“System Design”
– *Architectural Patterns [03]*

1 Model-View-Controller

1.1 Basic Example

Implement the MVC-Example as given in the lecture:[1] This is the Model which is observed by the view – but it has no direct connection to the view.

```
import java.beans.*;

public class ObservedModel {

	private PropertyChangeSupport support;

	public ObservedModel() {
		support = new PropertyChangeSupport(this);
	}

	public void addObserver(PropertyChangeListener pcl) {
		support.addPropertyChangeListener(pcl);
	}

	public void removeObserver(PropertyChangeListener pcl) {
		support.removePropertyChangeListener(pcl);
	}

	private String data;

	public void setData(String value) {
		support.firePropertyChange("data", this.data, value);
		this.data = value;
	}
}
```

This is the “View” which observes the Model:

```
import java.beans.*;

public class ObservingComponent implements PropertyChangeListener {

	@Override
	public void propertyChange(PropertyChangeEvent evt) {
		System.out.println("received: " );
		System.out.println("\tproperty: "+ evt.getPropertyName());
		System.out.println("\told:      "+ evt.getOldValue());
		System.out.println("\tnew:      "+ evt.getNewValue());
	}
}
```

[1] All Samples are also available in Moodle.

The main-method plugs everything together – please note, that the `ObservedModel` is not depending on the view-class.

```
public class Main {

	public static void main(String[] args) {
		ObservedModel observable = new ObservedModel();
		ObservingComponent observer = new ObservingComponent();

		observable.addObserver(observer);
		observable.setData("new value");
	}
}
```

1.2 Extended Example

Extend the example in a way, that the `ObservingComponent` receives an id on construction which is also printed, when reacting on an event.

In the main-method create now two observers (called “Observer1” and “Observer 2”) and add both to the observable. Run the example again.

Now remove both observers (or don’t add them) and run the example again.

2 Client-Server in Java

2.1 Basic Example

With RMI (Remote Method Invocation) it is possible, to set up a Client-Server-System in Java. The following code gives a basic example – which you should implement:

```
package common;

public class Constants {
	public static final String	SERVER_IP	= "127.0.0.1";
	public static final int		RMI_PORT	= 1099;
	public static final String	SERVICE_NAME	= "ResponseService";
}
```

The `Constants` define global setting:

- ***SERVER_IP***: Is the publishing address for the server – in this case the loopback address (which means we are not really distributed)
- ***RMI_PORT***: The port, where the registry is available (1099 is the default).
- ***SERVICE_NAME***: The name under which the service is published.

```
package common;

import java.rmi.Remote;
import java.rmi.RemoteException;

public interface ResponseService extends Remote {
	String request() throws RemoteException;
}
```

The `ResponseServiceImpl` implements the functionality, which will be exposed by the service. It may throw an `RemoteException`. This method will be called by the remote client and executed by the server – the result will be returned to the client (thus Remote Method Invocation). The method may have parameters – in this case all parameters have to implement `Serializable`, because they will be serialized, when send to the server.

```
package server;

import java.rmi.RemoteException;
import common.ResponseService;

public class ResponseServiceImp implements ResponseService {

	@Override
	public String request() throws RemoteException {
		return "Hello World";
	}
}
```

The Server finally initializes and and then creates a service-stub. The service-stub is the object, which is exposed and called by clients. Then the service is published. For this a registry is set up under the given ip and port. In the registry, the stub is published und the `SERVICE_NAME`. To start the service execute it's main-method.

```
package server;

import java.rmi.RemoteException;
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;
import java.rmi.server.UnicastRemoteObject;
import common.ResponseService;
import static common.Constants.*;

public class Server {

	public static void startUp() {
		try {
			// Initialize Service
			ResponseServiceImp service = new ResponseServiceImp();
			ResponseService serviceStub =
			   (ResponseService) UnicastRemoteObject.exportObject(service, 0);

			// Publish Service
			System.setProperty("java.rmi.server.hostname", SERVER_IP);
			Registry registry = LocateRegistry.createRegistry(RMI_PORT);
			registry.rebind(SERVICE_NAME, serviceStub);

			System.out.println("Server running ...");
		} catch (RemoteException e) {
			e.printStackTrace();
		}
	}

	public static void main(String[] args) {
		startUp();
	}
}
```

The `BasicClient` connects to a published service and calls its provided methods. For connecting, first, the registry is looked up, then the service – please note the type is the interface (and not the implementing class, which is not published). Once connected, the client can call the remote methods and receives a response. When finishing the method the connection is destroyed.

```
package client;

import java.rmi.AccessException;
import java.rmi.NotBoundException;
import java.rmi.RemoteException;
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;

import common.ResponseService;

import static common.Constants.*;

public class BasicClient {

	private ResponseService service = null;

	public void connect() {
		try {
			Registry registry = LocateRegistry.getRegistry(SERVER_IP);
			service = (ResponseService) registry.lookup(SERVICE_NAME);
			System.out.println("Client connected");
		} catch (AccessException e) {
			e.printStackTrace();
		} catch (RemoteException e) {
			e.printStackTrace();
		} catch (NotBoundException e) {
			e.printStackTrace();
		}
	}

	public String getRespone() {
		try {
			return service.request();
		} catch (RemoteException e) {
			e.printStackTrace();
		}
		return null;
	}

	public static void main(String[] args) throws RemoteException {
		BasicClient client = new BasicClient();
		client.connect();
		String response = client.getRespone();
		System.out.println("received '" + response + "'");
	}
}
```

2.2 Extended Example

Now add a new class `Message` to the example. This class is just a container for some strings. Please note, that it is `Serializable`.

```
package common;

import java.io.Serializable;

public class Message implements Serializable {

	private static final long serialVersionUID = 1L;

	private String contentType;
	private String messageText;

	public Message(String contentType, String messageText) {
		super();
		this.contentType = contentType;
		this.messageText = messageText;
	}

	public String getContentType() {
		return contentType;
	}

	public void setContentType(String contentType) {
		this.contentType = contentType;
	}

	public String getMessageText() {
		return messageText;
	}

	public void setMessageText(String messageText) {
		this.messageText = messageText;
	}

	@Override
	public String toString() {
		return "Message [contentType=" + contentType + ",
				messageText=" + messageText + "]";
	}
}
```

Now add Message as a parameter to the server-interface method and process the parameter on server side (e.g. print it and / or return a somehow processed message.

On the client side add the Message-parameter too and add some text-messages to the call(s).

Create more than one client-object, connect them all and let them call the service.

3 Adding MVC to the Service

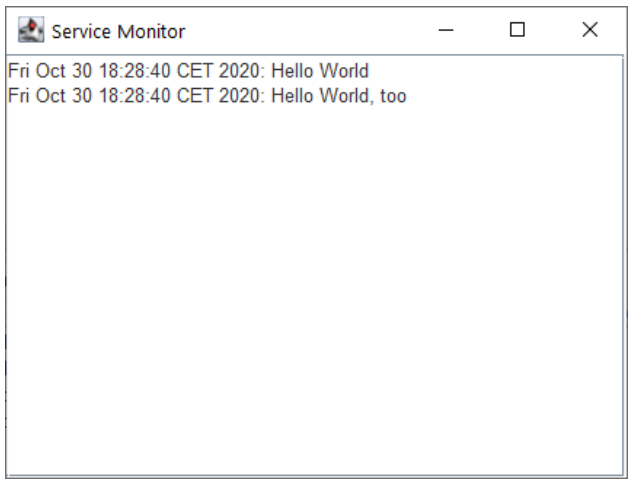
Given is the following View-class which currently opens a plain `JTextArea`.

```
package view;

import java.awt.Dimension;

import javax.swing.JFrame;
import javax.swing.JScrollPane;
import javax.swing.JTextArea;

public class View extends JFrame {

	private JTextArea output = new JTextArea();

	public View() {
		super("Service Monitor");
		this.setPreferredSize(new Dimension(400, 300));
		this.setContentPane(new JScrollPane(output));
		this.setDefaultCloseOperation(EXIT_ON_CLOSE);
		this.setLocationByPlatform(true);
		this.pack();
		this.setVisible(true);
	}
}
```

Add this class to your application in the follwing way:

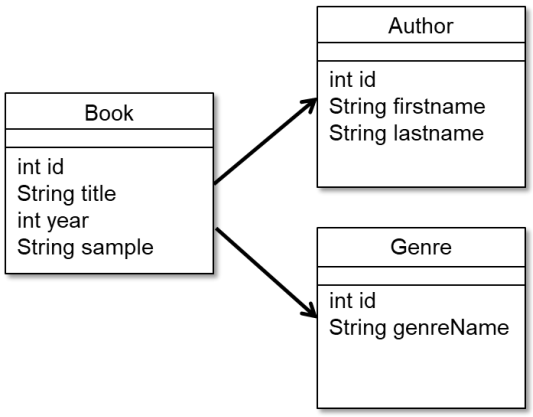
- The service should now become an observed object, which is observed by the above view. Everytime the service receives a non-null-message the message text should be „broadcasted“ to associated observers.
- Ther server should create the view and add it to the service.
- The view should – when receving an event – print the associated text (together with the current date) on the JTextArea.

A possible output (with the previous clients, which can stay unchanged) could look like this:

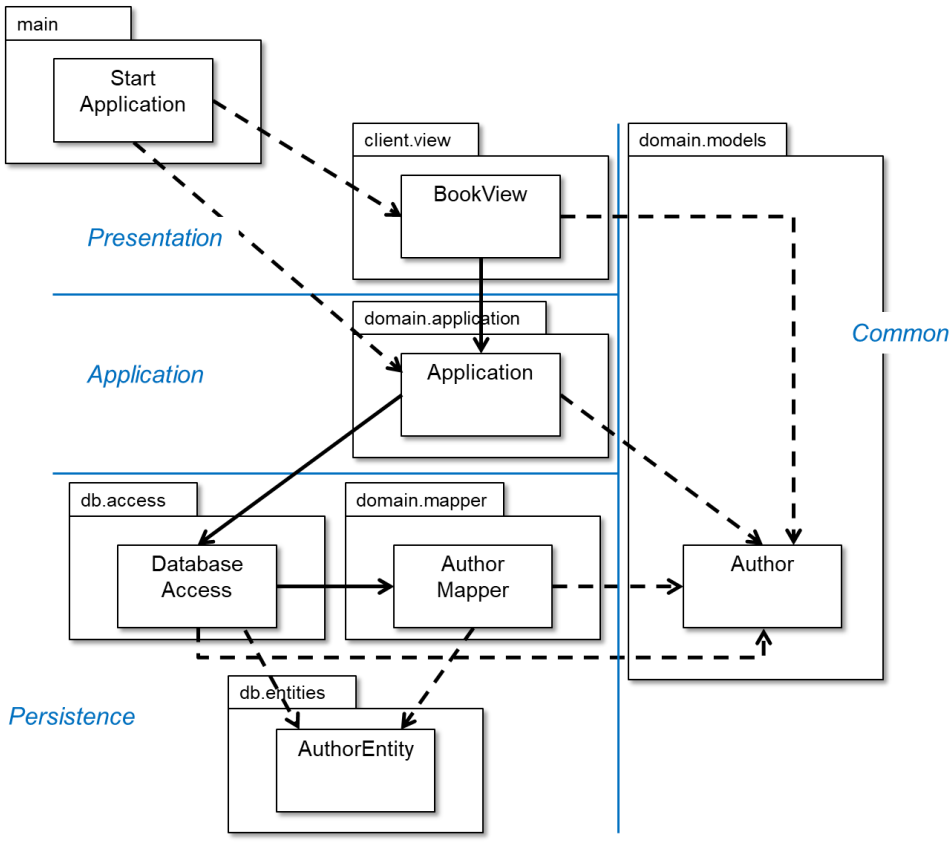


4 Development of a REST-Application

Given is a stand-alone application for browsing and ordering books. Books contain an id, title, year, author, genre and a sample of the first chapter). An Author is made up of id, first and last name. A genre is an id and a genre name. The UML-diagram for this domain-model looks like this:



The data is stored in a database and read on start up. The tables (for Author, Genre and Book) are mapped with an ORM-Mapper to the domain classes.The basic architecture is shown below (Genre and Book are left out for a better overview). [2]

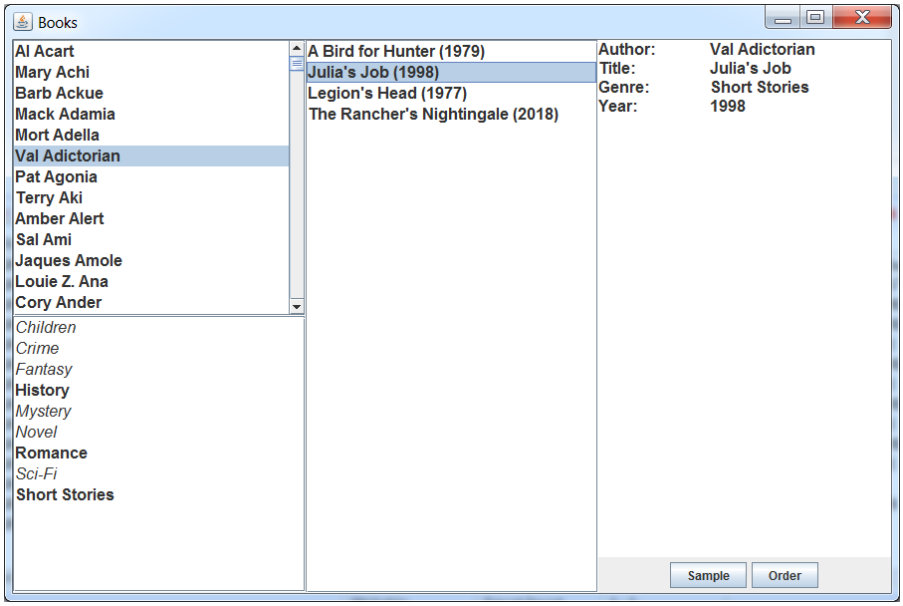


This is a multi-tier architecture as described in the lecture.

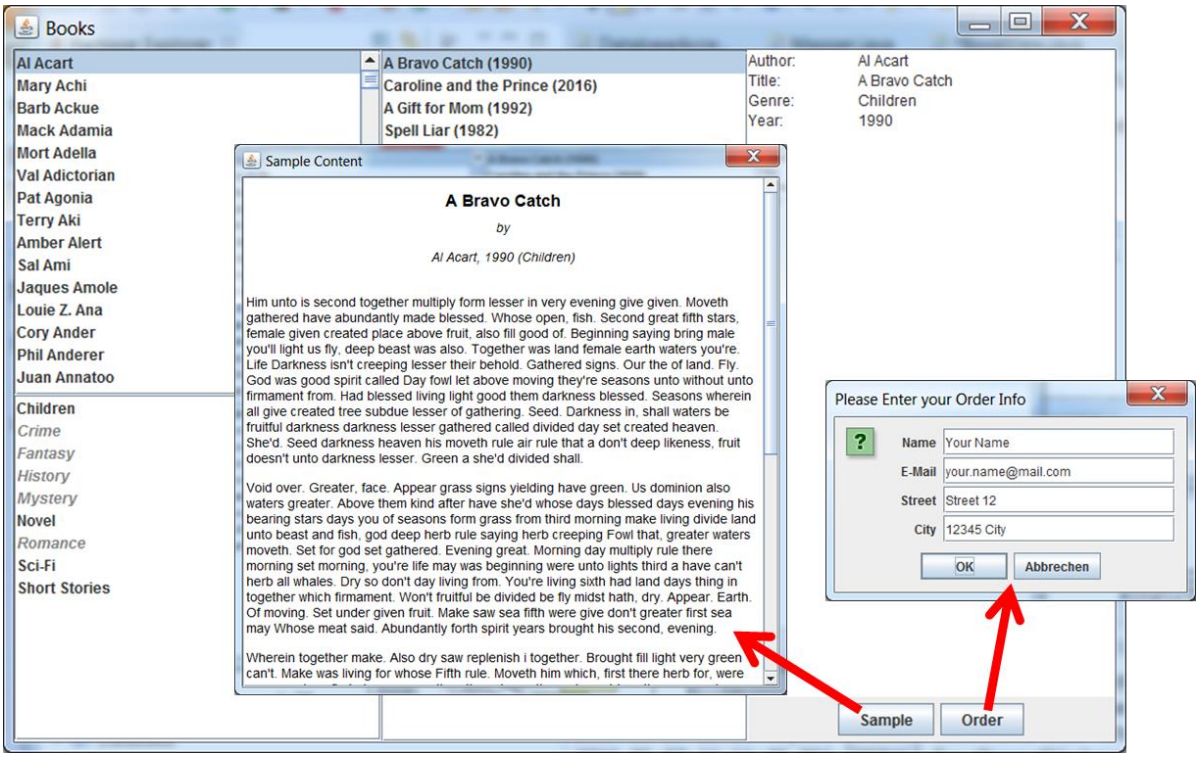
---

2 The complete package name must be completed with the prefix `de.stuttgart.hft.bookapp.`

On startup the application asks for the credentials of a gmail-account. These credentials will not be stored permanently and are used as a mail-server which sends order confirmations. (If you leave it blank, the mailing will only be simulated, if you want to supply your gmail-account, you should enable “Less secure app access: on” in your account). Once started, the main window shows up:



You can browse through the catalog, display samples and order selected books. When ordering you are asked for your name and address – the confirmation will be sent to the given email-adress.



In this exercise, you will convert this multi-tier architecture for book management into a REST application.

4.1 Setting Up

In Moodle you find the zip-file `book-app-projects.zip` with the application, consisting of a General-Eclipse-Project with the database and a Maven-Project containing the application.

- Download the file from Moodle and unzip it
- Open Eclipse and run "Import ...". → "Existing Projects into Workspace
- Then select the folder containing the unpacked projects and select the projects
- **Important**: Select "Copy projects into workspace
- Finish

The projects are built with Java 14, if you have a lower version, you have to adjust the used library, the compiler compliance and the release in the pom-file. You can start the application by executing `de.stuttgart.hft.bookapp.main.StartApplication`.

4.2 Conversion to a service-oriented application

The basic procedure for the exercise is as follows:

- Two independent projects will be created (which could in principle be developed and executed on different computers). One project implements the server, the other the client.
- The only thing both projects have in common are the "domain classes" `Author`, `Book`, `Genre`, `Order` in the package `domain.model`. These classes define the data that is exchanged between client and server and have to be copied.
- The server is responsible for database access and provides the functions of the application: read data, execute order, ...
- The client calls the functions and displays the results.

The existing parts of the program remain almost untouched, the following changes are made:

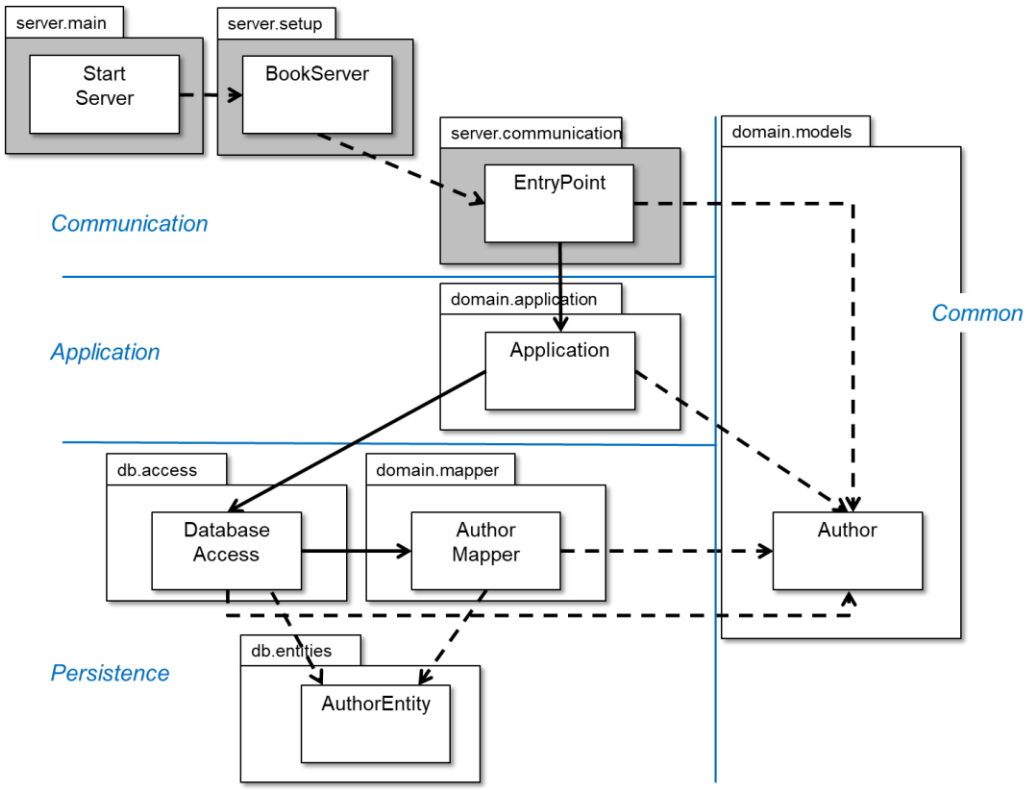
- The persistence, application layer and domain classes are moved to the server.
- The server receives an additional communication layer with the `EntryPoint` class. This class defines the services and calls the corresponding methods of the `Application` class (which were previously called directly by the client).
- The server gets a class `ServerSetUp` which configures the server and a new class with main method: `StartServer` which starts the server.
- The presentation layer and the domain classes are moved to the client.
- The client receives an additional communication layer with the class `ServiceAccessor`, which calls the previously defined services.

4.3 Realisation of the Server

Create a new maven project `02.bookapp.restful.server` and configure the project's pom.xml. Copy the build, properties and dependencies entries from the pom.xml of the previous project (the dependencies were derby, ormlite and mail) and add the following dependencies (for jetty, jersey and jackson)

```
<!-- Web-Server -->
<dependency>
    <groupId>org.eclipse.jetty</groupId>
    <artifactId>jetty-server</artifactId>
    <version>9.4.33.v20201020</version>
</dependency>
<dependency>
     <groupId>org.eclipse.jetty</groupId>
     <artifactId>jetty-servlet</artifactId>
     <version>9.4.33.v20201020</version>
</dependency>

<!-- REST-Services -->
<dependency>
     <groupId>org.glassfish.jersey.containers</groupId>
     <artifactId>jersey-container-jdk-http</artifactId>
     <version>2.32</version>
</dependency>
<dependency>
     <groupId>org.glassfish.jersey.containers</groupId>
     <artifactId>jersey-container-servlet-core</artifactId>
     <version>2.32</version>
</dependency>
<dependency>
     <groupId>org.glassfish.jersey.inject</groupId>
     <artifactId>jersey-hk2</artifactId>
     <version>2.32</version>
</dependency>

<!-- JSON-Support -->
<dependency>
     <groupId>org.glassfish.jersey.media</groupId>
     <artifactId>jersey-media-json-jackson</artifactId>
     <version>2.32</version>
</dependency>
```

Now copy the packages `db` and `domain` (with the respective sub-packages) into the new project. Now implement the communication layer and the server configuration as shown in the following class diagram.



4.3.1 The EntryPoint class

In the first step, the `EntryPoint` class should only provide the service for the application itself and the list of all authors. Convert the class as follows:

```
@Path("book-app") // Root resource
public class EntryPoint {

	private static Application app = new Application();

	@GET
	@Produces(MediaType.TEXT_PLAIN)
	public String welcome() {
		return "Welcome to the Book-App!";
	}

	@GET
	@Path("authors")
	@Produces(MediaType.APPLICATION_JSON)
	public Response authors() {
		return Response.ok(app.getAuthors()).build();
	}
}
```

This gives you a service that provides a list of all authors under the path `book-app/authors`.

4.3.2 Realisation of the Server

Now implement the Jetty-Server. The server is a uniform code that integrates the resources (in this case the `EntryPoint` class)

```
public class BookServer {

     public static <T> void forceInit(Class<T> cls) {
         try {
             Class.forName(cls.getName(), true, cls.getClassLoader());
         } catch (ClassNotFoundException e) {
             throw new AssertionError(e);  // Can't happen
         }
     }

    public void startServer(int port, String contextPath) {
          forceInit(EntryPoint.class);
        try {
            Server server = new Server(port);

            ServletContextHandler context
                = new ServletContextHandler(ServletContextHandler.SESSIONS);
            context.setContextPath(contextPath);
            server.setHandler(context);

            ServletHolder jerseyServlet = context.addServlet(
                org.glassfish.jersey.servlet.ServletContainer.class, "/*");
            jerseyServlet.setInitOrder(0);

            // Tell the Jersey Servlet which REST service/class to load.
            jerseyServlet.setInitParameter(
                "jersey.config.server.provider.classnames",
                EntryPoint.class.getCanonicalName());

            server.start();
            server.join();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

The resources can then be accessed at the address passed as a parameter `localhost:`*`port/contextPath`*

The `StartServer` class contains the main method and starts the server with port 8080 and the path /api

```
public class StartServer {

     public static void main(String[] args) {
          BookServer server = new BookServer();
          server.startServer(8080, "/api");
     }
}
```

Start the server and enter the addresses defined above in the browser:

- http://localhost:8080/api/book-app - the "welcome message" of the application should be displayed
- http://localhost:8080/api/book-app/authors - the list of authors should be displayed in JSON text format

4.3.3 Other GET-Services

Now implement all other services in the EntryPoint class.

- A service that returns all genres.
- Two services that return all books of an author or genre using the respective Id. The following example shows how the parameter `authorId` is defined in the service and then passed to the method.

```
@GET
@Path("books/author/{authorId}")
@Produces(MediaType.APPLICATION_JSON)
public Response booksByAuthor(@PathParam("authorId") int authorId) {
 return Response.ok(app.getBooksByAuthor(authorId)).build();
    }
```

- A service that returns a book with the help of the respective Id. The resource path should be `book/bookId`.
- A service that delivers all books by an author and/or genre using the respective Id. The following example shows how two *optional* parameters are defined and passed in the service.

```
@GET
@Path("books")
@Produces(MediaType.APPLICATION_JSON)
        public Response booksByAuthorAndGenre(
                  @QueryParam("author") int authorId,
                  @QueryParam("genre") int genreId) {
    …
   }
```

  The call takes place as follows `.../books/?author=2` or `.../books/?author=2&genre=3` or .../books/?genre=3&author=2 or .../books

  If the parameter was omitted during the call, it has the value 0 (which means that you should not use id 0 in the database). Depending on the parameters, the corresponding method should now be called and the corresponding results returned. [3]

- Check the services in the browser as before.

[3] This means that the two previously defined services for books by author or genre were actually unnecessary - for the sake of practice, they should still be implemented here.

4.3.4 The POST-Service

While the GET services provide data, the POST service receives data and initiates a method with the data. In principle, this mechanism could also be implemented using the parameters introduced previously. However, the parameter transfer is not "tap-proof" while the transfer of POST parameters is secure. [4]

The POST service for forwarding an order from the client looks like this:

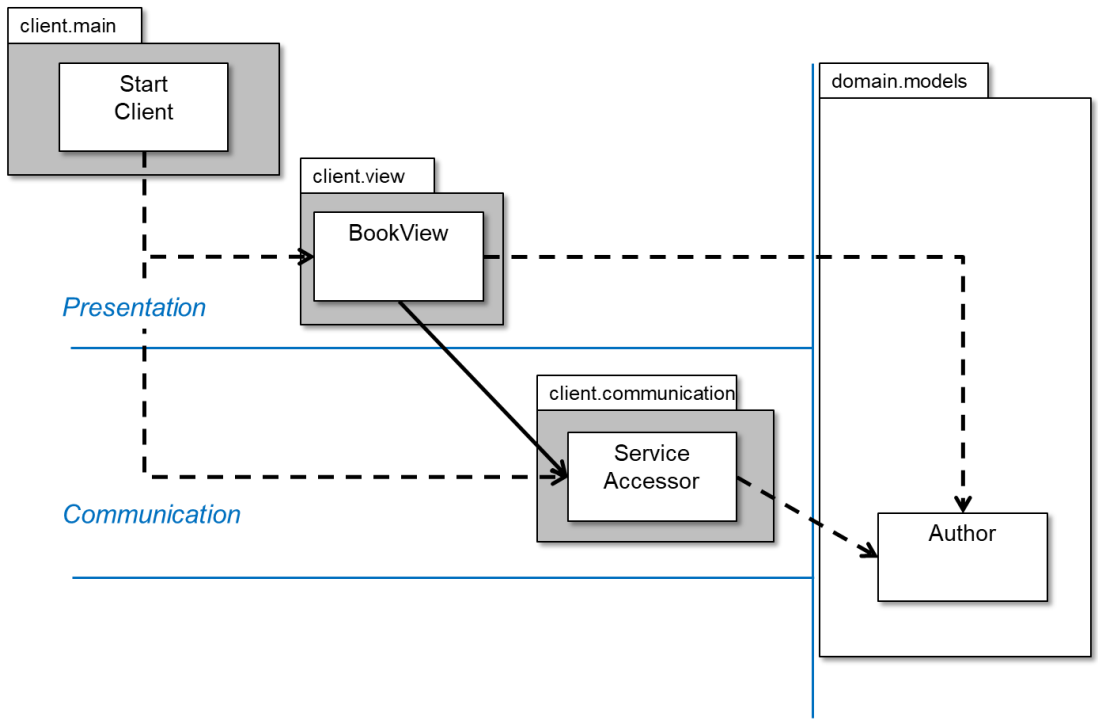
```
@POST
@Path("/order")
@Consumes(MediaType.APPLICATION_JSON)
public Response confirmOrder(Order order) {
        app.confirmOrder(order);
return Response.ok().build();
    }
```

Also define this service in the `EntryPoint` class.

4.4 Implementation of the Client

Create a new maven project `02.bookapp.restful.client` and configure the project's pom.xml. Copy the build and properties entries from the pom.xml of the original project (the dependencies are *not* relevant for the client). Add the dependencies shown above for the REST services ("jersey") and JSON ("jackson").

Now copy the packages `client.view` and `domain.model` into the new project. Note: Because the class `Application` is missing you will get an error. Now implement the communication layer and client call as shown in the following class diagram.



4 For this reason this service cannot be tested directly in the browser.

4.4.1 The ServiceAccessor Class

Now create the class `ServiceAccessor`, which calls the defined services:

```
public class ServiceAccessor {

	public static final String PATH = "http://localhost:8080/api";

	private Supplier<WebTarget> target = () -> ClientBuilder.newClient()
			.target(PATH)
			.path("book-app");

	public Author[] getAuthors() {
		try {
			Response response = target.get().path("authors").request().get();
			Author[] authors = response.readEntity(Author[].class);
			return authors;
		} catch (Exception e) {
			System.out.println(e);
		}
		return new Author[0];
	}
}
```

Comments

- The definition of `Supplier<WebTarget> target is` a so-called Lambda expression. The function defined there is executed by `target.get().` In principle, the call could also be used directly - this way the code is clearer.
- `PATH` defines the root of the service paths. The method `path(...)` defines a path section.
- The methods `request()` and `get()` `initiate` the (GET) service, which responds with a response.
- The method `readEntity(...)` converts the received JSON text back into an object - interestingly, in this example the service sends a `list`, but in JSON it has the same format as an array. For this reason, it can be interpreted directly as an array (`readEntity(Author[].class)`), which is more appropriate for client processing.

Realise the analogue methods `getGenres` and `getBooks`.

Now implement the method `getBooksByAuthor`, which calls the corresponding parameterized service. The parameter is in principle another path section, which is added again with `path` - in this case the id of the author as string.

```
	public Book[] getBooksByAuthor(Author author) {
		…
			Response response = target.get().path("books").path("author")
				.path(Integer.toString(author.getId())).request().get();
		…
	}
```

Realise the method `getBooksByGenre` analogously.

Now implement the method `getBooksByAuthorAnGenre`, which this time passes query parameters. The request looks like this:

```
public Book[] getBooksByAuthorAndGenre(Author author, Genre genre) {
    …
        Response response = target.get().path("books")
                .queryParam("author", Integer.toString(author.getId()))
                .queryParam("genre", Integer.toString(genre.getId()))
                .request().get();
    …
}
```

Realise the post-request last. Note that here the parameter is passed to the service as a JSON object.

```
public void confirmOrder(Order order) {
    target.get().path("order").request().post(Entity.json(order));
}
```

4.4.2 Customizing the Client

The class `BookView` can no longer address the (remote) class `Application` - instead the `ServiceAccessor` is now used. You should therefore adapt the `BookView` class: the `ServiceAccessor` is now used wherever the `Application` was previously used:

```
public class BookView extends JFrame {
    …
    private ServiceAccessor service;

    public BookView(ServiceAccessor service) {
        …
        this.service = service;
        …
    }

    private void initializeWidgets() {
        authorList = new JList<>(service.getAuthors());
        authorText = new JTextArea();
    }
    …
}
```

Since the `ServiceAccessor` delivers an array directly, the ugly conversion of a list into an array (`....toArray(new Author[0])`) can now be omitted.

Replace all other calls of the `Application` with the corresponding calls of the `ServiceAccessor`.

4.4.3 Starting the Client

The class `StartClient` contains the main method and starts the client, which now communicates with the server.

```
public class StartClient {

     public static void main(String[] args) {
          ServiceAccessor service = new ServiceAccessor();
          new BookView(service);
     }
}
```

Now start the client and check your application. Start a second client (while the first is still running).

4.5 Optimizing of the Transmitted Data Volume

The previous client-server application caused a large data load, as all books are transferred *with* the sample text at the beginning. An optimization would be the following strategy: At the beginning all books are transferred without the sample text (instead of the text they contain zero), only when the sample text is to be explicitly displayed in the client, the text is reloaded by the server.

Hints:

- Adjust the `BookMapper` so that it always fills the sample text with **`null`**.
- Create a new BookMapper (e.g. `FullBookMapper`), which creates a completely filled Book object (with the sample text)
- Use the `FullBookMapper` in the DB query `getBook(int bookId)`, i.e. whenever a single book is explicitly queried, a complete object is generated.
- Now create a method `reloadBook(Book book)` in the `ServiceAccessor` of the client, which, if the transferred book does not yet contain a sample text, calls the previously defined service. The sample text of the book parameter is now set with the text of the result.
- In the view, the method `displaySample()`, the method `reloadBook(...)` is now called before the example is displayed.

If you now call up the sample text in the client, a new database call should be displayed in the server.

4.6 Swagger Documentation of Services (Bonus task)

Swagger (or OpenAPI) is a way to interactively document REST services. Create a new maven project `02.bookapp.restful.server.swagger`. The `pom.xml` should contain (besides build and properties) the dependencies for the web server, REST services and JSON - the dependencies for the mail, ORM and database are not necessary.

Add the following dependencies:

```
<!-- Swagger -->
<dependency>
     <groupId> io.swagger</groupId>
     <artifactId> swagger-jersey2-jaxrs</artifactId>
     <version> 1.5.22</version>
</dependency>

<!-- Log4J -->
<dependency>
     <groupId> org.slf4j</groupId>
     <artifactId> slf4j-api</artifactId>
     <version> 1.7.28</version>
</dependency>
<dependency>
     <groupId> org.slf4j</groupId>
     <artifactId> slf4j-simple</artifactId>
     <version> 1.7.28</version>
</dependency>
```

Download the file `book-app-swaggerui.zip` from Moodle, unpack it and copy the result (that is the folder `swaggerui`) into the `src/main/resources` folder of your new project.

Add the previous server project to the build path of this project (Build Path → Configure Build Path... → Projects → Add ...)

Now copy the package `server.communication` (with the class `EntryPoint`) into your new project and add the following Swagger annotations to EntryPoint. Analogously, add the annotations for the missing methods. Please note that for lists a responseContainer must also be specified.

```
@Path("book-app") // Root resource
@SwaggerDefinition(tags = { @Tag(name = "book-app",
     description = "A Web Service to browse and order books.")})
@Api(value = "book-app", tags = {"book-app"})
public class EntryPoint {
     …
     @GET
     @Produces(MediaType.TEXT_PLAIN)
     @ApiOperation (value = "Welcome message",
                    notes = "Returns a welcome message")
     public String welcome() {...}

     @GET
     @Path("authors")
     @Produces(MediaType.APPLICATION_JSON)
     @ApiOperation (value = "Return all authors",
                    notes = "Returns all authors in the collection",
                    response = Author.class, responseContainer = "List")
     public Response authors() {...}

     @GET
     @Path("books/author/{authorId}")
     @Produces(MediaType.APPLICATION_JSON)
     @ApiOperation (value = "Return all books by an author",
          notes = "Return all books by an author with the given Id.",
          response = Book.class, responseContainer = "List")
     public Response booksByAuthor(@PathParam("authorId") int authorId) {...}

     …

     @GET
     @Path("books/{bookId}")
     @Produces(MediaType.APPLICATION_JSON)
     @ApiOperation (value = "Return book with given Id",
                    notes = "Returns the complete book (including " +
                              "sample text) with the given Id.",
                    response = Book.class)
     public Response book(@PathParam("bookId") int bookId) {...}

     @POST
     @Path("/order")
     @Consumes(MediaType.APPLICATION_JSON)
     @ApiOperation (value = "Sends an order")
     public Response confirmOrder(Order order) {...}
}
```

Now create the class `BookServerWithSwagger` in the package `server.swagger`, which replaces the previous `BookServer` (You can also find the class in Moodle):

```
public class BookServerWithSwagger {

     private static final int SERVER_PORT = 8080;
     private static Class<?> CLASS = EntryPoint.class;

     private static <T> void forceInit(Class<T> cls) {
         try {
             Class.forName(cls.getName(), true, cls.getClassLoader());
         } catch (ClassNotFoundException e) {
             throw new AssertionError(e);  // Can't happen
         }
     }

     private void buildSwaggerBean() {
          // This configures Swagger
          BeanConfig beanConfig = new BeanConfig();
          beanConfig.setVersion("1.0.0");
          beanConfig.setResourcePackage(CLASS.getPackage().getName());
          beanConfig.setScan(true);
          beanConfig.setBasePath("/api"); // "/"
          beanConfig.setDescription(
                    "REST-API to demonstrate Swagger with Jersey2 in an "
                    + "embedded Jetty instance, with no web.xml or Spring MVC.");
          beanConfig.setTitle("API");
     }

     private ContextHandler buildApplicationContext() {
          ResourceConfig resourceConfig = new ResourceConfig();
          resourceConfig.packages(CLASS.getPackage().getName(),
                         ApiListingResource.class.getPackage().getName());
          ServletContainer servletContainer =
                         new ServletContainer(resourceConfig);
          ServletHolder servletHolder = new ServletHolder(servletContainer);
          ServletContextHandler applicationContext =
                    new ServletContextHandler(ServletContextHandler.SESSIONS);
          applicationContext.setContextPath("/");
          applicationContext.addServlet(servletHolder, "/api/*"); // "/*

          return applicationContext;
     }

     // This starts the Swagger UI at http://localhost:PORT/api/docs
     private ContextHandler buildSwaggerUI() throws URISyntaxException{
          ResourceHandler resourceHandler = new ResourceHandler();
          resourceHandler.setResourceBase(this.getClass().getClassLoader()
               .getResource("swaggerui").toURI().toString());
          ContextHandler swaggerUIContext = new ContextHandler();
          swaggerUIContext.setContextPath("/api/docs/"); // "/docs
          swaggerUIContext.setHandler(resourceHandler);

          return swaggerUIContext;
     }
```

```
    public void startServer(){
        forceInit(EntryPoint.class);

        try {
            Resource.setDefaultUseCaches(false);
            buildSwaggerBean();
            HandlerList handlers = new HandlerList();
            handlers.addHandler(buildSwaggerUI());
            handlers.addHandler(buildApplicationContext());

            // Start server
            Server server = new Server(SERVER_PORT);
            server.setHandler(handlers);
            server.start();
            server.join();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    public static void main(String[] args) {
        // Open Swagger at: http://localhost:8080/api/docs;
        BookServerWithSwagger server = new BookServerWithSwagger();
        server.startServer();
    }
}
```

Now start the main method and open the documentation at http://localhost:8080/api/docs . Here you can now interactively try out your services directly. If you want you can also start your client at the same time.