Developing CRM Web App REST Client for CRM REST API

Introduction

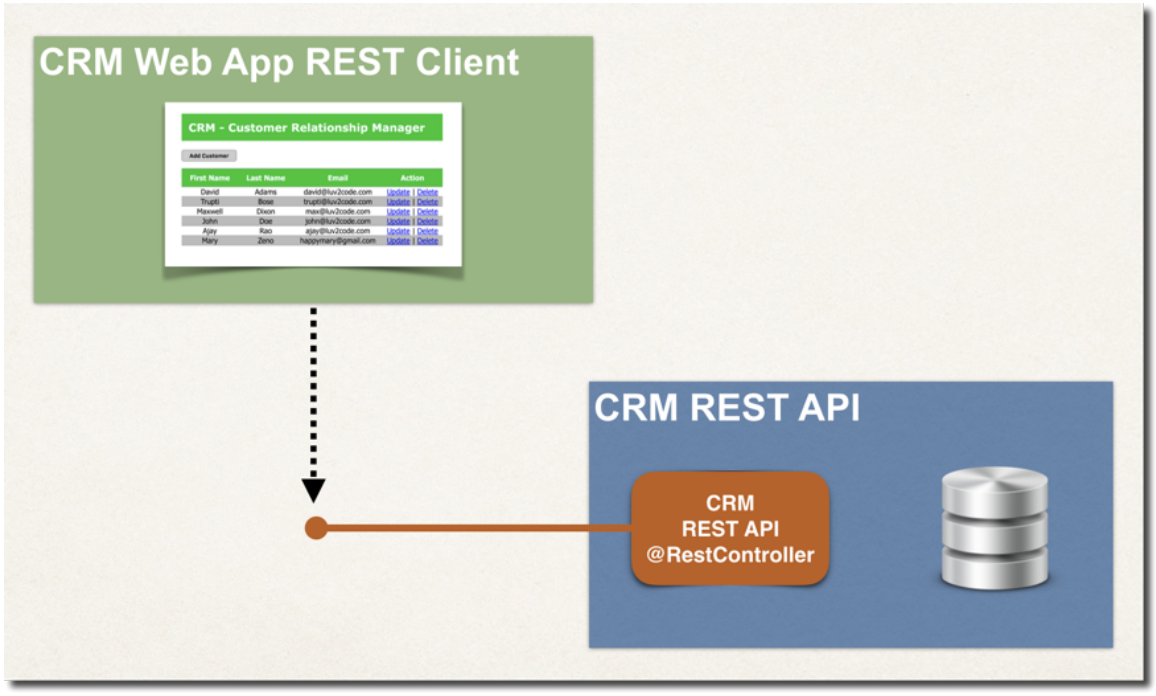
In this document, you will learn how create a CRM Web App REST client that communicates with the CRM REST API.

In the course videos, we previously created a REST API for the CRM application. This REST API exposed the following endpoints.

We tested this REST API using the Postman client. This allowed us to verify the functionality of the REST API.

However, we also previously created a CRM web app using JSP. Wouldn't it be great if we could have the CRM web app leverage the CRM REST API for data? We'll cover this functionality in this document.

This is our desired architecture.



During our discussion, I'll refer to the client app as **CRM Web App REST Client**. I'll refer to the backend REST API as **CRM REST API**.

Overview of Steps

1. Application Architecture
2. Review project structure
3. Maven POM file
4. Application configuration properties file
5. Spring Bean configuration for RestTemplate
6. GET: Get a list of customers
7. GET: Get a single customer by id
8. POST: Add a new customer
9. PUT: Update an customer
10. DELETE: Delete an customer

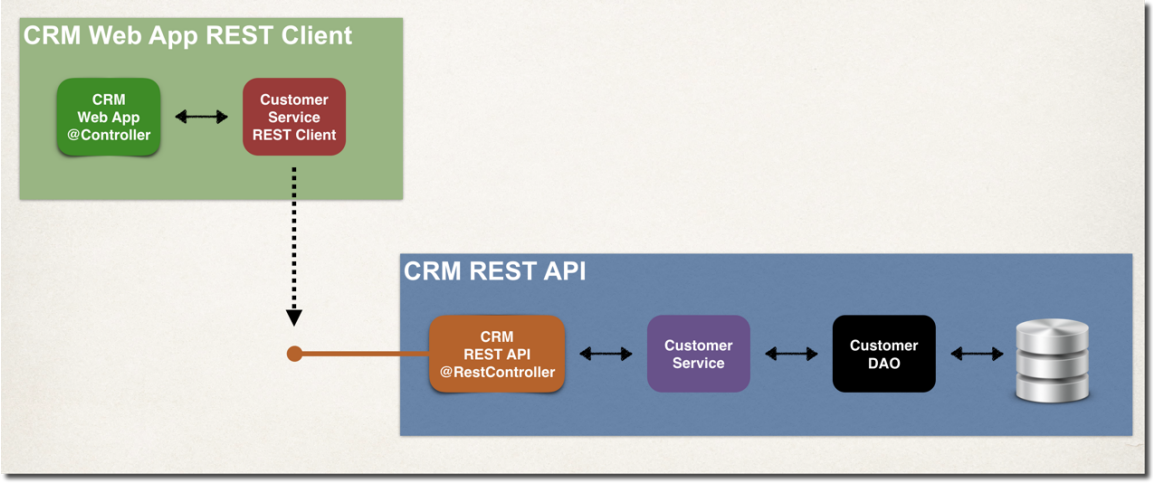
Source Code

The full source code is available for download.

http://www.luv2code.com/crm-web-app-rest-client-demo-code

1. Application Architecture

At a low-level, the application will have the following architecture.



The CRM Web App REST Client will have a new CustomerService implementation. In a previous version of the application, the CRM Web App REST Client would use a service and a Hibernate DAO implementation. However, the Hibernate DAO code is no longer required in the CRM Web App REST Client. The database interaction is now handled by the backend CRM REST API.

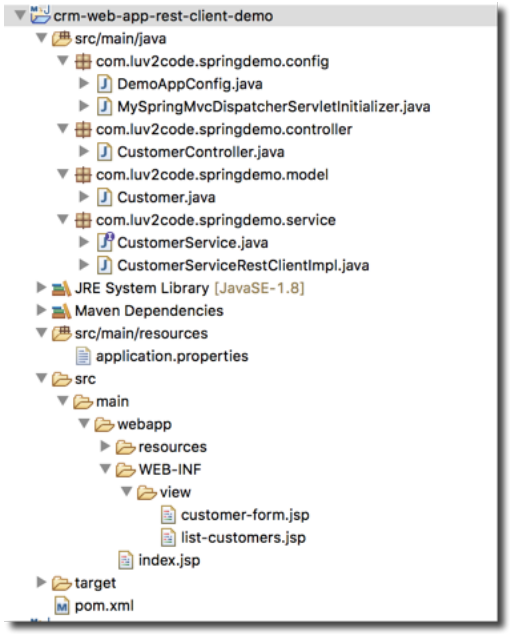
In this version of the CRM Web App REST Client, we will simply make REST client calls to the CRM REST API to retrieve the data.

2. Review Project Structure

We have the source code for the CRM Web App REST Client available online. Download it from this link

Unzip the project and import it into your IDE as a Maven project.

Here's a description of the key components of the project.



3. Maven POM File

For the CRM Web App REST Client, the Maven POM file is slightly modified. Here's a snippet of the Maven properties and dependencies

File: pom.xml

```
…
    <properties>
            <springframework.version>5.1.7.RELEASE</springframework.version>

            <maven.compiler.source>1.8</maven.compiler.source>
            <maven.compiler.target>1.8</maven.compiler.target>
    </properties>

    <dependencies>

            <!-- Spring -->
            <dependency>
                    <groupId>org.springframework</groupId>
                    <artifactId>spring-webmvc</artifactId>
                    <version>${springframework.version}</version>
            </dependency>
            <dependency>
                    <groupId>org.springframework</groupId>
                    <artifactId>spring-tx</artifactId>
                    <version>${springframework.version}</version>
            </dependency>
            <dependency>
                    <groupId>org.springframework</groupId>
                    <artifactId>spring-orm</artifactId>
                    <version>${springframework.version}</version>
            </dependency>

            <!-- Add Jackson for JSON converters -->
            <dependency>
                    <groupId>com.fasterxml.jackson.core</groupId>
                    <artifactId>jackson-databind</artifactId>
                    <version>2.9.5</version>
            </dependency>

    …

    </dependencies>
…
```

Notice that that there is no need for Hibernate dependencies or MySQL JDBC dependencies. All of the database access is handled by the backend CRM REST API.

The only new entry needed is for Jackson. We'll use Jackson to automatically convert the JSON data to Java objects.

4. Application Configuration Properties

The CRM Web App REST Client needs to connect to the backend REST API. To accomplish this, we will add a configuration property that has the url of the REST API.

File: src/main/application.properties

```
#
# The URL for the CRM REST API
# - update to match your local environment
#
crm.rest.url=http://localhost:8080/spring-crm-rest/api/customers
```

5. Spring Bean configuration for RestTemplate

Spring includes the helper class, RestTemplate, for making REST client calls.

Here is the JavaDoc link for the RestTemplate class. Please refer to it for the methods available in the class.

We can create an instance of this class as a Spring Bean. As a result, we can easily inject it into any of our components.

File: DemoAppConfig.java

```
@Configuration
@EnableWebMvc
@ComponentScan("com.luv2code.springdemo")
@PropertySource({ "classpath:application.properties" })
public class DemoAppConfig implements WebMvcConfigurer {

…
    // define bean for RestTemplate
    // this is used to make client REST calls

    @Bean
    public RestTemplate restTemplate() {
           return new RestTemplate();
    }
…
}
```

This code also loads the application.properties file with the @PropertySource annotation.

Next, we can inject this bean into our CustomerServiceRestClientImpl

File: CustomerServiceRestClientImpl.java

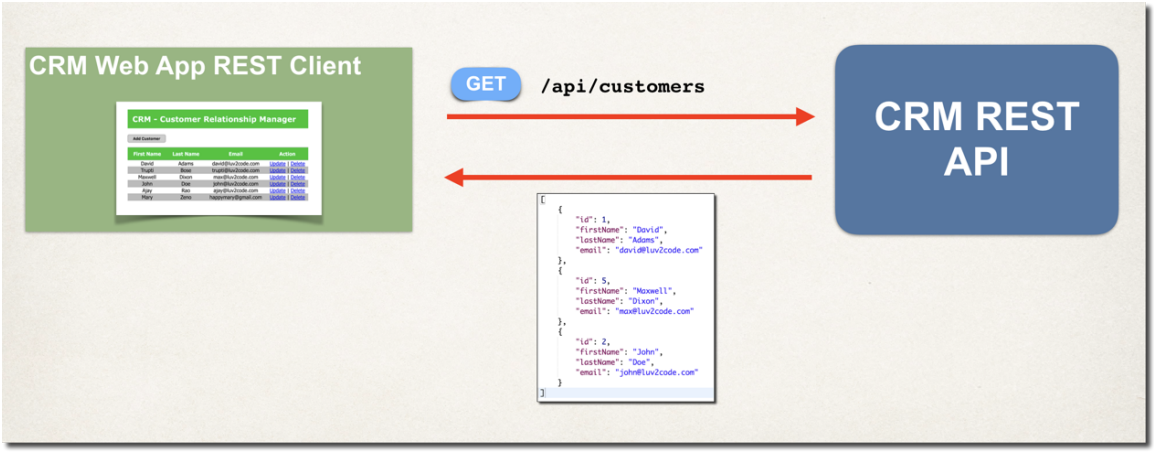
```
@Service
public class CustomerServiceRestClientImpl implements CustomerService
{
    private RestTemplate restTemplate;

    private String crmRestUrl;

    private Logger logger = Logger.getLogger(getClass().getName());

    @Autowired
    public CustomerServiceRestClientImpl(
                        RestTemplate theRestTemplate,
                        @Value("${crm.rest.url}") String theUrl) {

            restTemplate = theRestTemplate;
            crmRestUrl = theUrl;

            logger.info("Loaded property:  crm.rest.url="
                        + crmRestUrl);

    }

…
}
```

Also, notice that we inject the crm.rest.url property using the @Value annotation. This is an example of constructor injection. As another option, we could have injected the dependencies using field injection.

6. GET: Get a list of Customers

The REST API exposes the following endpoint for getting a list of Customers.

GET /api/customers



Here's the code to make the REST client call.

File: CustomerServiceRestClientImpl.java

```
@Override
public List<Customer> getCustomers() {

        logger.info("in getCustomers(): Calling REST API "
                        + crmRestUrl);

        // make REST call
        ResponseEntity<List<Customer>> responseEntity =
                        restTemplate.exchange(crmRestUrl,
                        HttpMethod.GET, null,
                new ParameterizedTypeReference<List<Customer>>() {});

        // get the list of customers from response
        List<Customer> customers = responseEntity.getBody();

        logger.info("in getCustomers(): customers" + customers);

        return customers;
}
```

In this code, we make a call to get the customers. The main code happens with the method call: restTemplate.exchange(….)

This method has the following parameters

- url: the URL

- method: the HTTP method for GET
- requestEntity: - additional request headers or body, null in our case
- responseType - the type of the return value. ParameterizedTypeReference is used to pass generic type information:

The REST API returns a JSON object. In the background, Spring uses Jackson to convert the JSON data to a list of Customer objects with List<Customer>.

7. GET: Get a single Customer by ID

The REST API exposes the following endpoint for getting a single customer by ID.

GET /api/customers/{customerId}

Here's the code to make the REST client call.

File: CustomerServiceRestClientImpl.java

```
@Override
public Customer getCustomer(int theId) {

	logger.info("in getCustomer(): Calling REST API "
			+ crmRestUrl);

	// make REST call
	Customer theCustomer =
		restTemplate.getForObject(crmRestUrl + "/" + theId,
					Customer.class);

	logger.info("in saveCustomer(): theCustomer="
			+ theCustomer);

	return theCustomer;
}
```

The main work happens in the method restTemplate.getForObject(…). The method makes an HTTP GET request to the URL.

The method has the following parameters

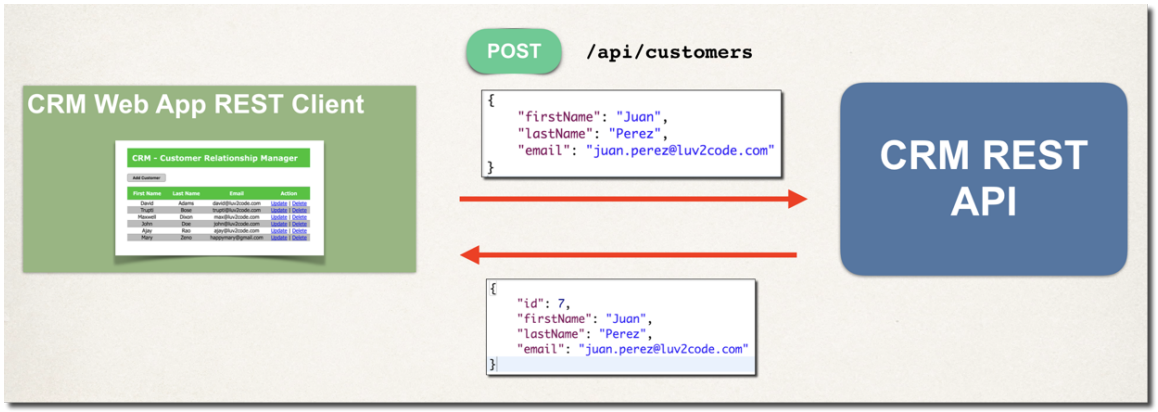
- url: the URL
- responseType: the type of the return value

As discussed earlier, the REST API returns the data as JSON. Jackson handles converting the JSON to a Customer object.

8. POST: Add a new Customer

The REST API exposes the following endpoint for adding a new customer.

POST /api/customers



The body of the request will contain the Customer data.

Here's the code to make the REST client call.

File: CustomerServiceRestClientImpl.java

```
@Override
public void saveCustomer(Customer theCustomer) {

	logger.info("in saveCustomer(): Calling REST API "
			+ crmRestUrl);

	int employeeId = theCustomer.getId();

	// make REST call
	if (employeeId == 0) {
		// add employee
		restTemplate.postForEntity(crmRestUrl, theCustomer,
						String.class);


	} else {
		// update employee
		restTemplate.put(crmRestUrl, theCustomer);
	}

	logger.info("in saveCustomer(): success");
}
```

When we add a customer, the customer ID is 0, so we'll execute the following snippet:

```
        int employeeId = theCustomer.getId();

        // make REST call
        if (employeeId == 0) {
               // add employee
               restTemplate.postForEntity(crmRestUrl,
                                          theCustomer, String.class);
        }
```

The main code happens in the method: restTemplate.postForEntity(…). The method sends an HTTP POST to the URL.

The method has the following parameters

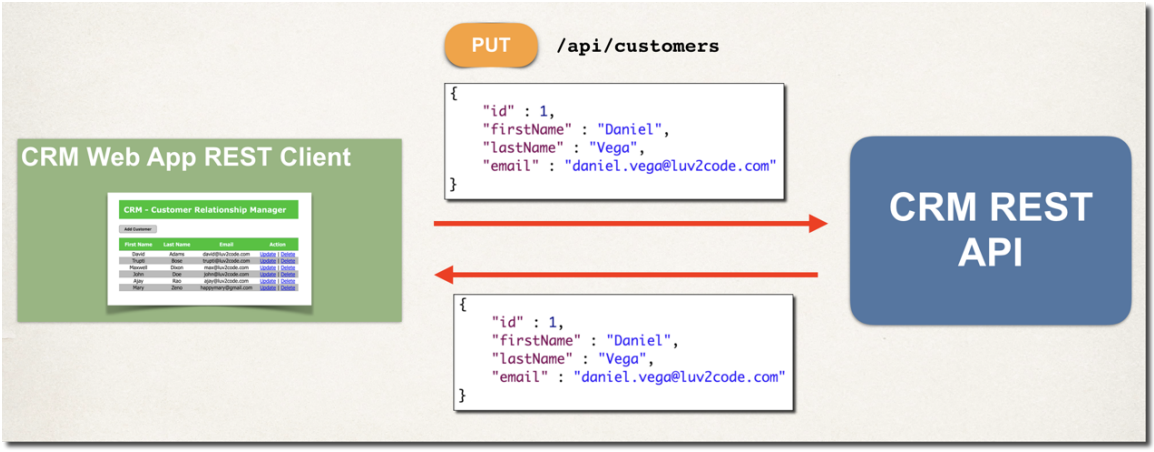
- url - the URL
- request - the Object to be POSTed
- responseType - the type of the response

As usual, Jackson handles converting the Java objects to JSON and vice-versa.

9. PUT: Add an existing Customer

The REST API exposes the following endpoint for updating an existing customer.

PUT /api/customers



Here's the code to make the REST client call.

File: CustomerServiceRestClientImpl.java

```
@Override
    public void saveCustomer(Customer theCustomer) {

            logger.info("in saveCustomer(): Calling REST API "
                        + crmRestUrl);

            int employeeId = theCustomer.getId();

            // make REST call
            if (employeeId == 0) {
                    // add employee
                    restTemplate.postForEntity(crmRestUrl, theCustomer,
                                                    String.class);


            } else {
                    // update employee
                    restTemplate.put(crmRestUrl, theCustomer);
            }

            logger.info("in saveCustomer(): success");
    }
```

When we update a customer, the customer ID is not equal to 0, so we'll execute the following snippet:

```
		} else {
			// update employee
			restTemplate.put(crmRestUrl, theCustomer);
		}
```

The main code happens in the method: restTemplate.put(…). The method sends an HTTP PUT to the URL.

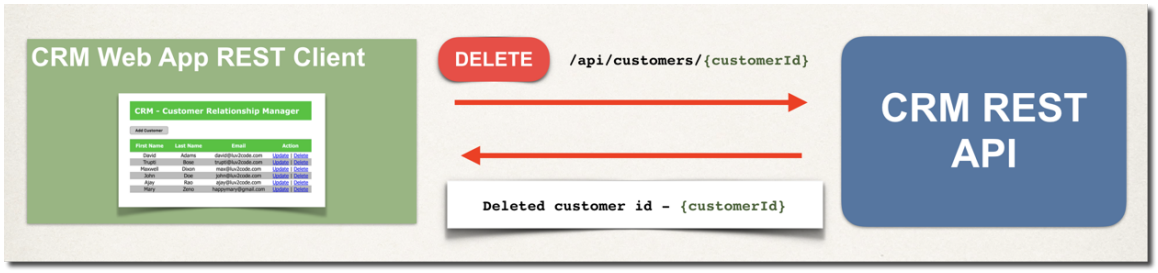
The method has the following parameters

- url - the URL
- request - the Object to PUT

10. DELETE: Delete a Customer

The REST API exposes the following endpoint for deleting a customer.

DELETE /api/customers/{customerId}



Here's the code to make the REST client call.

File: CustomerServiceRestClientImpl.java

```
@Override
public void deleteCustomer(int theId) {

        logger.info("in deleteCustomer(): Calling REST API "
                    + crmRestUrl);

        // make REST call
        restTemplate.delete(crmRestUrl + "/" + theId);

        logger.info("in deleteCustomer(): deleted customer theId="
                     + theId);
}
```

The main work happens in the method restTemplate.delete(…). This method sends an HTTP DELETE to the URL.

The method has the following parameter

- url - the URL

Now, when you run the CRM Web App REST Client, it makes all of the requests to the backend REST API. That's it!

Source Code

The full source code is available for download.

http://www.luv2code.com/crm-web-app-rest-client-demo-code