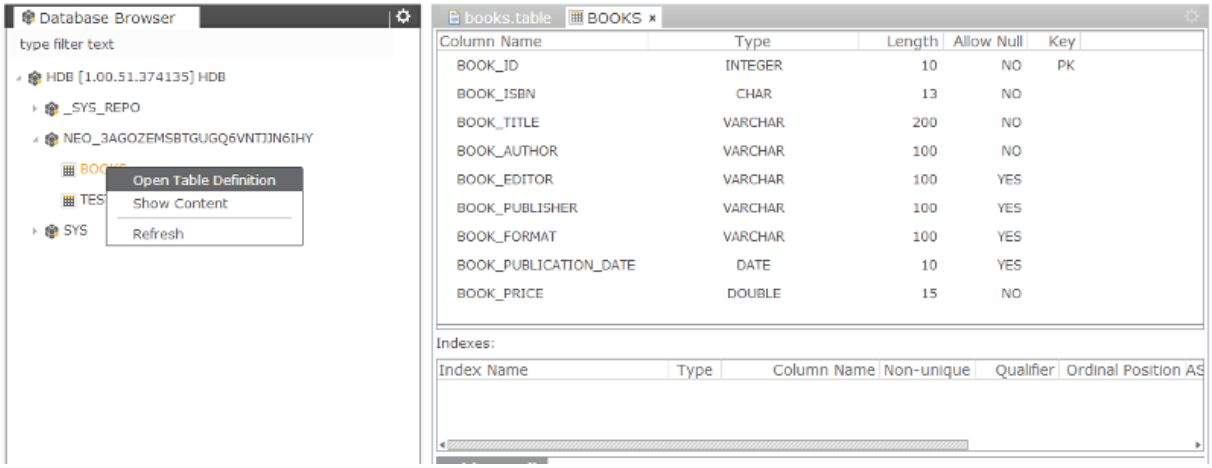
Once we have published successfully the project, we can go to Database perspective to double-check the table definition. From main menu go to Window->Show Perspective->Database



Open the database schema node and find the BOOKS table. Right-click and choose "Open Table Definition"



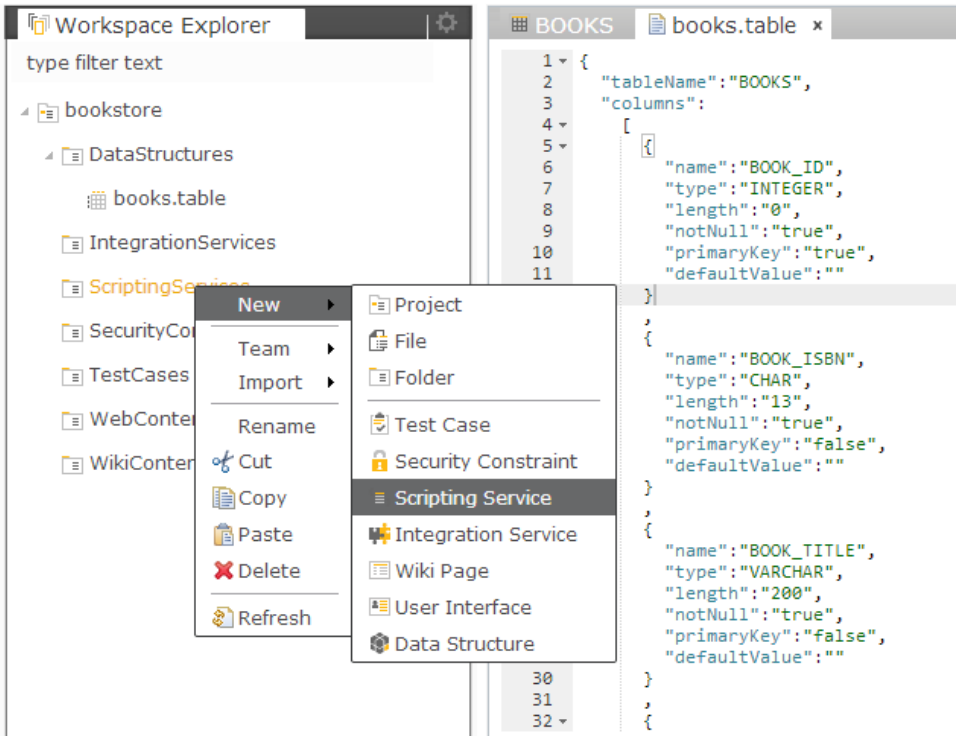
This will open the Table Definition Viewer



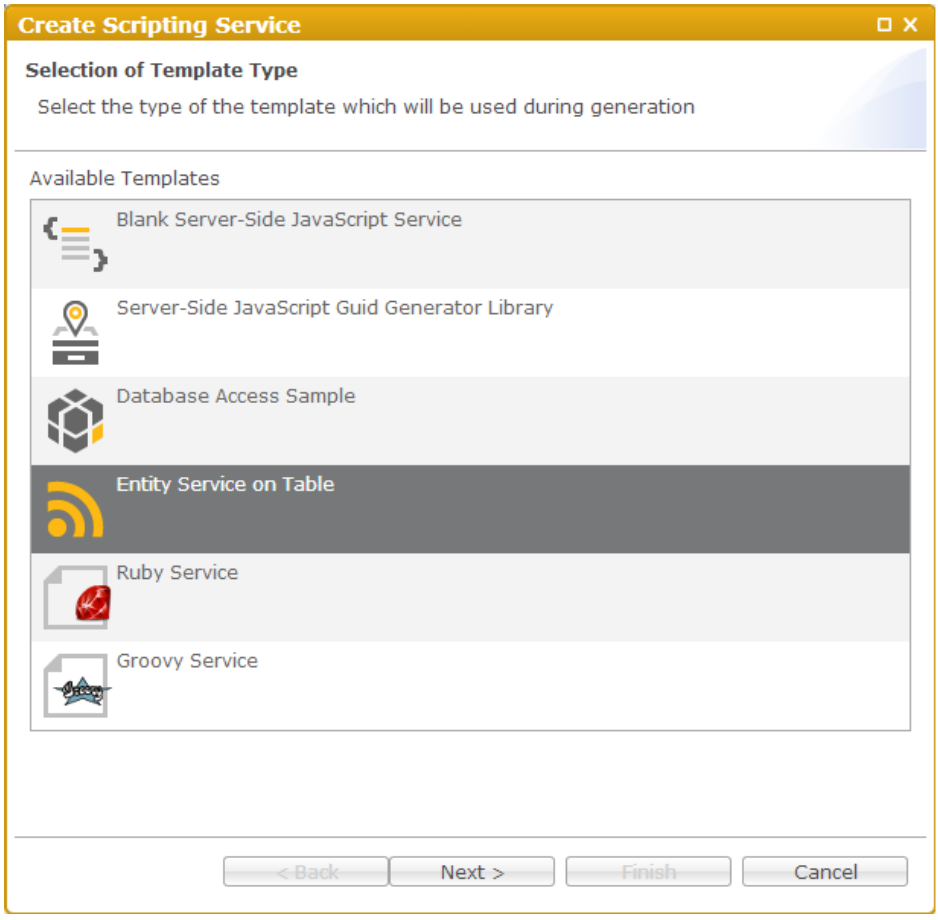
Entity Service

Entity Service meaning in terms of Dirigible is a RESTful service, which exposes the [CRUD](http://en.wikipedia.org/wiki/Create,*read*,update*and*delete] methods on top of the database table. The following steps shows how to generate such an entity service on top of existing table.

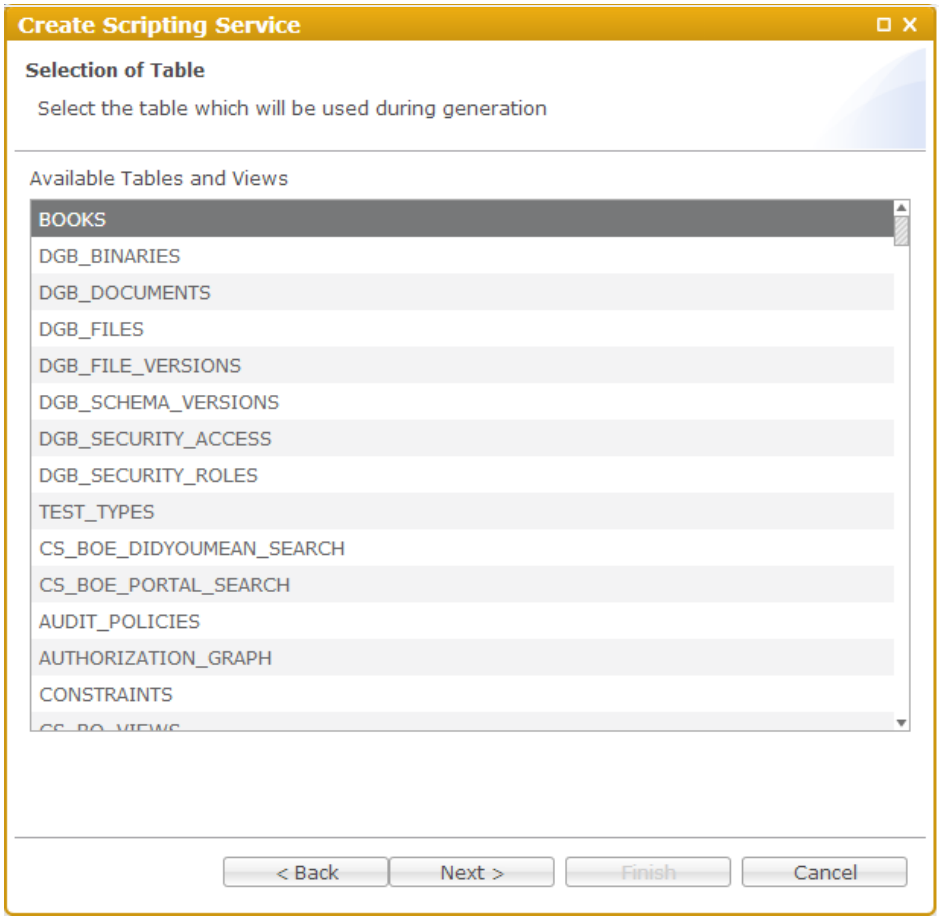
Select the ScriptingServices sub-folder of the project and open the pop-up menu From the menu go to *New->Scripting Service*



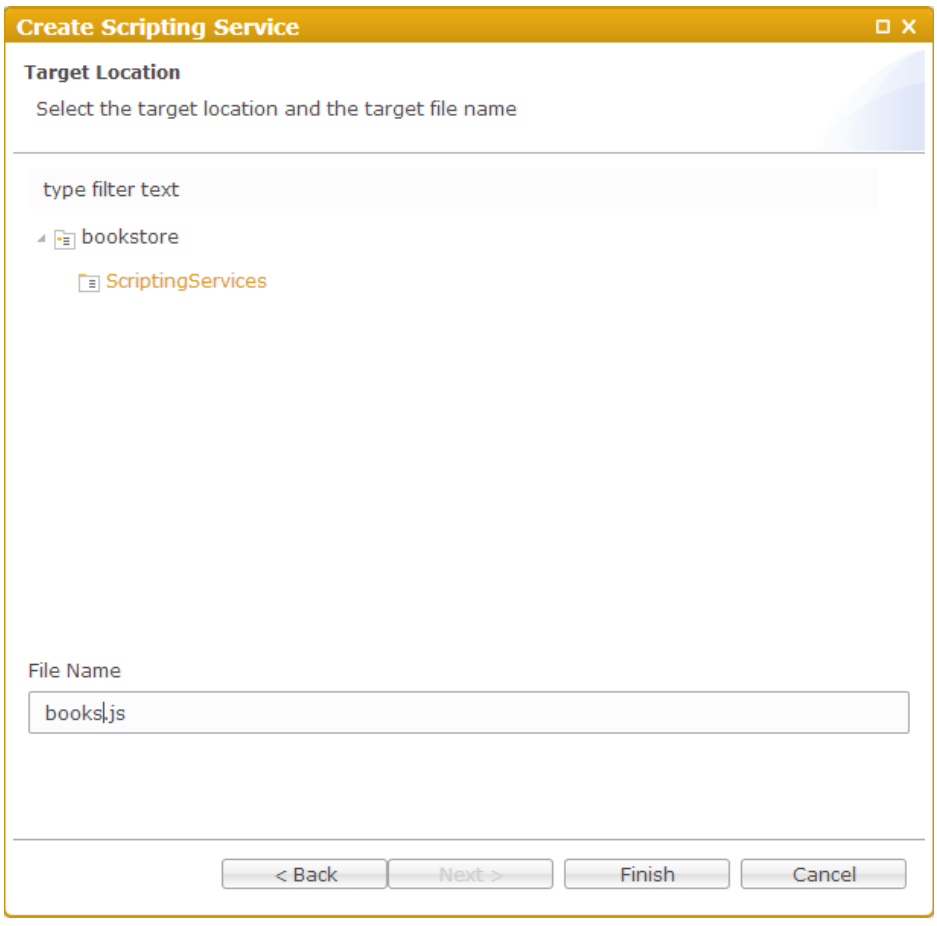
From the opened wizard select the "Entity Service" from the list of available templates.



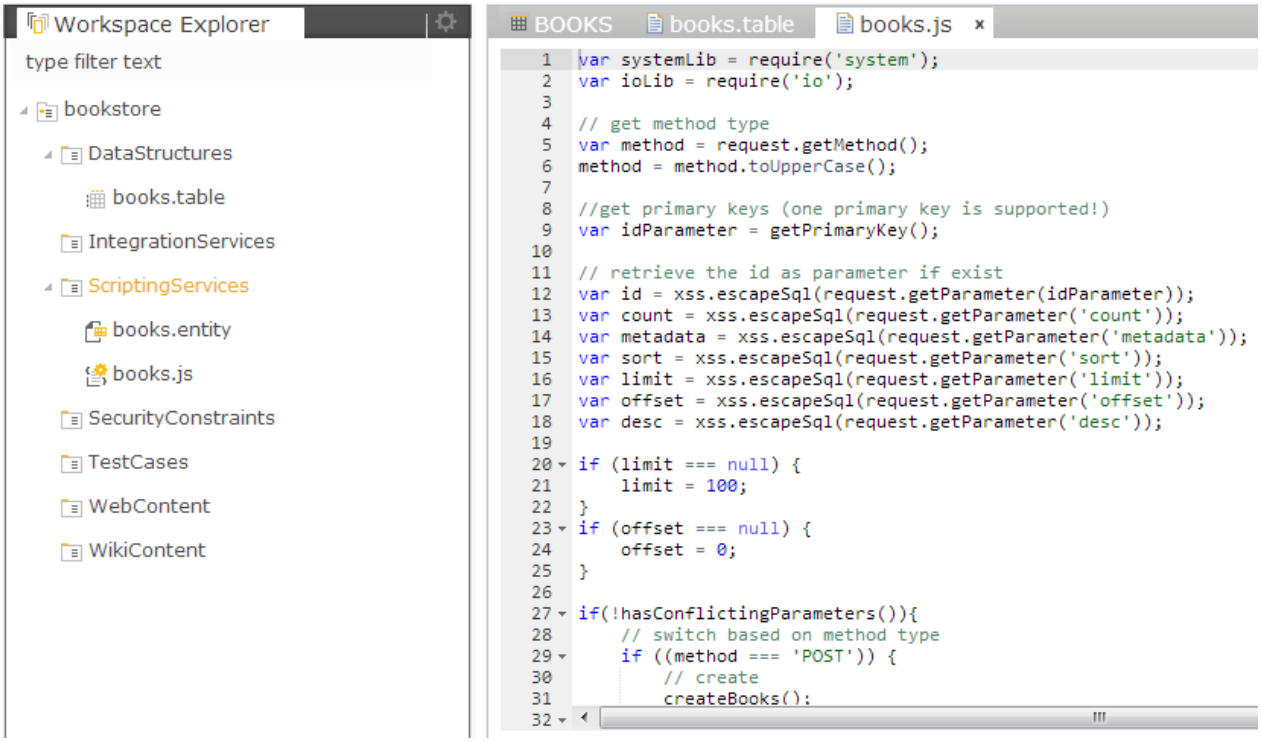
Once you have selected the "Entity Service" template and click "Next", the page which will be shown is specific one - lists all the available tables from the current database schema.



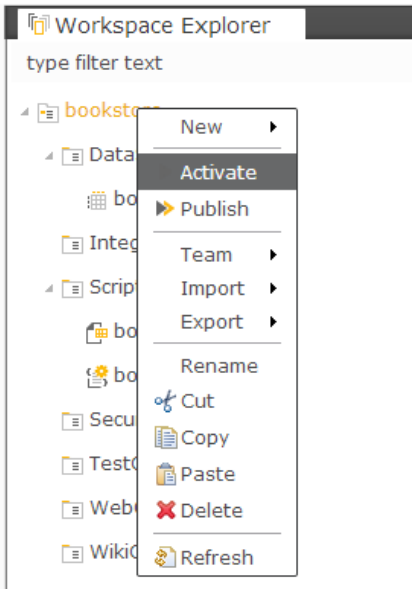
Give the name of your entity service on the next page and click finish



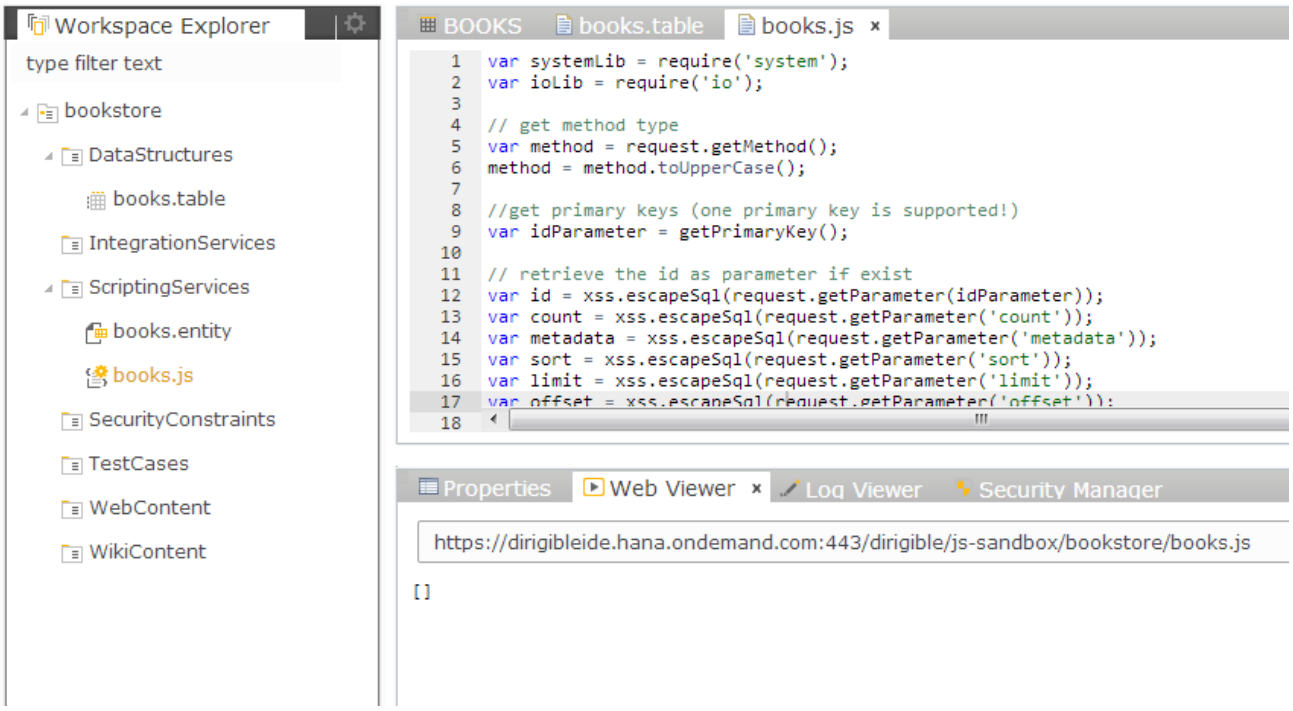
The generated service should be opened in the editors area.



Now we can use activation action from the project's pop-up menu to enable the service

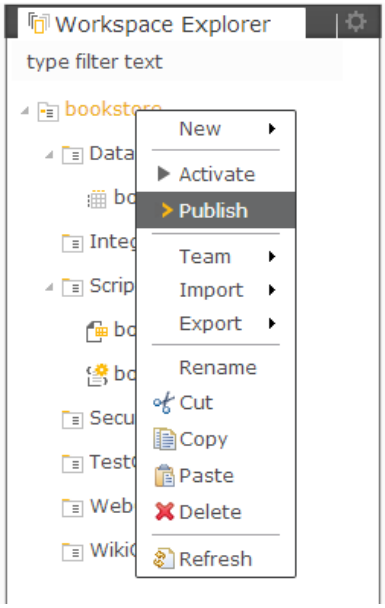


During the activation the artifact goes to the sandbox of the logged-in user. We can see the result of calling the service right away in the Preview (sould be opened by default in the Workspace Perspective), so find it (next to Properties view) and select it. Now go to the Workspace Explorer where the project is managed and select the service artifact (books.js). This will trigger the construction of the right URL of the service endpoint in the sandbox, hence you will see the result in the Preview.

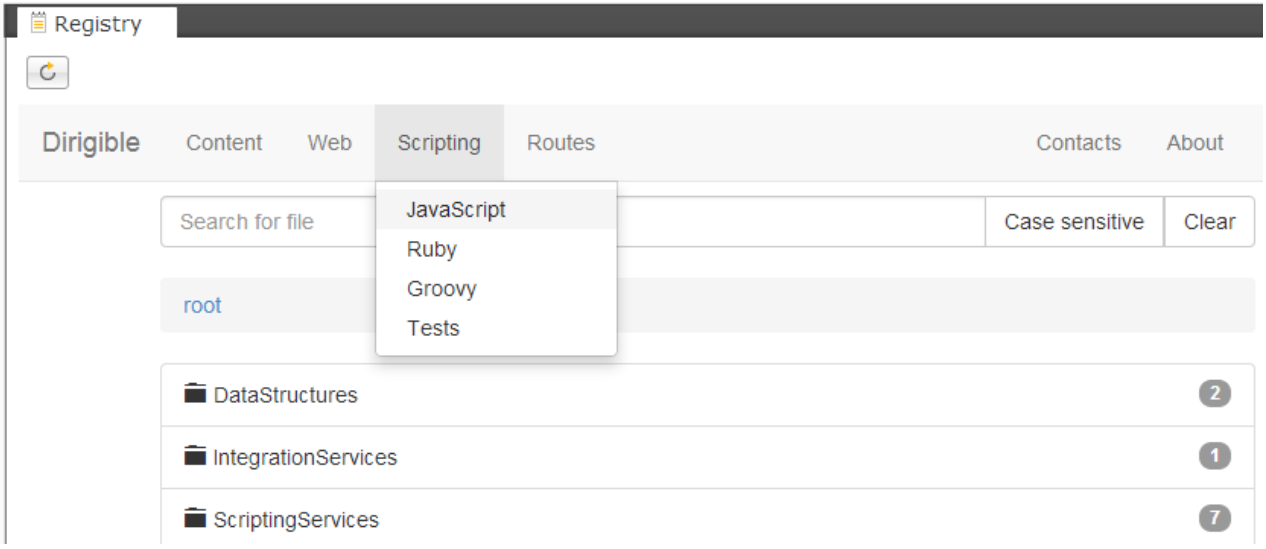


in this case just an empty JSON array.

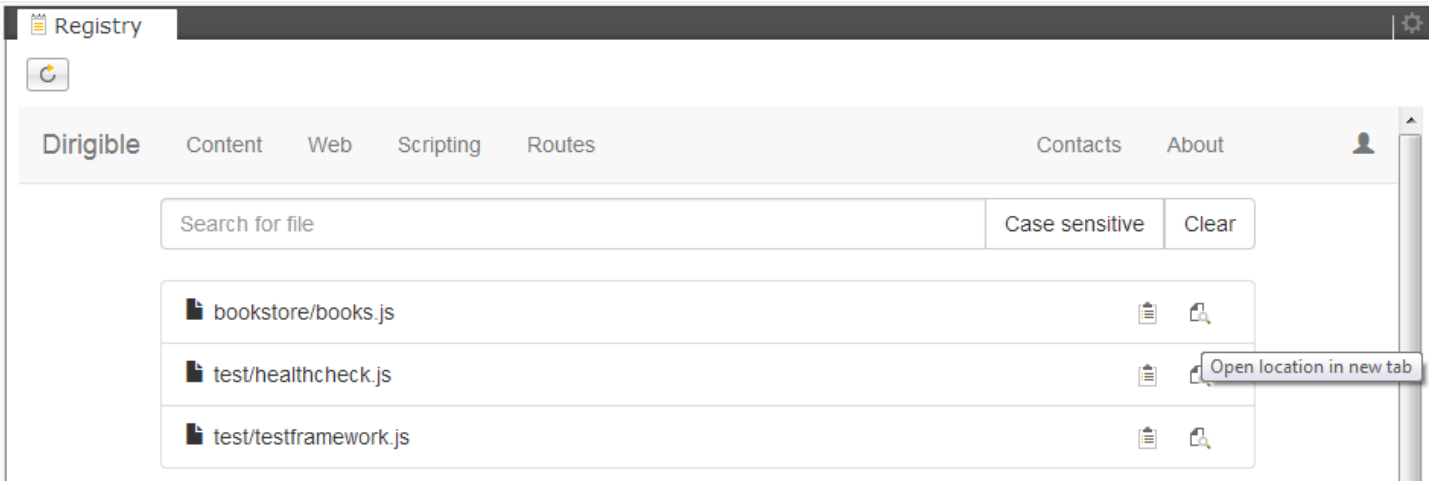
If you like the result in the sandbox you can publish the service, so that it become accessible by the other users.



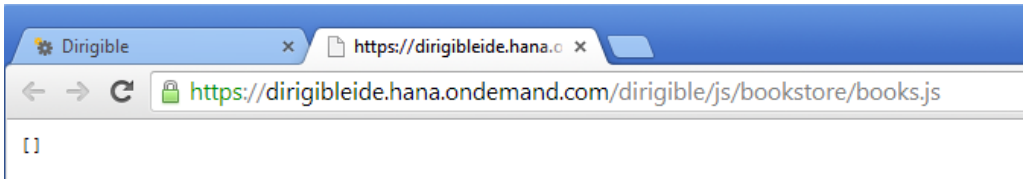
To find the URL you can go to the Registry Perspective. From the main menu go to Window->Show Pespective->Registry The Registry Perspective is representing a view to the enabled runtime content. From its menu choose Scripting->JavaScript to open the currently available server-side JavaScript service endpoints.



You can see the list of the available end-points, where you can find yours by naming convention .



The link to the service can be copied to the clipboard via the first image at the right side of the row or can be directly opened by clicking on the second image.



The naming convension for the service' endpoints URLs is as follows:

> *[protocol]://[host]:[port]/[dirigible's runtime application context]/[scripting container mapping]/[project]/[service path]*

e.g.

> *https ://dirigibleide.hana.ondemand.com/dirigible/js/bookstore/books.js*

The scripting containers mappings are:

- JavaScript
  - /js
  - /js-secured
- Ruby
  - /rb
  - /rb-secured
- Groovy
  - /groovy
  - /groovy-secured
- Test
  - /test

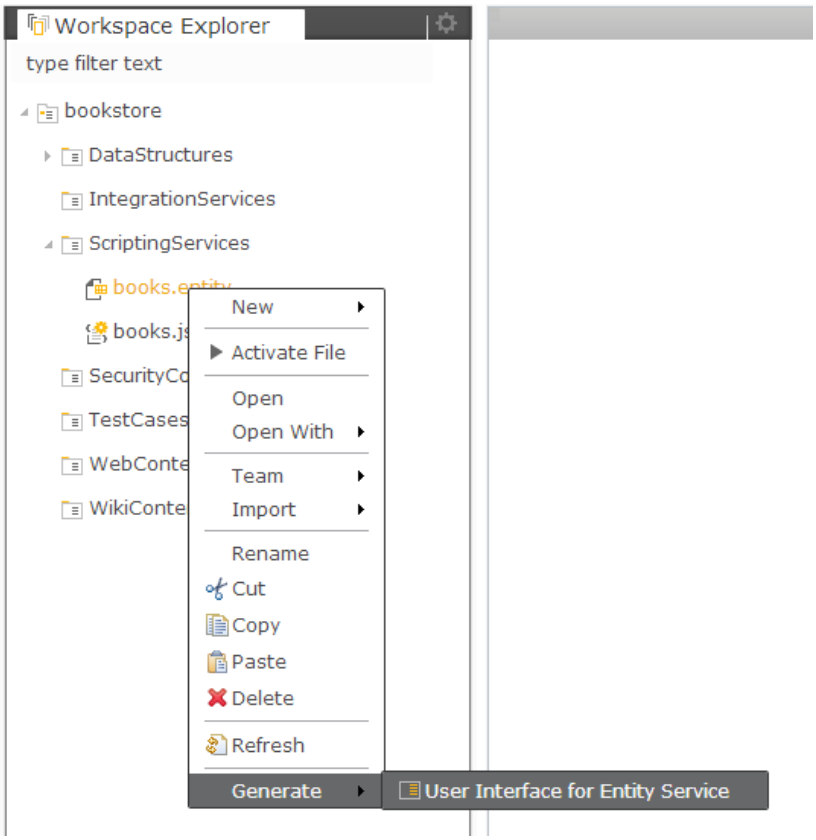
The suffix "-secured" above shows the access point for the secured end-points. More info can be found here.

There are some specific requirement for the table to be able to be exposed as entity service (e.g. primary key have to be defined, it should be a single column, etc.). Also the entity service itself support a bit more operations than the standard ones defined by HTTP. More about the entity services can be found here.

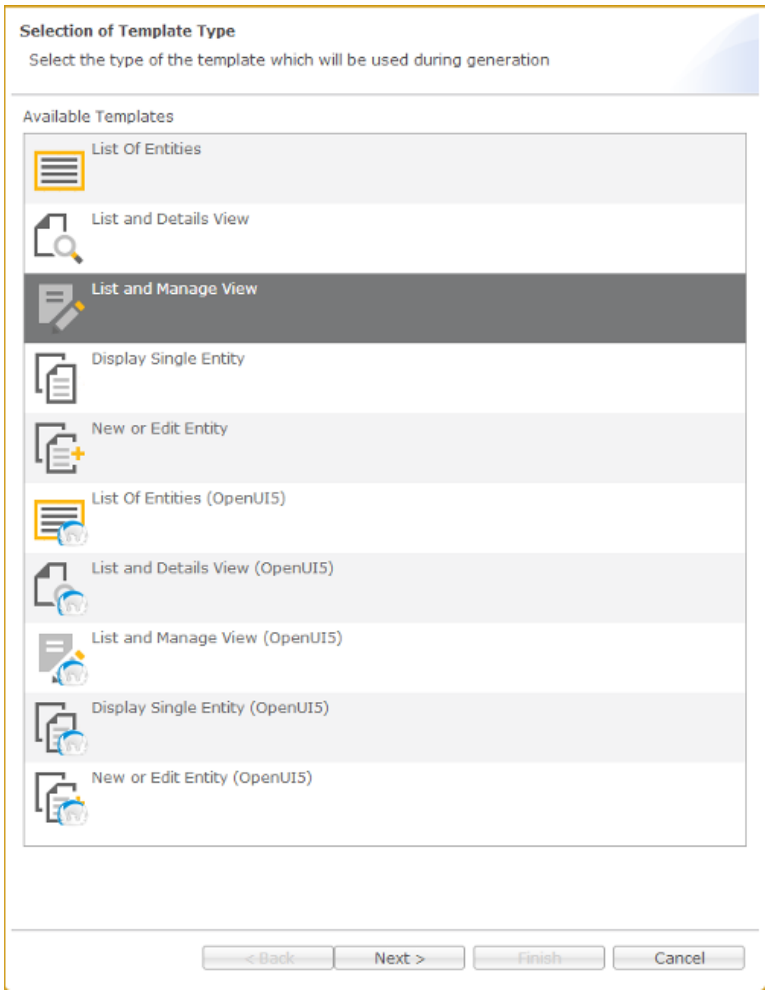
Entity User Interface

After the creation of the data model and the entity service, now we will going to generate an user interface for entity management (list, new, edit, delete...)

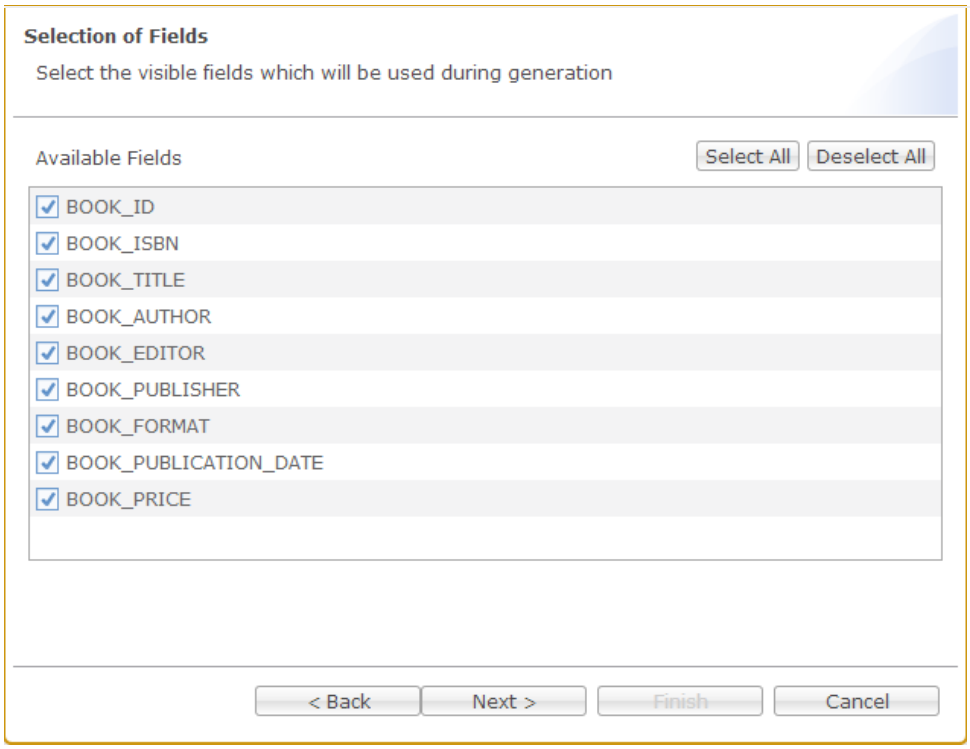
Select the *books.entity* and open the pop-up menu. Choose *Generate->User Interface for Entity Service*



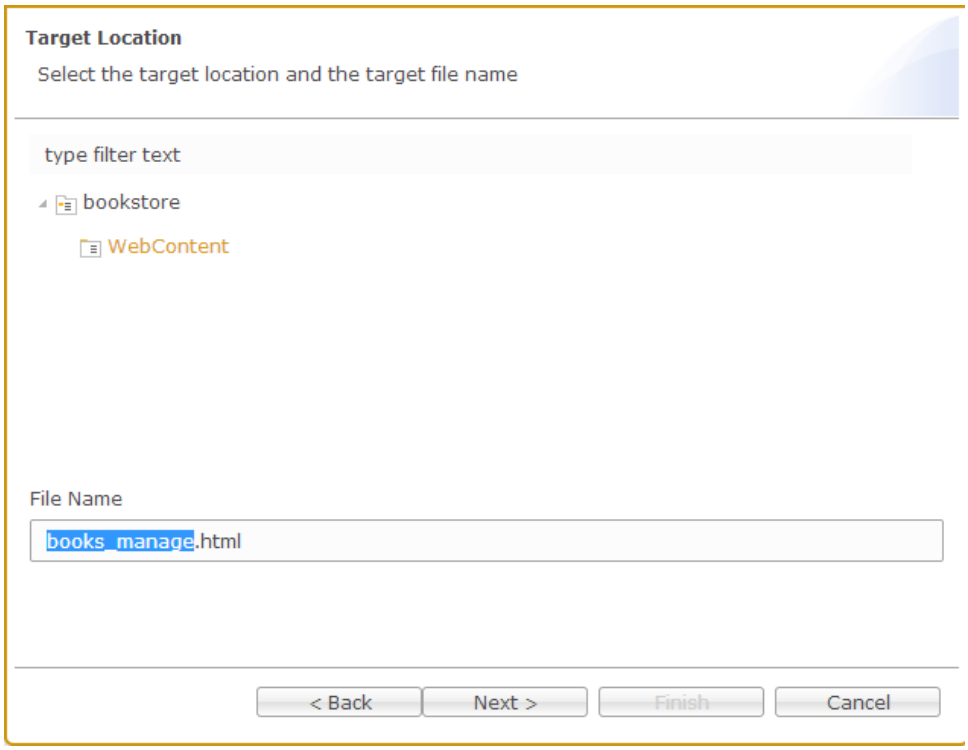
From the wizard select the template "List and Manage View"



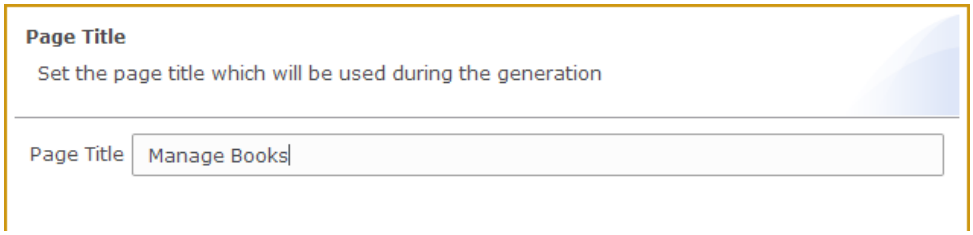
Click Next and select all the columns from the list. You can use "Select All" button



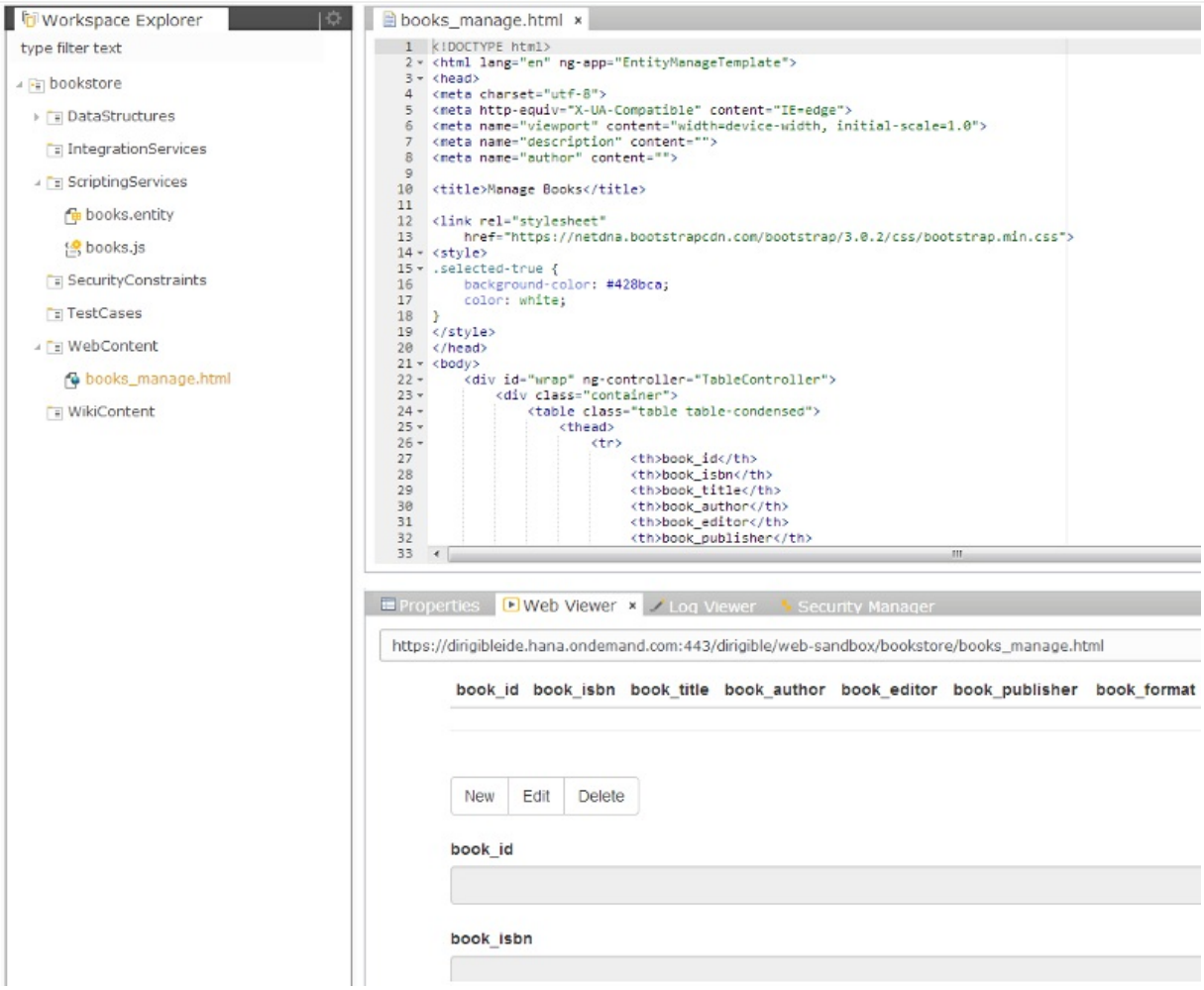
On the next page enter the name of the page *books_manage.html*



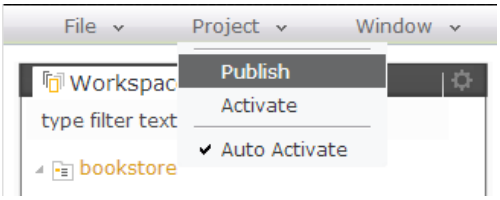
For the Title on the next page you can enter *Manage Books*



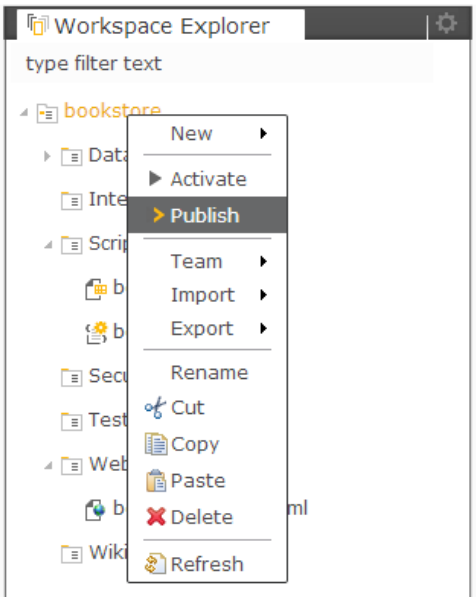
After clicking Finish button the generation is triggered. You can see the result under the WebContent folder When you select the file with active Preview you shall see the resulted running page.



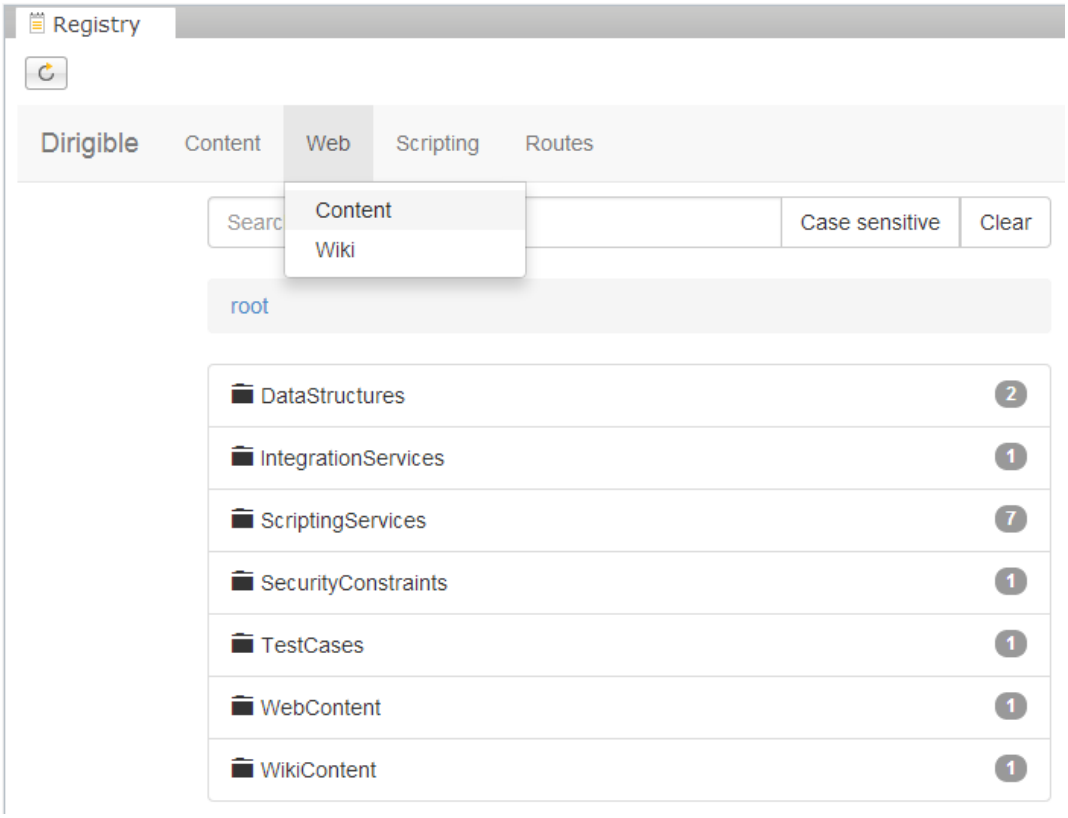
For the real test of the web page and the entity service you can Publish the project



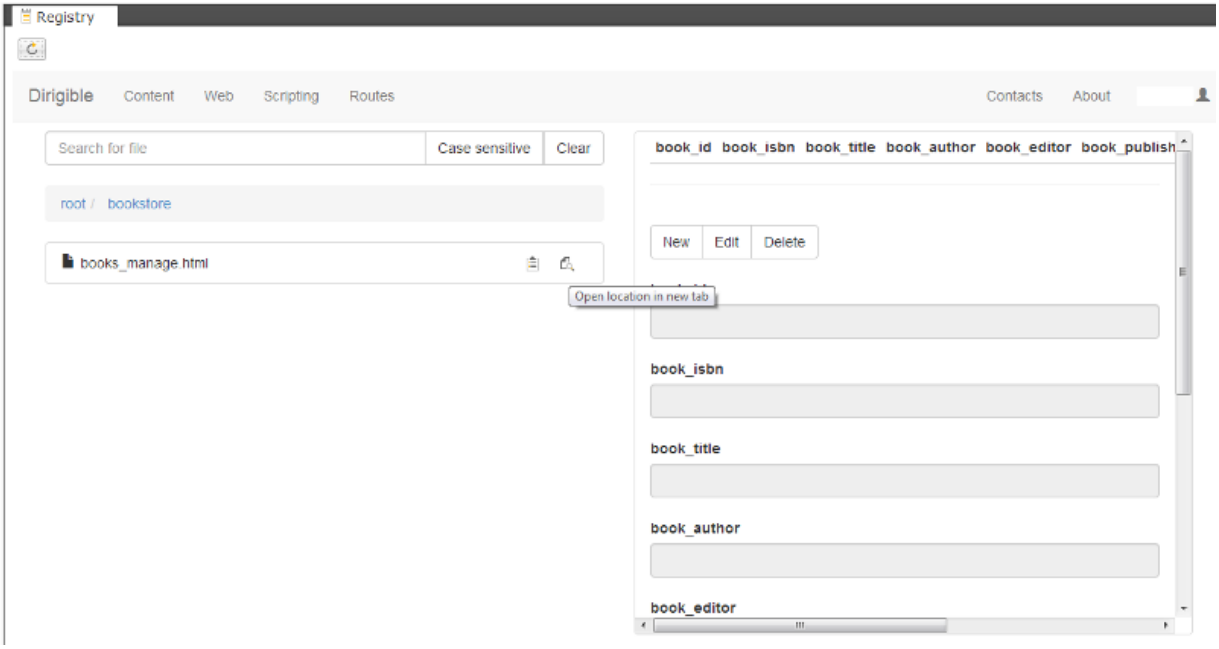
or



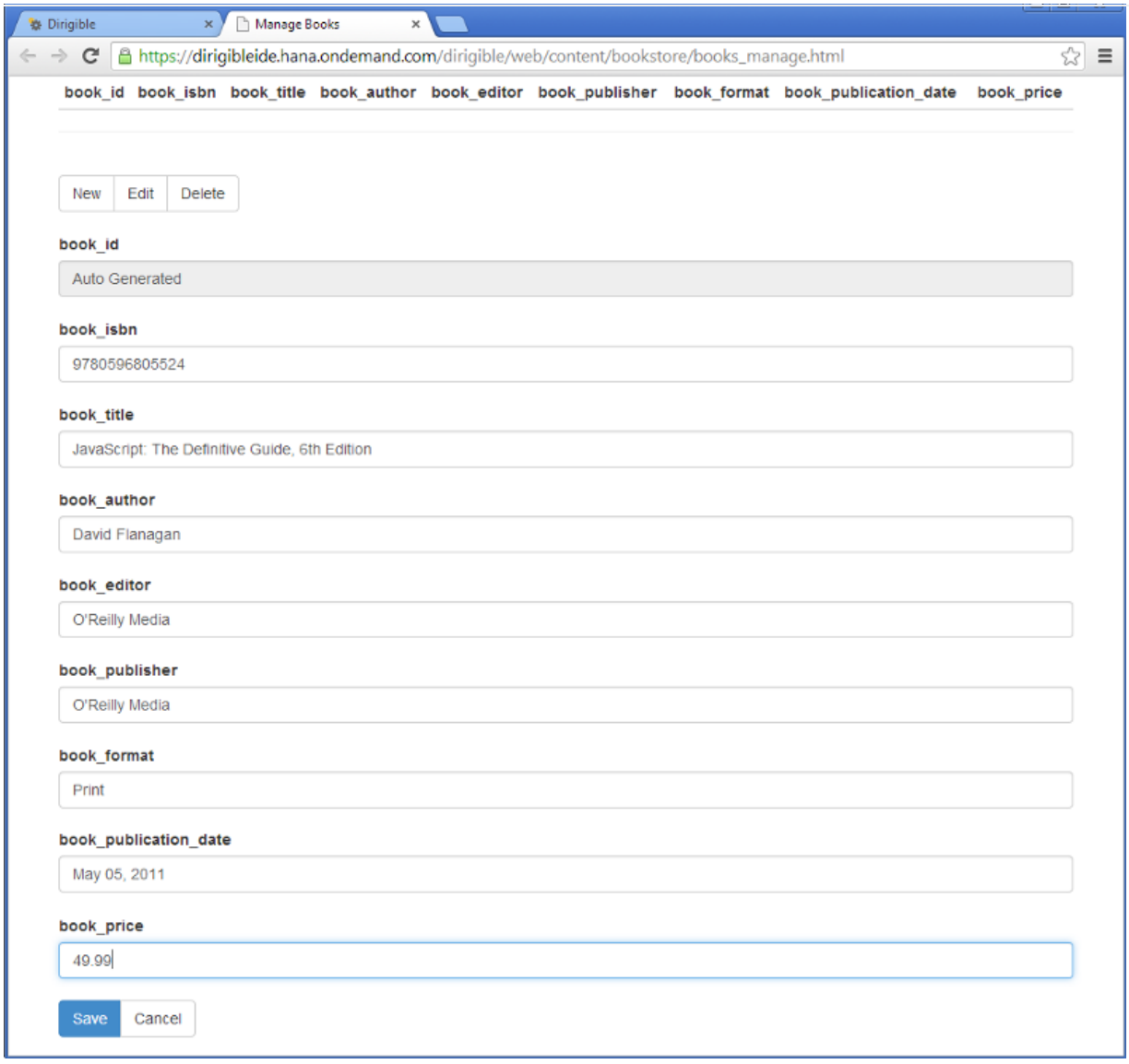
Now fo to the Registry perspective to find the link to the page, so that we can open it in an external browser. From the Registry embedded page menu choose Web->Content



Drill-down in the bookstore project folder and click on the page which is listed. To open the page in a new tab click on the icon on the right side



Click on "Edit" button and input the information about the first book you want to have in your store.



Click Save button and see the inserted record in the table above