Writing real time applications using Spring, AngularJS and WebSockets

In the previous tutorial I already configured the entire web application, so that part is already behind us. In this tutorial I will add the code necessary to make the data access + websockets + RESTful webservice work. I'm going to work my way from the bottom up to the top, so the first step is the data access layer.

Data access

The first thing to do is to specify the entity we're going to use. In this case I'm going to write a simple web application showing a list of ideas that can be voted on. This means we need three fields called `description`, `title` and `votes`. Our entity will look like:

```
@Entity
public class IdeaDto implements Serializable {

    private static final long serialVersionUID = -6809049173391335091L;

    @Id
    @GeneratedValue(strategy=GenerationType.IDENTITY)
    private int id;

    @Column
    private String title;

    @Column
    private String description;

    @Column
    private long votes;

    public IdeaDto() {
        super();
    }

    public int getId() {
        return id;
    }

    public void setId(int id) {
        this.id = id;
    }

    public String getTitle() {
        return title;
    }

    public void setTitle(String title) {
        this.title = title;
    }

    public String getDescription() {
        return description;
    }

    public void setDescription(String description) {
        this.description = description;
    }

    public long getVotes() {
        return votes;
    }

    public void setVotes(long votes) {
        this.votes = votes;
    }
}
```

Nothing special here, just a simple class with JPA annotations.

The next step is to provide a way to create/update/delete and read from our database. We're going to use **spring-data-jpa** for this. The only thing we need to define is a repository interface, for example:

```
public interface IdeaRepository  extends JpaRepository {

}
```

As you can see, you can leave this interface entirely empty, this will act as a proxy and all actions (CRUD) are already described in the `JpaRepository` interface. That's all we need for our data access layer.

Service layer

The next step is to do is to provide a small service that encapsulate all access to the repository and convert our entities in other beans (that can be used as our model). To do that I wrote a simple interface:

```
public interface IdeaService {

    List getIdeas();

    @Transactional
    Idea addIdea(Idea idea);

    @Transactional
    Idea updateIdea(Idea idea);

    @Transactional
    void deleteIdea(Idea idea);
}
```

And the implementation of it:

```
package be.g00glen00b.service.impl;

import java.util.ArrayList;
import java.util.List;

import org.dozer.Mapper;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;

import be.g00glen00b.dto.IdeaDto;
import be.g00glen00b.model.Idea;
import be.g00glen00b.repository.IdeaRepository;
import be.g00glen00b.service.IdeaService;

@Service
public class IdeaServiceImpl implements IdeaService {

    @Autowired
    private IdeaRepository repo;

    @Autowired
    private Mapper mapper;

    public List getIdeas() {
        List list = repo.findAll();
        List out = new ArrayList();
        for (IdeaDto dto : list) {
            out.add(mapper.map(dto, Idea.class));
        }
        return out;
    }

    @Transactional
    @Override
    public Idea addIdea(Idea idea) {
        IdeaDto dto = mapper.map(idea, IdeaDto.class);
        return mapper.map(repo.saveAndFlush(dto), Idea.class);
    }

    @Transactional
    @Override
    public Idea updateIdea(Idea idea) {
        IdeaDto dto = repo.findOne(idea.getId());
        dto.setDescription(idea.getDescription());
        dto.setTitle(idea.getTitle());
        dto.setVotes(idea.getVotes());
        return mapper.map(repo.saveAndFlush(dto), Idea.class);
    }

    @Transactional
    @Override
    public void deleteIdea(Idea idea) {
        repo.delete(idea.getId());
    }
}
```

As you can see here we mapped our `IdeaDto` to objects of the type `Idea`. Now all that's left is to define `Idea`. This model class will look quite similar to the entity, but in real situations you might have to change some things. An example, if you have an entity that contains a hashed version of a password, then you probably don't want to send that to the clients. To solve that we could remove the password field from the model and by mapping between them we "lose" the unwanted fields.

The code:

```
public class Idea {


    private int id;

    private String title;

    private String description;

    private long votes;

    public Idea() {
        super();
    }

    public int getId() {
        return id;
    }

    public void setId(int id) {
        this.id = id;
    }

    public String getTitle() {
        return title;
    }

    public void setTitle(String title) {
        this.title = title;
    }

    public String getDescription() {
        return description;
    }

    public void setDescription(String description) {
        this.description = description;
    }

    public long getVotes() {
        return votes;
    }

    public void setVotes(long votes) {
        this.votes = votes;
    }
}
```

Controller

The next step is to define the controller itself. This controller will provide the initial view and a small RESTful webservice to access the data. So first create a class and annotate it properly:

```
@Controller
@RequestMapping("/")
public class IdeaController {

    @Autowired
    private IdeaService service;

    // Controller mappings

}
```

The next step is to add a method mapped to `"/"` so we can deliver the HTML/JSP to the client, the code for that is:

```
@RequestMapping(method = RequestMethod.GET)
public String viewIdeas() {
    return "ideas";
}
```

Due to the view mapping we made in our earlier project, `"ideas"` is actually referring to the view **/WEB-INF/views/ideas.jsp**. The next step is that we add our RESTful webservice to it:

```
@RequestMapping(value = "/ideas", method = RequestMethod.GET)
public @ResponseBody List getIdeas() {
    return service.getIdeas();
}

@NotifyClients
@RequestMapping(value = "/ideas/{id}", method = RequestMethod.PUT)
public @ResponseBody Idea update(@PathVariable int id, @RequestBody Idea idea) {
    idea.setId(id);
    Idea out = service.updateIdea(idea);
    return out;
}

@NotifyClients
@RequestMapping(value = "/ideas", method = RequestMethod.POST)
public @ResponseBody Idea add(@RequestBody Idea idea) {
    Idea out = service.addIdea(idea);
    return out;
}

@NotifyClients
@RequestMapping(value = "/ideas/{id}", method = RequestMethod.DELETE)
@ResponseStatus(HttpStatus.NO_CONTENT)
public void delete(@PathVariable int id) {
    Idea task = new Idea();
    task.setId(id);
    service.deleteIdea(task);
}
```

As you can see here there are a lot of annotations. First we're mapping the REST calls to `/ideas/`, then we also add a path variable called `id` so when we want to change an idea we should use **/ideas/1** where 1 is the ID of the idea. We're returning a plain object here, but since we're using the `@ResponseBody` annotation and we have included Jackson, the object will be translated to JSON.

What we also used is a custom annotation called `@NotifyClients`. This annotation will be used by our aspect, so after the annotated method is executed, all clients are notified, allowing them to retrieve the new data.

Notifying clients

Right now we could actually just use the web application (if you provide the view). TO make our application real time we will have to write some code that is injected in our controllers through aspects. The first step we need to complete is to create the `@NotifyClients` annotation that I used in the controller. This annotation is quite simple:

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface NotifyClients {

}
```

Then we can define an aspect that will look for the `@NotifyClients` annotation for each method in the controller package. In this case I wrote the following simple aspect:

```
@Aspect
public class NotifyAspect {

    @Autowired
    private SimpMessagingTemplate template;

    private static final String WEBSOCKET_TOPIC = "/topic/notify";

    @Pointcut("@annotation(be.g00glen00b.aspects.NotifyClients)")
    public void notifyPointcut() {}

    @Pointcut("execution(* be.g00glen00b.controller.**.*(..))")
    public void methodPointcut() {}

    @After("methodPointcut() && notifyPointcut()")
    public void notifyClients() throws Throwable {
        template.convertAndSend(WEBSOCKET_TOPIC, new Date());
    }

}
```

This aspect has two pointcuts, the `notifyPointcut()` looks for all methods annotated with our annotation and the `methodPointcut()` looks for all methods in the controllers-package. We then weave our code into the controller **after** the given pointcuts. We have to wait until all changes are persisted before we can notify the clients. When such a controller method is executed, it will send the current timestamp to a websocket called `/topic/notify`. We will also use this topic in the client-side code to subscribe on.

But with this aspect we actually defined our entire back-end code. The next tutorial will show you the code that will be used in the client.