# Full Stack Project for beginners: Volume one

Author: A.J.L.
Contact: ajlBooksInk@gmail.com

Topics Menu

- Relation database:  MySql
- Java coding using Eclipse.
- Maven and pom file.
- Use of Spring Frameworks 'Spring Boot'
- Postman
- A React application

## Table of Contents

# Data Base

## Data Base Installation

We use a MySQL Server – a free relational database server.

We create a database user: Smithy and you get to pick Smithy's password!

The following MySQL install on Windows took me about 5 minutes.

- https://dev.mysql.com/downloads/installer/
- No thanks, just start my download.
- Choosing a Setup Type
- Server only
- Type and Networking: use the checked off defaults. Note port ***3306***
- Enter and remember the root user ***password***
- Add a new user ***Smithy*** enter and remember Smithy's ***password***
- Execute : Apply Configuration– it will decide what to do
- Finish
- Look at the ***log file*** via 'Copy Log to Clipboard'

- ✓ REMEMBER the root user password
- ✓ REMEMBER user Smithy
- ✓ REMEMBER port 3306
- ✓ REMEMBER Smithy's password – I used Smithy!8787

In the **log file** you should see a bunch of stuff; including a path like

Directory: C:\Program Files\MySQL\MySQL Server 8.0\bin\

- ✓ REMEMBER C:\Program Files\MySQL\MySQL Server 8.0\bin\

*Connect*: Open a DOS window (Command Prompt) add type

- cd C:\Program Files\MySQL\MySQL Server 8.0\bin\
- mysql -u Smithy -p

See the prompt and enter Smithy's *password*

You can enter various MySQL DB commands; they end in a semicolon. Type:

- show databases;
- create database ***users***;
- show databases;
- use ***users***;

We have created a database called ***users***.

- ✓ REMEMBER the database name ***users***

## Database tables

*Connect* to mySQL and create three tables for our two related entities: User and Task. Note the primary keys and foreign key constraints.

```
use users;

create table user(
  id     char(4) not null,
  name   varchar(132) not null,
  email  varchar(132) not null,
  PRIMARY KEY(id) );

create table taskStatus(
  status varchar(24)not null,
  isDefault int,
  PRIMARY KEY(status) );

create table task(
  user_id char(4) not null,
  item  varchar(256) not null,
  due  date not null,
  status varchar(24) not null,
  PRIMARY KEY(user_id, item, due),

  CONSTRAINT fk_user
   FOREIGN KEY (user_id)
   REFERENCES user(id),

  CONSTRAINT fk_task_status
   FOREIGN KEY (status)
   REFERENCES taskStatus(status));
```

A Task **MUST** belong to a user. A *Task's* status **MUST** be one of the values from the *task status* table. Again, take note of our Primary Keys and Foreign Key constraints.

Next, we populate the database;  afterwards you can look at the data via

- select * from task;
- select * from taskStatus;
- select * from user;

## Populating the Data base

Task status table. The ‘is default’ attribute of value 1 labels the ON-TIME status as the default status for any new Task that we create. A Task is ‘on-time’ when it is first created! Later, we will see how this default in the database relieves our React application of choosing a default. In the spirit of KID (keep it dumb).

```
insert taskStatus values ('PENDING',0);
insert taskStatus values ('COMPLETE',0);
insert taskStatus values ('CANCELLED', 0);
insert taskStatus values ('ON-HOLD', 0);
insert taskStatus values ('DUPLICATE-CANCELLED', 0);
insert taskStatus values ('ON-TIME', 1);
insert taskStatus values ('BEHIND-SCHEDULE', 0);
```

User table – feel free to make your own!

```
insert user values ('0001','A Lopes',  'ALopesMail.com');
insert user values ('0002','B Lopes',  'BLopesMail.com');
insert user values ('0003','C Lopes',  'CLopesMail.com');
insert user values ('0004','D Lopes',  'DLopesMail.com');
insert user values ('0005','E Lopes',  'ELopesMail.com');
insert user values ('0006','F Lopes',  'FLopesMail.com');
insert user values ('0016','A Looper', 'AlooperMail.com');
insert user values ('0017','P Looper', 'PlooperMail.com');
insert user values ('0018','Kei Lee',  '@KLMail.com');
insert user values ('0019','Joe Key',  '@JKeyMail.com');

insert user values ('1024','Amy Hoops','AHOPPsMail.com');
insert user values ('2024','J Done',   'HappyInkCorpMail.com');
insert user values ('2025','J Aone',   'HappyInkCorpMail.com');
insert user values ('2026','J Bone',   'HappyInkCorpMail.com');
insert user values ('2027','J J Cone', 'HappyInkCorpMail.com');
insert user values ('2028','J Done III', 'HappyInkCorpMail.com');
insert user values ('2029','J Eone', 'HappyInkCorpMail.com');
insert user values ('2014','J Fone', 'HappyInkCorpMail.com');
insert user values ('2024','J Gone', 'HappyInkCorpMail.com');

insert user values ('2034','J Hone Jr.', 'HappyInkCorpMail.com');
insert user values ('2044','J Ione I', 'HappyInkCorpMail.com');
insert user values ('2054','J Jone', 'HappyInkCorpMail.com');
insert user values ('2064','J Kone', 'HappyInkCorpMail.com');
insert user values ('2074','J Lone', 'HappyInkCorpMail.com');
insert user values ('2084','J Mone', 'HappyInkCorpMail.com');
insert user values ('2094','J None', 'HappyInkCorpMail.com');
insert user values ('2124','J Zone', 'HappyInkCorpMail.com');
insert user values ('2124','J Pone', 'HappyInkCorpMail.com');
insert user values ('2224','Jegg DZone', 'HappyInkCorpMail.com');

insert user values ('2324','Gem Done II', 'HappyInkCorpMail.com');
insert user values ('2424','Laura Zone', 'HappyInkCorpMail.com');
```

Task table

```
insert task values ('0017','File Corp Taxes',    '2021-04-15' , 'COMPLETE');
insert task values ('0017','Attend Corp Annual Meeting',  '2021-12-15' , 'CANCELLED');
insert task values ('0001','React Training 101',   '2021-08-28' , 'ON-TIME');
insert task values ('0001','Vacation',  '2021-07-15' , 'COMPLETE');
insert task values ('1024','Leave',  '2022-11-15' , 'PENDING');

insert task values ('1024','Deliver Release 2.9.88',   '2021-11-16' , 'ON-TIME');
insert task values ('1024','Visit Gunther',  '2021-11-17' , 'PENDING');
insert task values ('1024','Outing: Base Ball Game', '2021-09-18' , 'PENDING');
insert task values ('1024','Vacation Day',   '2021-11-19' , 'PENDING');
insert task values ('1024','Deliver Widget Project',      '2021-11-11' , 'ON-TIME');

insert task values ('1024','Send Pidget project for integration testing',   '2021-11-01' ,
'BEHIND-SCHEDULE');

insert task values ('1024','Group meeting',   '2021-11-01' , 'PENDING');
insert task values ('1024','Paint office',  '2021-12-12' , 'PENDING');
insert task values ('1024','Buy new PCs',    '2021-11-15' , 'PENDING');
insert task values ('2027','New office furniture',    '2021-08-15' , 'ON-TIME');
```

Feel free to make your own!

Try these; they should all fail!

1. insert task values ('1024', 'Do my homework', 'TODAY', 'PENDING');
2. insert task values ('XXX', 'Do my homework', '2021-11-15', 'PENDING');
3. insert task values ('1024', 'Do my homework',  '2021-11-15', 'NOT-READY');
4. insert task values ('1024', 'Do my homework',  '2021-13-15', 'PENDING');

## Our java server project

### Install Eclipse

Go to: https://www.eclipse.org/downloads/

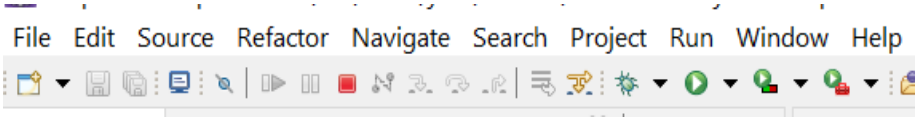
See the following and download.

**Get:** Eclipse IDE 2021-06

Install your favorite desktop IDE packages.

Download x86_64

Open eclipse and find the File tab




Pick/click:

- File
- New
- Project
- Web
- Dynamic Web Project
- Enter a project name of your liking: I will use **FullStack**
- Target Runtime
- Download and Install: Apache Tomcat V8.0 (if not already installed)
- *Pick* (create first! if needed) a local directory for the Tomcat installation
- Finish

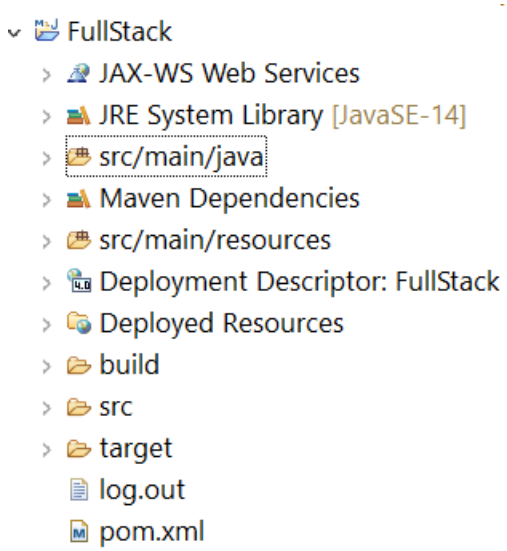
You should see your project in the eclipse **Project Explorer**. Right click on your project name

Pick/click:

- Configure
- Convert to Maven Project
- Finish

Magically we now have a pom.xml file. This is now a Maven Project.

You see something very much like this.



Open the pom file. Find the two lines in the pom file

```
<packaging>war</packaging>
<build>
```

In between the two lines add: (I use **bold** here or emphasis)

```
<parent>
   <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>2.4.3</version>
</parent>

<dependencies>
   <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-data-jpa</artifactId>
   </dependency>

   <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-data-rest</artifactId>
   </dependency>

   <dependency>
            <groupId>mysql</groupId>
            <artifactId>mysql-connector-java</artifactId>
            <scope>runtime</scope>
   </dependency>
</dependencies>

<properties>
   <java.version>1.8</java.version>
</properties>
```

We have just asked maven to grab the following java stuff we need for our application

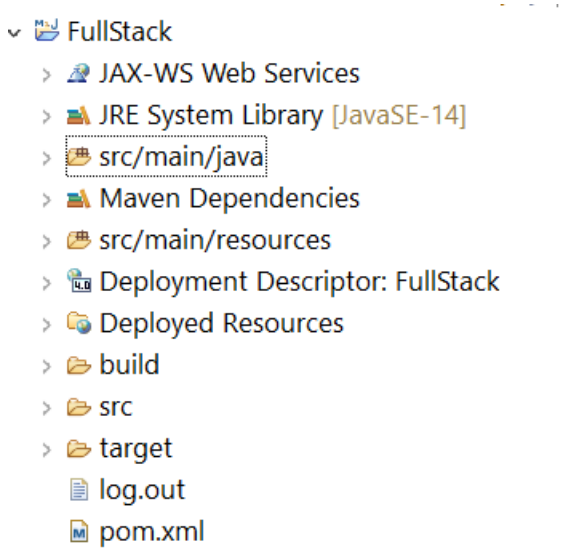
- Spring Boot
- Spring JPA (Java Persistence API) - to help data base work
- MySQL java API
- Spring REST support for GET, POST, etc.

Now we ask maven to do an update.

Pick/click:

- Right click on your project in your eclipse **Project Explorer**
- Maven
- Update Project
- OK

Right click in Project Explorer and pick **Refresh**. Expand the 'Maven Dependences;' look at all the jar files that now belong to your project!




Ready to code!

## Java Coding

Term: POJO are simple plain old java classes! (see our entity examples below)

Term: MVC . We follow the Spring provided MVC framework (Model-View-Controller).

**Model**: Regarding our DB tables we implement corresponding Java POJOs

1. User
2. Task
3. TaskStatus

## Creating a Package in Eclipse

In Project Explorer, right click on your project's folder **src/main/java**.

- New
- Package
- Pick the new package's name:, i.e**., the name YOU choose**

## Package: entities

Create the package **entities.** In Project Explorer, right click on your project's folder **src/main/java**.

- New
- Package
- Pick the new package's name: **entities**

Create a new java class **User.** Right click on the **entities** package

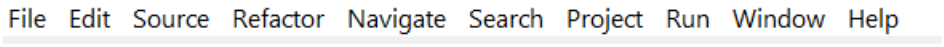
- New
- Class
- class name **User**

**Package**: entities

**Class:** User

```
package entities;

public class User {
	public User() {
	}

	private String id;
	private String name;
	private String email;
}
```

E*clipse magic* follows.




Find and click the **Source** tab on top. Choose these two:

- Generate Getters and Setters
- Generate toString

You should see Eclipse generated setter and getters methods, along with a toString() method!

**Package**: entities

**Class:** Task

```
package entities;

public class Task {

	public Task() {
	}

	private String userId;
	private String item;
	private String dueDate;
	private String status;
}
```

Again, use the above *eclipse magic* to complete the class.

**Package**: entities

**Class:** Task Status

```
package entities;

public class TaskStatus {

	public TaskStatus() {
	}

	private String status;
	private int isDefault;
}
```

Again, use the above *eclipse magic* to complete the class.

## Package: logger

Normally – in a 'real world application' - we would use an industry standard logging API for our logging. To keep this example as small and simple as possible we 'roll our own' with a simple logger to the console.

Create the package **logger.**

**Package**: logger

**Class:** Logger

```
package logger;

import java.sql.Timestamp;

public class Logger {

	private static String TS(String msg) {
		return ""+new Timestamp(System.currentTimeMillis()).getTime()+" "+msg;
	}

	public static void logError(String msg) {
		System.out.println(TS(msg));
	}

	public static void logError(String msg, Exception E) {
		System.out.println(TS(msg)+ " " + E);
	}

	public static void logWarning(String msg) {
		System.out.println(TS(msg));
	}

	public static void logWarning(String msg, Exception E) {
		System.out.println(TS(msg) + " " + E);
	}

	public static void logInfo(String msg) {
		System.out.println(TS(msg));
	}
}
```

## Package: dbInterface

Create the package **dbInterface.**

**Package**: dbInterface

**Class**: Db Connector

This class creates and returns a database connection via a Data Source instance

```
package dbInterface;
import javax.sql.DataSource;
import java.sql.Connection;
import java.sql.SQLException;

class DbConnector {
	static Connection makeConnection(DataSource ds) throws Exception {
		int RETRIES = 3;
		Connection cc = null;
		Exception  E = null;
		int tries = 0;

		while (tries < RETRIES) {
			try {
				cc = ds.getConnection();
				if (cc != null) {
					break;
				}
			} catch (SQLException e) {
				E = e;
			}
			tries++;
			Thread.sleep(1000);  // one second sleep in between retries
		}

		if (cc == null) {
			if (E != null) {
				throw E;
			} else {
				throw new Exception ("Data Base Connection failed");
			}
		}
		return cc;
	}
}
```

Create data base related  classes for managing our database transactions

**Package**: dbInterface

**Class**: Task Fetcher

This class builds a list of *Task* and *Task Status* objects from a database query (select) via the connection

```
package dbInterface;

import java.sql.Connection;
import java.sql.ResultSet;
import java.sql.Statement;
import java.util.ArrayList;
import java.util.List;
import entities.Task;
import entities.TaskStatus;

class TaskFetcher {

	private static Task buildTask(ResultSet rs) throws Exception {

		Task task = new Task();
		task.setUserId(rs.getString("userId"));
		task.setItem(rs.getString("item"));
		task.setDueDate(rs.getString("due"));
		task.setStatus(rs.getString("status"));

		return task;
	}
```

```
static List<TaskStatus> selectTaskStatusSet(Connection conn)
  throws Exception {
        List<TaskStatus> set = new ArrayList<TaskStatus>();

        String sql = "select status, IFNULL(isDefault, 0) dflt from taskStatus";

        Statement stmt = null;
        ResultSet rs = null;

        try {
                stmt = conn.createStatement();
                rs =  stmt.executeQuery(sql);
                while (rs.next()) {
                        TaskStatus status = new TaskStatus();
                        status.setStatus(rs.getString("status"));
                        status.setIsDefault(rs.getInt("dflt"));
                        set.add(status);
                }
        } catch (Exception e) {
                throw e;
        }

        try {
                if (rs != null)   rs.close();
                if (stmt!=null)   stmt.close();
        } catch (Exception e) {
        }

        return set;
}
```

```
	static List<Task> selectTasks(Connection conn) throws Exception {
		List<Task> list = new ArrayList<Task>();

		String sql =
		"select user_id userId, item, due, status from task order by user_id";

		Statement stmt = null;
		ResultSet rs = null;

		try {
			stmt = conn.createStatement();
			rs =  stmt.executeQuery(sql);
			while (rs.next()) {
				Task task = buildTask(rs);
				list.add(task);
			}
		} catch (Exception e) {
			throw e;
		}

		try {
			if (rs != null)   rs.close();
			if (stmt!=null)   stmt.close();
		} catch (Exception e) {
		}

		return list;
	}

}
```

**Package**: dbInterface

**Class**: Task Inserter

This class inserts a Task into the database via the connection.

```
package dbInterface;
import java.sql.Connection;
import java.sql.Statement;
import entities.Task;

class TaskInserter {
	private static String tick (String str) {
		return "'" + str + "'";
	}

	static void insertTask(Connection conn, Task task) throws Exception {
		final String COMMA = "," ;
		String sql = "insert task values (" +
		   tick(task.getUserId())     + COMMA +
		   tick(task.getItem())       + COMMA +
		   tick(task.getDueDate())  + COMMA +
		   tick(task.getStatus()) + ")";
		Statement stmt = null;
		int insertCount=0;

		try {
			stmt = conn.createStatement();
			insertCount =  stmt.executeUpdate(sql);
		} catch (Exception e) {
			throw e;
		}

		try {
			if (stmt!=null) {
				stmt.close();
			}
		} catch (Exception e) {
		}

		if (insertCount != 1) {
			throw new Exception ("Task Insert failed");
		}
	}
}
```

**Package**: dbInterface

**Class**: Task Updater

This class updates a Task's due date and status in the database via the connection. Note that the Task Update Data instance contains two tasks.

```
package dbInterface;

import java.sql.Connection;
import java.sql.Statement;
import entities.Task;
import webControl.TaskUpdateData;

class TaskUpdater {

	private static String tick (String str) {
		return "'" + str + "'";
	}

	static void updateTask(Connection conn, TaskUpdateData updateData)
			throws Exception {

		Task taskBeforeUpdate = updateData.getTaskBeforeUpdate();
		Task taskAfterUpdate = updateData.getTaskAfterUpdate();

		String sql = "update task set"
				+ " status = " + tick(taskAfterUpdate.getStatus())
			+  ", due = "  + tick(taskAfterUpdate.getDueDate())
				+ " where"
				+ " user_id = "  + tick(taskBeforeUpdate.getUserId())
				+ " and item = " + tick(taskBeforeUpdate.getItem())
				+ " and due = "  + tick(taskBeforeUpdate.getDueDate());


		Statement stmt = null;
		int updateCount=0;

		try {
			stmt = conn.createStatement();
			updateCount =  stmt.executeUpdate(sql);

		} catch (Exception e) {
			throw e;
		}
```

```
		try {
			if (stmt!=null) {
				stmt.close();
			}
		} catch (Exception e) {
		}

		if (updateCount != 1) {
			throw new Exception ("Task Insert failed");
		}
	}
}
```

**Package**: dbInterface

**Class**: User Fetcher

This class builds lists of *User* objects from a database query via the connection

```
package dbInterface;
import java.sql.Connection;
import java.sql.ResultSet;
import java.sql.Statement;
import java.util.ArrayList;
import java.util.List;
import entities.User;

class UserFetcher {
	private static User buildUser(ResultSet rs) throws Exception {
		User user = new User();
		user.setId(rs.getString("id"));
		user.setEmail(rs.getString("email"));
		user.setName(rs.getString("name"));
		return user;
	}
	static List<User> selectUsers(Connection conn) throws Exception {
		List<User> list = new ArrayList<User>();
		String sql = "select id, name, email from  user";
		Statement stmt = null;
		ResultSet rs = null;
```

```
                try {
                        stmt = conn.createStatement();
                        rs =  stmt.executeQuery(sql);
                        while (rs.next()) {
                                User user = buildUser(rs);
                                list.add(user);
                        }
                } catch (Exception e) {
                        throw e;
                }
                try {
                        if (rs != null)   rs.close();
                        if (stmt!=null)   stmt.close();
                } catch (Exception e) {
                }
                return list;
        }
}
```

**Package**: dbInterface

**Class**: DB Manager

This is our **public** interface to the database used by our other project packages.

```
package dbInterface;

import java.util.List;
import java.sql.Connection;
import java.sql.SQLException;

import javax.annotation.PostConstruct;
import javax.sql.DataSource;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Repository;
import org.springframework.jdbc.core.support.JdbcDaoSupport;

import entities.Task;
import entities.TaskStatus;
import entities.User;
import logger.Logger;
```

```
@Repository
public class DbManager extends JdbcDaoSupport {

	@Autowired
	private DataSource dataSource;

	private Connection connection;
	private Object connectionLock;

	public DbManager() {
		super();
		this.connectionLock = new Object();
	}

	@PostConstruct
	private void init() {
		super.setDataSource(dataSource);

		try {
			dataSource.setLoginTimeout(15);   // 15 seconds
		} catch (Exception e) {
			try {
				// ZERO => use the (none zero!) default
				dataSource.setLoginTimeout(0);
			} catch (SQLException e1) {
				Logger.logError("DB error. Failed to set Login Timeout", e1);
			}
		}
		try {
			connection = DbConnector.makeConnection(dataSource);
		}catch (Exception e2) {
			Logger.logError("DB error. Failed DB connection" ,  e2);
		}

		Logger.logInfo("DB Manager initiaized");
	}

	public List<User> selectUsers() throws Exception {
		List<User> list;
		synchronized (connectionLock) {
			list = UserFetcher.selectUsers(connection);
		}
		return list;
	}
```

```
    public List<TaskStatus> selectTaskStatusSet() throws Exception {
        List<TaskStatus> set;
        synchronized (connectionLock) {
            set = TaskFetcher.selectTaskStatusSet(connection);
        }
        return set;
    }

    public List<Task> selectTasks() throws Exception {
        List<Task> list;
        synchronized (connectionLock) {
            list = TaskFetcher.selectTasks(connection);
        }
        return list;
    }

    public void insertTask(Task task) throws Exception {
        try {
            synchronized (connectionLock) {
                TaskInserter.insertTask(connection, task);
            }
        } catch (Exception E) {
            throw E;
        }
    }

    public void updateTask(TaskUpdateData taskUpdateData) throws
        Exception {
        try {
            synchronized (connectionLock) {
                TaskUpdater. updateTask (connection, taskUpdateData);
            }
        } catch (Exception E) {
            throw E;
        }
    }
}
```

## Package: web Control

MVC: we already have discussed the Model. We now move on to the Control.

How about the View? It is the React application that we develop a bit later that is responsible for *representing and styling and displaying* the data, i.e., the React manages the V!

This is the C in MVC.

Create the package named webC**ontrol.**

**Package**: webControl
**Class**: Entity Container

This is a wrapper class for an array of our entities. We use the variable name **data** which will end up being the entity array name in our JSON!

```
package webControl;
import java.util.List;

public class  EntityContainer <ENTITY>{

	private List<ENTITY> data;

	public EntityContainer() {

	}
	public EntityContainer(List<ENTITY> data) {
		this.data = data;
	}

	public List<ENTITY> getData() {
		return data;
	}

	public void setData(List<ENTITY> data) {
		this.data = data;
	}
}
```

**Package**: webControl
**Class**: Task Update Data

This class bundles two tasks that represent a task update, i.e., a 'before' and 'after' task.

```
package webControl;

import entities.Task;

public class TaskUpdateData {

	private Task taskBeforeUpdate;
	private Task taskAfterUpdate;

	public Task getTaskBeforeUpdate() {
		return taskBeforeUpdate;
	}

	public void setTaskBeforeUpdate(Task taskBeforeUpdate) {
		this.taskBeforeUpdate = taskBeforeUpdate;
	}

	public Task getTaskAfterUpdate() {
		return taskAfterUpdate;
	}

	public void setTaskAfterUpdate(Task taskAfterUpdate) {
		this.taskAfterUpdate = taskAfterUpdate;
	}
}
```

**Package**: webControl
**Class**: Web Controller

This class provides the following web services (URL endpoints) . We will evaluate these endpoints via Postman when our server is up and running.

1. GET http://localhost:8081/users/ping
2. GET http://localhost:8081/users/getUsers
3. GET http://localhost:8081/users/getTasks
4. GET http://localhost:8081/users/getTaskStatusSet
5. POST http://localhost:8081/users/postTask
6. PUT http://localhost:8081/users/putTask

```
package webControl;

import java.io.IOException;
import java.util.ArrayList;
import java.util.List;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.HttpStatus;
import org.springframework.http.MediaType;
import org.springframework.http.ResponseEntity;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.CrossOrigin;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.bind.annotation.ResponseBody;
import org.springframework.web.bind.annotation.ResponseStatus;

import com.fasterxml.jackson.core.JsonProcessingException;
import com.fasterxml.jackson.databind.ObjectMapper;

import dbInterface.DbManager;
import entities.Task;
import entities.TaskStatus;
import entities.User;
import logger.Logger;
```

```
@Controller
public class WebController {

	@Autowired
	private DbManager db;

	public WebController() {
		Logger.logInfo("Web Controller initiaized");
	}

	private static final  ObjectMapper mapper = new ObjectMapper();

	static String toText(Object obj, Class<?> clazz) throws Exception {
		try {
			String text = mapper.writeValueAsString(obj);
			return text;
		} catch (JsonProcessingException e) {
			throw new Exception
			("Failed to convert <" +clazz.getName()+ "> to Text");
		}
	}

	@RequestMapping  ( value = "/users/ping", method= RequestMethod.GET)
	@ResponseStatus ( value = HttpStatus.CREATED)
	public @ResponseBody String ping () throws IOException{
		Logger.logInfo("Web Control pinged");
		return "Status: OK";
	}
```

```
@CrossOrigin
@RequestMapping (value = "/users/getUsers",
method= RequestMethod.GET,
produces = MediaType.APPLICATION_JSON_VALUE)
public ResponseEntity<String> getUsers() throws Exception{
        Logger.logInfo("Web Control: Start Get Users");
        List<User> list;
        try {
                list = db.selectUsers();
        }catch (Exception e) {
                Logger.logError("ERROR geting Users",  e);
                list = new ArrayList<User>();
        }

        EntityContainer<User> users = new EntityContainer<User>(list);
        String responseStr = toText(users, users.getClass());
        ResponseEntity<String> response =
                new ResponseEntity<String>(responseStr, HttpStatus.CREATED);

        Logger.logInfo("Web Control: Get Users: " +list.size()+ " users");
        return response;
}

//    Note to test a delayed response try adding this
//
//          Logger.logInfo("Web Control: Get Tasks: " +list.size()+ " items");
//          Logger.logInfo("Web Control: Get Tasks FAKE 7 seconds Delay" );
//          Thread.sleep(7000);
//           return response;
```

```
@CrossOrigin
@RequestMapping (value = "/users/getTasks",
method= RequestMethod.GET,
produces = MediaType.APPLICATION_JSON_VALUE)
public ResponseEntity<String> getTasks() throws Exception{
        Logger.logInfo("Web Control: Start Get Tasks");
        List<Task> list;
        try {
                list = db.selectTasks();
        }catch (Exception e) {
                Logger.logError("ERROR geting Tasks",e );
                list = new ArrayList<Task>();
        }

        EntityContainer<Task> tasks = new EntityContainer<Task>(list);
        String responseStr = toText(tasks, tasks.getClass());
        ResponseEntity<String> response =
                new ResponseEntity<String>(responseStr, HttpStatus.CREATED);
        Logger.logInfo("Web Control: Get Tasks: " +list.size()+ " items");
        return response;
}

@CrossOrigin
@RequestMapping (value = "/users/getTaskStatusSet",
method= RequestMethod.GET,
produces = MediaType.APPLICATION_JSON_VALUE)
public ResponseEntity<String> getTaskStatusSet() throws Exception{
        Logger.logInfo("Web Control: Start Get Task Status Set");
        List<TaskStatus> list;
        try {
                list = db.selectTaskStatusSet();
        }catch (Exception e) {
                Logger.logError("ERROR geting Task Status Set",e );
                list = new ArrayList<TaskStatus>();
        }

        EntityContainer<TaskStatus> set =
           new EntityContainer<TaskStatus>(list);
        String responseStr = toText(set, set.getClass());
        ResponseEntity<String> response =
                new ResponseEntity<String>(responseStr, HttpStatus.CREATED);
        Logger.logInfo("Web Control: Get Task Status Set: " +list.size()+ " items");
        return response;
}
```

```
 class RESTAck {   //  POJO for Post Puts ack
        public RESTAck() {
        }
        public boolean isOk() {
                return ok;
        }
        public void setOk(boolean ok) {
                this.ok = ok;
        }
        public String getErrorMessage() {
                return errorMessage;
        }
        public void setErrorMessage(String errorMessage) {
                this.errorMessage = errorMessage;
        }
        private boolean ok;
        private String errorMessage=null;
}

@RequestMapping  ( value = "/users/postTask",
                method= RequestMethod.POST,
                produces = MediaType.APPLICATION_JSON_VALUE)
@CrossOrigin
public ResponseEntity<RESTAck> postTask(@RequestBody Task task)
        throws IOException{

        Logger.logInfo("Web Control: Post for " + task.toString());
        RESTAck ack = new RESTAck();
        HttpStatus status;
        try{
                db.insertTask(task);
                status = HttpStatus.CREATED;
                ack.setOk(true);

        } catch (Exception E) {
                status = HttpStatus.INTERNAL_SERVER_ERROR;
                ack.setOk(false);
                ack.setErrorMessage(E.getMessage());
                Logger.logError("Task Post failed", E);

        }
        ResponseEntity<RESTAck> response  =
             new ResponseEntity<RESTAck>(ack, status);
        return response;
}
```

```
	@RequestMapping  ( value = "/users/putTask",
			method= RequestMethod.PUT,
			produces = MediaType.APPLICATION_JSON_VALUE)
	@CrossOrigin
	public ResponseEntity<RESTAck> putTaskStatus(@RequestBody
		TaskUpdateData taskData) throws IOException{

		Logger.logInfo("Web Control: Task before update: " +
				taskData.getTaskBeforeUpdate().toString());

		RESTAck ack = new RESTAck();
		HttpStatus status;

		try{
			db.updateTask(taskData);
			status = HttpStatus.CREATED;
			ack.setOk(true);
			Logger.logInfo("Web Control: Task after update: " +
					taskData.getTaskAfterUpdate().toString());
		} catch (Exception E) {
			status = HttpStatus.INTERNAL_SERVER_ERROR;
			ack.setOk(false);
			ack.setErrorMessage(E.getMessage());
			Logger.logError("Task Update failed", E);

		}
		ResponseEntity<RESTAck> response  =
		    new ResponseEntity<RESTAck>(ack, status);
		return response;
	}

}
```

## Package: control

Create the package named **control.** This is our server 'main.'

**Package**: control

**Class**: Main Control

```
package control;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.annotation.ComponentScan;

import logger.Logger;

@ComponentScan (basePackages= {"webControl", "dbInterface"} )

@SpringBootApplication
public class MainControl {

	public static void main(String[] args) {
		Logger.logInfo("Server starting");
		try {
			SpringApplication.run(MainControl.class, args);
		} catch ( Exception e) {
			Logger.logError("Server failed to start", e);
		}
	}
}
```

Note: Spring Component Scan

The Spring related **Component Scan** annotation is in our main class. This tells Spring to look in the listed packages (and only there!) for Classes that '**qualify'** such that Spring instantiates these classes for us as singleton instances at startup (when **Spring Application** first starts).

Spring looks in the listed packages and sees the two annotations we used @Repository and @Controller. These annotations(and others that we do not use here) tell Spring that the classes **qualify** per the above.

Code snippet

```
@Repository
public class DbManager extends JdbcDaoSupport {

        @Autowired
        private DataSource dataSource;

        private Connection connection;
        private Object connectionLock;

        public DbManager() {
                super();
                this.connectionLock = new Object();
        }

        @PostConstruct
        private void init() {
            …
```

Code snippet

```
@Controller
public class WebController {

        @Autowired
        private DbManager db;

       ….
```

If there is a **Post Construct** function annotation in a **qualifying** class Spring invokes the annotated method just after calling the class constructor. The invocations for the constructor and the post construct method are executed as a single unit of work before Spring continues looking for more things to do!

Spring also sees the @**Autowired** annotation. Spring creates a singleton instance of the given class and assigns it to the variable DB (in the snippet just above). Spring may see multiple auto wired annotations in multiple ***classes*** for the same class (say Widget). Spring instantiates JUST one Widget (a singleton) and the ***classes*** share the single instance – Widget BETTER BE thread safe!!!!

## Our Data Source

In case you have not noticed there are no references in our Java project to our database information (look at our **Db Connector** and **Db Manager** classes!)

- DBMS Port Number
- Database Name
- Username
- User Password

Also, there is no reference to the port number we assign to our server.

Providing the above information is the 'just one more thing' that we take care of now. We create the project **folder** …/src/main/resources. In the project explorer right click on src

- New
- Folder
- Enter: main/resources

Go to the above **folder** in eclipse and create a new file named application.properties enter the following and save.

```
server.port=8081

spring.datasource.url=jdbc:mysql://localhost:3306/users?useSSL=false
spring.datasource.username=Smithy
spring.datasource.password=Smithy!878
```

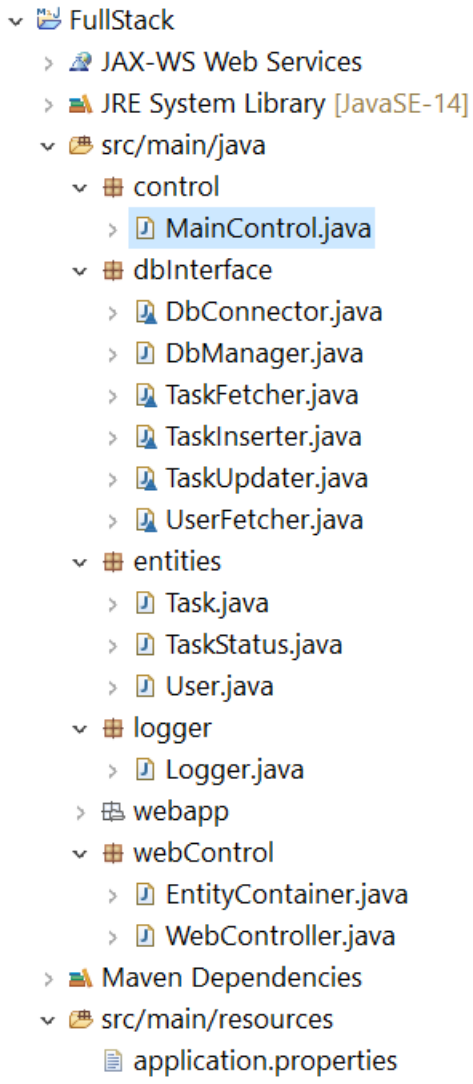
Note: **replace** *Smithy!878* with the password you picked!

1. We configured our server to run on port **8081**.
2. We provided data for our java Data Source to locate and connect to our DB as user Smithy

In our DB Manager class Spring 'sees' the following **Auto wired Data Source** reference and creates an instance for us using our application.properties file

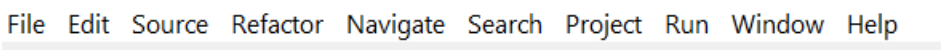
```
@Autowired
    private DataSource dataSource;
```

This is my Project Explorer:



## Starting and Stopping the server

Make sure your application compiles clean!. Click on **Project** and pick **Clean;** you should see no issues in the Problems Tab.




Set the eclipse perspective to Debug. Right click on **Window**

- Perspective
- Open Perspective
- Pick: Debug (it may be under Other)

You should see your **Project Explorer** and a **Debug** tab on the left.

You may need to refresh eclipse.

- Perspective
- Reset Perspective

Right on the **MainControl.java** class in the Project Explorer; pick one of these two

- Debug as a Java Application
- Run as a Java Application

You should see a bunch of logging to the console. Some of the log entries are yours!

Click on the Debug tab next to the Project Explorer; you should see a bunch of java Threads and more if you picked debug

Possible issues:

1. Database issues:
   Make sure your DB sever is up. Try connecting to DB via a DOS Command prompt. Make sure your DB information is correct in application.properties file. Check your pom.file property for MySQL.

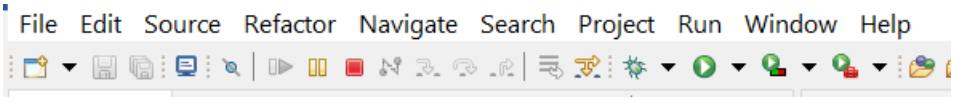
2. If you see an error message on the console like: Web server failed to start: Port 8081 was already in use.

Action: Identify and stop the process that is listening on port 8081 or configure our application (via application.properties file) to listen on another port.

3. Put on your debug face and find those insects and/or invite a Java friend over for Pizza.

You can stop the server by:

Clicking the red square on the Eclipse Tool bar on top




## Exercise the java server via Postman

We evaluate using Postman before trying to integrate with our React application which we have not written yet!

### Postman Install

https://learning.postman.com/docs/getting-started/installation-and-updates/

### GET

Bring up Postman and try these 'GETs.' Compare the following URLs with the java class Web Controller

- http://localhost:8081/users/ping
- http://localhost:8081/users/getUsers
- http://localhost:8081/users/getTasks
- http://localhost:8081/users/getTaskStatusSet

Make sure the above GETs are working. If not, check the Postman results and your Java server log for clues! Make sure your Java server is running.

## POST

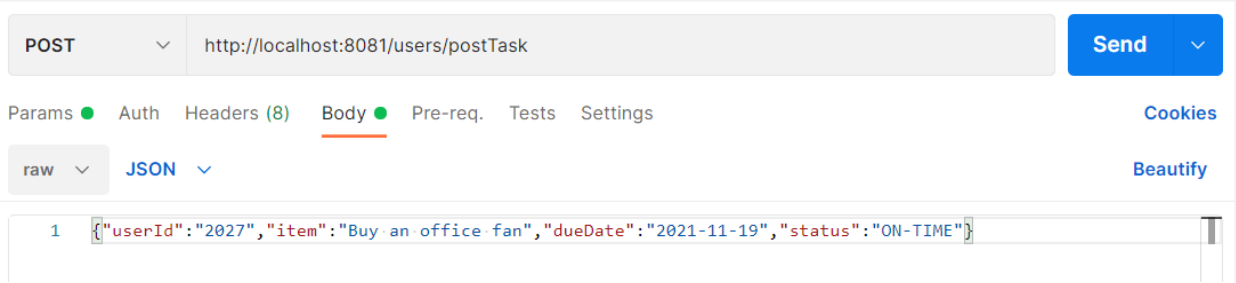
Now, try adding a new Task via postman. First connect to your database and type use users;
select * from task order by user_id;

In postman enter the url: http://localhost:8081/users/postTask

Pick POST. Click on Body and select JSON -JSON is the format for the Task we are about to POST – M. FullStack, please note our server expects JSON!

In the body area enter the task as JSON, i.e., type this valid JSON

```
{"userId":"2027","item":"Buy an office fan","dueDate":"2021-11-19","status":"ON-TIME"}
```




Click on Send. You should see the server's response Ack (see java class Web Controller) indicating a successful Post.

```
"ok": true,
"errorMessage": null
```

Now try your data base select and you should see the new task

select * from task order by user_id;

Error cases:

In postman try numerous 'error cases' (where the POST fails). Check the server response in Postman and your java server logs

Try Posting this:

```
{"userId":"2027","item":"Buy an office sink","dueDate":"2021-11-19","status":"NOT_ON-TIME"}
```

Make sure the server 'Nacks' the POST. If you do not like the related logs or error messages then change them.

## PUT

Our java server also provides us with a PUT to update a task .

First, recall our Java class:

```
public class TaskUpdateData {

        private Task taskBeforeUpdate;
        private Task taskAfterUpdate;
        ...
```

Back to postman. We will update an existing task in our data base data .

Since we already successfully evaluated the following URL, we use it to pick a task to update:

In postman pick GET, enter the following URL and Send

http://localhost:8081/users/getTasks

Pick any task and make a copy. I picked this task

```
{
    "userId": "0001",
    "item": "React Training 101",
    "dueDate": "2021-08-28",
    "status": "ON-TIME"
}
```

In postman pick PUT and enter the following URL

http://localhost:8081/users/putTask

Prepare the body for an update. If you have issues created test JSON body here is a 'trick.'

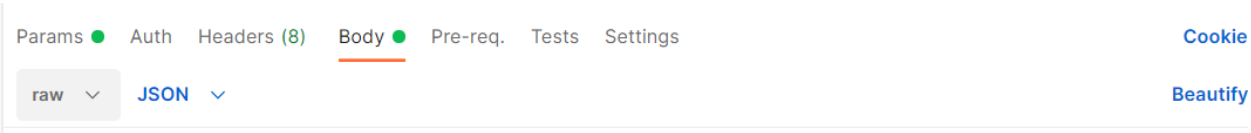
In our Web Controller, we have the following:

```
private final static ObjectMapper mapper = new ObjectMapper();

private static String toText(Object obj, Class<?> clazz) throws Exception {
	try {
		String text = mapper.writeValueAsString(obj);
		return text;
	} catch (JsonProcessingException e) {
		throw new Exception
		 ("Failed to convert <" +clazz.getName()+ "> to Text");
	}
}
```

Try something like this if JSON is an issue

```
package webControl;

import entities.Task;

public class JsonForTaskUpdate {

	public static void main (String[] args) {
		Task t1 = new Task();
		t1.setUserId("0001");
		t1.setItem("React Training 101");
		t1.setStatus("PENDING");
		t1.setDueDate("2021-08-28");

		Task t2 = new Task();
		t2.setUserId("0001");
		t2.setItem("React Training 101");
		t2.setStatus("COMPLETE");
		t2.setDueDate("2021-08-08");

		TaskUpdateData data = new TaskUpdateData();
		data.setTaskBeforeUpdate(t1);
		data.setTaskAfterUpdate(t2);


		try {
			String json = WebController.toText(data, data.getClass());
			System.out.println(json);
		} catch (Exception e) {
			System.out.println("ERROR: " + e);
		}

	}
}
```

Grab the **output** and paste it into Postman for your PUT body. You can Beautify it if you wish.




The body

```
{
    "taskBeforeUpdate": {
        "userId": "0001",
        "item": "React Training 101",
        "dueDate": "2021-08-28",
         "status": "ON-TIME"
    },
    "taskAfterUpdate": {
        "userId": "0001",
        "item": "React Training 101",
        "dueDate": "2021-08-08",
        "status": "COMPLETE"
    }
}
```

Send the data and verify that the server ack'ed our PUT.

Try the following GET to see our updated task. If you are a person that needs more convincing also look at the database!

http://localhost:8081/users/getTasks

# The REACT application

## Overview

Here is the plan:

The App - our topmost - component

This component renders differently via state changes. The changes correspond to these three scenarios

1. Waiting for remote data to load  (initial render)
2. Remote data failed to arrive
3. Remote data arrives successfully

The App component includes a *lookup function* for a single user's task list.

lookup(user id) returns an array of *tasks* owned by the user

The App provides access to the *lookup function* to the Users Page component via props.

**Users Page component**

The *users page* displays a table of all users. A *click* on a row of the table causes the user's *Tasks* to be displayed for the (one) *selected* user.

**Tasks Page component**

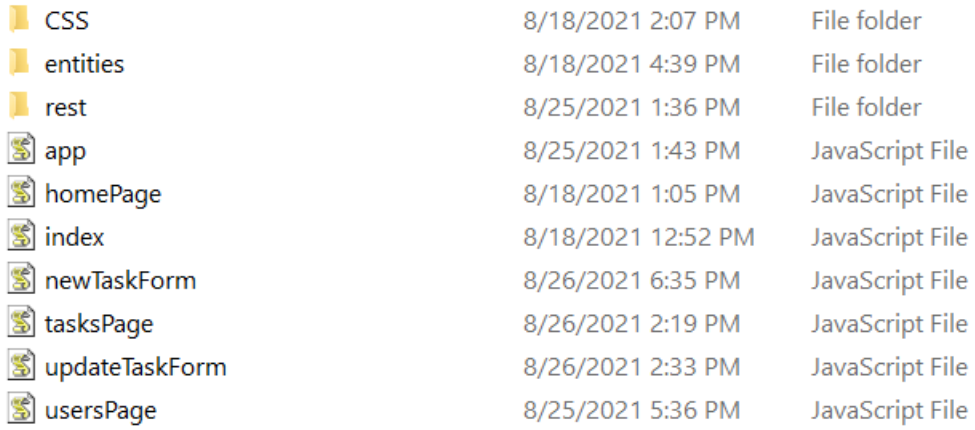
This page provides the ability to:

1. Add a new task assigned to the user.
2. Update the status of a given task.

**Home page component**

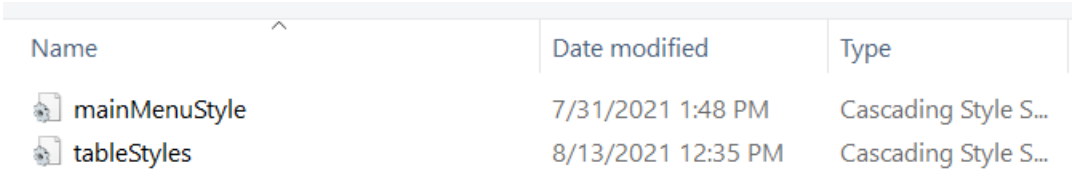
This component displays hard-coded information about our 'fake' company!

The src folder for the project:



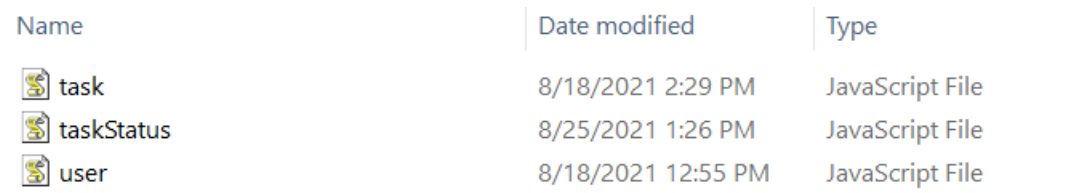
| | | |
|---|---|---|
| CSS | 8/18/2021 2:07 PM | File folder |
| entities | 8/18/2021 4:39 PM | File folder |
| rest | 8/25/2021 1:36 PM | File folder |
| app | 8/25/2021 1:43 PM | JavaScript File |
| homePage | 8/18/2021 1:05 PM | JavaScript File |
| index | 8/18/2021 12:52 PM | JavaScript File |
| newTaskForm | 8/26/2021 6:35 PM | JavaScript File |
| tasksPage | 8/26/2021 2:19 PM | JavaScript File |
| updateTaskForm | 8/26/2021 2:33 PM | JavaScript File |
| usersPage | 8/25/2021 5:36 PM | JavaScript File |

The CSS folder:



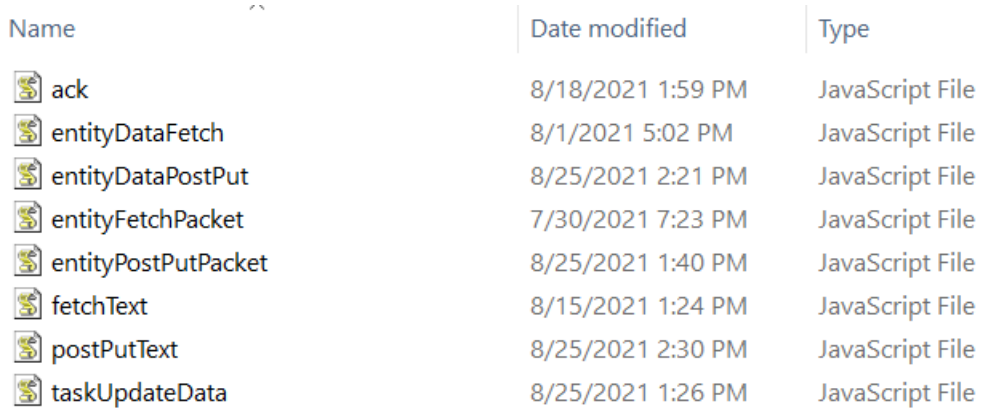
| Name | Date modified | Type |
|---|---|---|
| mainMenuStyle | 7/31/2021 1:48 PM | Cascading Style S... |
| tableStyles | 8/13/2021 12:35 PM | Cascading Style S... |

The entities folder conations our two entities and a task status class



| Name | Date modified | Type |
|---|---|---|
| task | 8/18/2021 2:29 PM | JavaScript File |
| taskStatus | 8/25/2021 1:26 PM | JavaScript File |
| user | 8/18/2021 12:55 PM | JavaScript File |

The rest folder



| Name | Date modified | Type |
|---|---|---|
| ack | 8/18/2021 1:59 PM | JavaScript File |
| entityDataFetch | 8/1/2021 5:02 PM | JavaScript File |
| entityDataPostPut | 8/25/2021 2:21 PM | JavaScript File |
| entityFetchPacket | 7/30/2021 7:23 PM | JavaScript File |
| entityPostPutPacket | 8/25/2021 1:40 PM | JavaScript File |
| fetchText | 8/15/2021 1:24 PM | JavaScript File |
| postPutText | 8/25/2021 2:30 PM | JavaScript File |
| taskUpdateData | 8/25/2021 1:26 PM | JavaScript File |

# React Code

## The entities folder

Compare the following classes to the corresponding Java entities package.

### Class: Task

```
export class Task {
    userId = null;
    item = null;
    dueDate = null;
    status = null;

     constructor(task){
        if (task){
            this.userId  = task.userId;
            this.item    = task.item;
            this.dueDate = task.dueDate;
            this.status  = task.status;
        }
    }

    toString(){
        return "Task. User: " + this.userId +
                " Item: " + this.item +
                 " due date: " + this.dueDate +
                " status: "  + this.status;
    }
}
```

## Class: User

```
export class User {

    id = null;
    name = null;
    email = null;

    constructor(user){
        if (user){
            this.id = user.id;
            this.name = user.name;
            this.email = user.email;
        }
    }
}
```

## Class: Task Status

Compare to our corresponding DB table and Java class

```
export class TaskStatus {

    status = null;
    isDefault = null;

    constructor(taskStatus){
        if (taskStatus){
            this.status  = taskStatus.status;
            this.isDefault = taskStatus.isDefault;
        }
    }
}
```

## The rest folder

This folder contains our REST (we use: GET, PUT, POST) related classes.

### Class: Ack

An 'ack' or 'nack.' Compare to the corresponding Java class

```
export class Ack {
    ok = false;
    errorMessage = null;

    constructor(j){
        if (j){
            this.ok = j.ok;
            this.errorMessage = j.errorMessage;
        }
    }

     toString(){
        var str;
        if (this.ok){
            str = "Status: Success";
        }else{
            str = "Status: Error  Issue: " + this.errorMessage;
        }
        return str;
    }
}
```

### Class: Entity Fetch Packet

This class replaces what would be SIX separate arguments for the Entity Data Fetch class constructor. We try to keep things generic; entity is either a User or a Task

```
export class EntityFetchPacket{
    url;
    jsonKey;
    entityArr;
    entityClass;
    errorHandler;
    dataArrivedHandler;
}
```

## Class: Entity Data Fetch

```
import {fetchText} from './fetchText.js';

export class EntityDataFetch {

   constructor(fetchPacket){
      this.packet = fetchPacket;
   }

   errorHandler(e){
      this.packet.errorHandler(e);
   }

   fetchData(){
      fetchText( this.packet.url, this);
   }

   textHandler (responseText){
      let responseObj = JSON.parse(responseText);
      let rows = responseObj[this.packet.jsonKey];
      let entityClass = this.packet.entityClass;

      for (var i = 0; i < rows.length; i++) {
         let entity = new entityClass(rows[i]);
         this.packet.entityArr.push(entity);
      }

      // our entity array is all loaded, so...
      this.packet.dataLoadedHandler();
   }
}
```

## Function: Fetch Text

```
export function fetchText(url, entityDataFetch) {

    get(url).then (
        (responseText) => {
            entityDataFetch.textHandler(responseText);
        } ).catch((e) => {
            entityDataFetch.errorHandler(e);
        })
}

async function  get (url)  {
    let responsePromise = await fetch(url);
    if (!responsePromise.ok){
        let issue = 'ERROR: HTTP issue. Status: ' + responsePromise.status;
        throw new Error(issue);
    }

    let retText = await responsePromise.text();
    return retText;
}
```

## Class: Entity Post Put Packet

This class replaces what would be FIVE separate arguments for the **Entity Data Post Put** class constructor. We try to keep things generic; entity is either a User or a Task. The method is either POST or PUT

```
export class EntityPostPutPacket{
    url;
    method;
    dataToSend;
    errorHandler;
    ackHandler;
}
```

## Class: Entity Post Put

The method is either POST or PUT

```
import {postPutText} from './postPutText.js';
import {Ack} from './ack.js';

export class EntityDataPostPut {

   constructor(packet){
      this.packet = packet;
   }

   getMethod(){
      return this.packet.method;
   }

   getDataToSend(){
         return this.packet.dataToSend;
   }

   getURL(){
      return this.packet.url;
   }

   errorHandler(e){
      this.packet.errorHandler(e);
   }

   sendData(){
      postPutText(this);
   }

   textHandler (responseText){
     let ackObj = JSON.parse(responseText);
     let ack = new Ack(ackObj);
     this.packet.ackHandler(ack);
   }
}
```

## Function: Post Put Text

```
export function postPutText(entityDataPostPut) {

    sendData(entityDataPostPut.getMethod(),
                entityDataPostPut.getURL(),
                entityDataPostPut.getDataToSend()).then (
        (responseText) => {
            entityDataPostPut.textHandler(responseText);
        }
        ).catch((e) => {
            entityDataPostPut.errorHandler(e);
        })
}

async function  sendData (method, url , data)  {

    const bodyText = JSON.stringify(data);

    let responsePromise = await fetch(url, {
        method: method,
        mode: 'cors',
        body: bodyText,
        headers: {"Content-type": "application/json"},
    });


    if (!responsePromise.ok){
        let issue = 'ERROR: HTTP issue. ' + responsePromise.status;
        throw new Error(issue);
    }

    let retText = await responsePromise.text();
    return retText;
}
```

## Class: Task Update Data

Compare this class with the corresponding Java class.

```
export class TaskUpdateData {
          taskBeforeUpdate=null;
          taskAfterUpdate=null
}
```

## CSS Folder

### File: Main Menu Style

```
.menu {
background-color: lightgray;
list-style-type:none;
padding: 4px;
margin:3px;
width:28%;
}

.menuItem {
    text-decoration: none;
    display: inline-block;
    padding: 8px;
}

.active {
    background-color:lightsalmon;
}

.blink_me {
    animation: blinker 1s linear infinite;
  }

  @keyframes blinker {
    50% {
      opacity: 0;
    }
  }
```

## File: Table Styles

```
.rowStyle {
                border: 1px solid black;
                border-collapse: collapse;
                background-color:lightgrey;
                text-align: left;
                color: black;
}
.trVanillaRow {
        color: black;
        font-style: normal;
        background-color:lightgrey;
}

.trSelectedRow {
        color: green;
        font-style: normal;
}
```

# Folder: src

## File: index.js

```
import React from 'react';
import ReactDOM from 'react-dom';

import {App} from './app.js' ;

ReactDOM.render(
   <App></App>,
   document.querySelector('#root')
);
```

## Component: App

This component perform REST GETs for our users, tasks and set of task status values. Earlier, we evaluated the Java server logic for these via Postman

- http://localhost:8081/users/getUsers
- http://localhost:8081/users/getTasks
- http://localhost:8081/users/getTaskStatusSet

```
import React from 'react';
import {Route, NavLink, HashRouter} from "react-router-dom";
import {HomePage} from "./homePage.js"
import {UsersPage} from "./usersPage.js"
import "./CSS/mainMenuStyle.css"
import {EntityDataFetch} from './rest/entityDataFetch.js';
import {User} from "./entities/user.js"
import {Task} from "./entities/task.js"
import {TaskStatus} from "./entities/taskStatus.js"
import {EntityFetchPacket} from './rest/entityFetchPacket.js';

export class App extends React.Component{

   usersUrl ='http://localhost:8081/users/getUsers';
   usersJsonKey = 'data';

   tasksUrl ='http://localhost:8081/users/getTasks';
   tasksJsonKey = 'data';

   taskStatusSetUrl ='http://localhost:8081/users/getTaskStatusSet';
   taskStatusSetJsonKey = 'data';

   constructor(props){
      super(props);
      this.userArr = [];
      this.taskArr = [];
      this.taskStatusSet= [];

      this.state = {dataLoadStatus: 'PENDING'};
      this.dataLoadedHandler = this.dataLoadedHandler.bind(this);

      this.fetchIssue = null;
      this.handleFetchException = this.handleFetchException.bind(this);

      this.dataLoadingRender = this.dataLoadingRender.bind(this);
```

```
    this.dataRender = this.dataRender.bind(this);
    this.taskLookUp = this.taskLookUp.bind(this);
    this.dataLoadedCount = 0;
}

taskLookUp(userId){
    let list = this.taskArr.filter((task) => {return task.userId === userId});
    return list;
}

dataLoadedHandler(){
    this.dataLoadedCount ++;
    if (this.dataLoadedCount === 3){
        this.setState({dataLoadStatus : 'COMPLETE'});
    }
}

handleFetchException(e){
    this.fetchIssue = e;
    this.setState({dataLoadStatus : 'ERROR'});
}

componentDidMount(){

   const  usersPacket = new EntityFetchPacket();
   usersPacket.url = this.usersUrl;
   usersPacket.jsonKey = this.usersJsonKey;
   usersPacket.entityArr = this.userArr;
   usersPacket.entityClass = User;
   usersPacket.errorHandler = this.handleFetchException;
   usersPacket.dataLoadedHandler = this.dataLoadedHandler;

   const userDataFetch = new EntityDataFetch(usersPacket);
   userDataFetch.fetchData();

   const  taskStatusSetPacket = new EntityFetchPacket();
   taskStatusSetPacket.url = this.taskStatusSetUrl;
   taskStatusSetPacket.jsonKey = this.taskStatusSetJsonKey;
   taskStatusSetPacket.entityArr = this.taskStatusSet;
   taskStatusSetPacket.entityClass = TaskStatus;
   taskStatusSetPacket.errorHandler = this.handleFetchException;
   taskStatusSetPacket.dataLoadedHandler = this.dataLoadedHandler;

   const taskStatusSetFetch = new EntityDataFetch(taskStatusSetPacket);
   taskStatusSetFetch.fetchData();
```

```
  const  tasksPacket = new EntityFetchPacket();
  tasksPacket.url = this.tasksUrl;
  tasksPacket.jsonKey = this.tasksJsonKey;
  tasksPacket.entityArr = this.taskArr;
  tasksPacket.entityClass =Task;
  tasksPacket.errorHandler = this.handleFetchException;
  tasksPacket.dataLoadedHandler = this.dataLoadedHandler;

  const tasksDataFetch = new EntityDataFetch(tasksPacket);
  tasksDataFetch.fetchData();
}

render(){
   let JSX;

   if (this.state.dataLoadStatus === 'COMPLETE'){
      JSX = this.dataRender()
   } else if (this.state.dataLoadStatus === 'PENDING'){
      JSX = this.dataLoadingRender()
   } else {
      JSX = this.errorRender()
   }

   return JSX;
}

dataLoadingRender(){
  return (
  <div>
  <h2>Welcome to Cold Beans for lunch Ink</h2>
  <p className="blink_me">Please be patient.  We are initilaizing data...</p>
  <HomePage></HomePage>
  </div> );
}

errorRender(){
   let errorMsg = 'Loading Data Error occurred Call IT now! Provide the msg: ' +
                     this.fetchIssue;
   return <div>{errorMsg}</div>
}
```

```
    dataRender(){

        let users = this.userArr;
        var JSX =
        <HashRouter>
        <div>
            <h2>Welcome to Cold Beans for lunch Ink</h2>
            <div>
                <ul>
                <li className="menuItem"><NavLink exact to= "/">Home</NavLink></li>
                <li className="menuItem"><NavLink to= "/users">Users</NavLink></li>
                </ul>
            </div>

            <div>
                <Route exact path= "/"  component= {HomePage}></Route>
                <Route path= "/users" render={(props) =>
                <UsersPage {...props}   users={users}
                                taskStatusSet = {this.taskStatusSet}
                                taskLookUp={this.taskLookUp}/>}>
                </Route>
            </div>
        </div>
        </HashRouter>

        return JSX;
    };
};
```

## Component: Home Page

```
import React from 'react';
export class HomePage extends React.Component {

   constructor(props){
      super(props);
      this.buildList = this.buildList.bind(this);
   }

   buildList(){
      let I = 0;
      let list = [];
      list.push(<li key={I++}>CEO: Rene Popcorn Callaway</li>);
      list.push(<li key={I++}>VP: Jose Callaway</li>);
      list.push(<li key={I++}>Tester: Wendy Max</li>);
      list.push(<li key={I++}>Lead Designer: SuSam Sam</li>);
      list.push(<li key={I++}>Junior Programmer: Lee Adams</li>);
      list.push(<li key={I++}>Office Manager: JJ Brown</li>);
      return list;
   }

   shouldComponentUpdate(){  // never!
      return false;
   }

   render(){
      return (
         <div>
            <h2>Welcome to Cold Cuts and Paste Ink</h2>
            <p>Employee list</p>
            <ul>
            {this.buildList()}
            </ul>
         </div>
      );
   }
}
```

## Component Users Page

This component displays a HTML table of all users. Most of the logic is in the click handler function; when a row is clicked we display the selected user's tasks via the Task Page component. The user array contains the complete set of users. The state attributes username and userId refer to the selected user. A state change forces a re-render (new task list) for the newly selected user.

```
import React from 'react'
import "./CSS/tableStyles.css"
import {TasksPage} from "./tasksPage.js"

export class UsersPage extends React.Component {

   constructor(props){
     super(props);
     this.buildHdr = this.buildHdr.bind(this);
     this.buildRows = this.buildRows.bind(this);
     this.clickHandler = this.clickHandler.bind(this);

     this.state = {taskArr : [], userName : null, userId : null };
     this.userIdCol = 1;
     this.userNameCol = 2;
     this.selectedRow = null;
   }
```

```
clickHandler(evt){
  evt.preventDefault();
  let row = evt.target.parentElement;
  if (row.rowIndex === 0){
    return;
  }

  if ( this.selectedRow){
    this.selectedRow.className='trVanillaRow';
  }

  this.selectedRow = row;
  row.className = 'trSelectedRow';

  let cell= row.cells[this.userNameCol];
  let userName = cell.innerText;

  cell = row.cells[this.userIdCol];
  let userId = cell.innerText;

  let tasks = this.props.taskLookUp(userId);
  if (tasks && tasks.length > 0){
    this.setState({taskArr : tasks,  userName : userName, userId : userId});
  } else {
    this.setState({taskArr : [], userName : userName, userId : userId});
  }
 }

 buildRows(){
    let I = 0;
    const list = this.props.users.map(u =>
    {
      let oneRow = <tr className='rowStyle' key = {I} >
          <td>{++I}</td><td>{u.id}</td><td>{u.name}</td>
          <td>{u.email}</td>
          </tr>

      return oneRow;
    });

    return list;
 }
```

```
  buildHdr(){
    let JSX =
      <tr className='rowStyle'>
        <th></th><th>ID</th><th>Name</th><th>Email</th>
      </tr>
    return JSX;
  }

  render(){
    let JSX =

    <div style= {{height: "410px", width: "1300px" , display:"flex", flexDirection:"row"}}>
      <div id="userTableID" style = {{overflow: "auto"}}>
        <table  border = '1|1' >
          <tbody  onClick={this.clickHandler}>
            {this.buildHdr()}
            {this.buildRows()}
          </tbody>
        </table>
      </div>

      <div id="tasksTableID" style = {{overflow: "auto", flexDirection:"row"}}>
        {this.selectedRow !== null ?
        <TasksPage
          user= {this.state.userName}
          userId= {this.state.userId}
          tasks= {this.state.taskArr}
          taskStatusSet = {this.props.taskStatusSet}>
        </TasksPage> : null}

      </div>

    </div>

    return JSX;
  }

}
```

## Component: Tasks Page

This component displays the list of tasks for a single user. It is a child of the Users component. There are two forms (one of the two is always visible). The list is sorted by due date.

One form allows our web-user to add a new task to the user's task list.

One form allows our web-user to update an existing user's task list. Updates are limited to a task's due date and/or the status.

The parent component (Users Page) forces *this component* to re-render when our web-user clicks on a different user. *This component* also forces re-renders based on state changes. Since we persist and use data regarding the select task, we must clear that data at the very top of *our* render function, so our component includes this function

```
adjustInstanceDataBeforeRender() {

  if (this.selection.row && this.selection.task.userId !== this.props.userId){
    this.selection.row = null;
    this.selection.task = null;
  }
}
```

The full component

```
import React from 'react';
import "./CSS/tableStyles.css"
import {NewTaskForm} from './newTaskForm.js';
import {UpdateTaskForm} from './updateTaskForm.js';

export class TasksPage extends React.Component {
  constructor(props){
    super(props);

    this.buildHdr = this.buildHdr.bind(this);
    this.buildRows = this.buildRows.bind(this);
    this.adjustInstanceDataBeforeRender =
        this.adjustInstanceDataBeforeRender.bind(this);
    this.buildTable = this.buildTable.bind(this);
```

```
    this.clickHandler = this.clickHandler.bind(this);

    this.state = {reRenderToggle:1}

    // User can select (click-on) a table row
    this.selection = {row : null, task: null};
  }

   buildRows(){

    // first sort

    this.props.tasks.sort((t1,t2) => {
      let x = 0;
      if (t1.dueDate < t2.dueDate){
        x=-1;
      } else if (t1.dueDate > t2.dueDate){
         x=+1;
      }
      return x;
    } );

    let I = 0;
    const list = this.props.tasks.map(t => {
          let oneRow =
               <tr className='rowStyle' key = {I++}>
               <td>{t.item}</td><td>{t.dueDate}</td><td>{t.status}</td>
               </tr>

          return oneRow;
        });

      return list;
    }

    buildHdr(){

      let JSX =
        <tr className='rowStyle'>
        <th>Item</th><th>Due</th><th>Status</th>
        </tr>

       return JSX;
  }
```

```
buildTable(){
  if (this.props.tasks && this.props.tasks.length>0){
    return (
      <div>
       <>     User: {this.props.user} assigned tasks</>
      <div id="usersTableID" style = {{overflow: "auto",
                height: "224px" , marginRight: "50px" }}>

        <table style = {{  margin:"25px", border: '1|1',  width: "590px"}}>
          <tbody onClick={this.clickHandler}>
            {this.buildHdr()}
            {this.buildRows()}
          </tbody>
        </table>
      </div>
      </div> )
  } else if (this.props.user) {
    return (
      <p>      User: {this.props.user} has no assigned tasks
              
      </p>
    )
  }
}

clickHandler(evt){

    evt.preventDefault();

    let selectedRow = evt.target.parentElement;
    if (selectedRow.rowIndex <= 0) {
      return;
    }

    // previous selection
    if ( this.selection.row){
      this.selection.row.className='trVanillaRow';
    }

    selectedRow.className = 'trSelectedRow';

    //Table headers occupy row ZERO - our task array has no header so subtract ONE
    let selectedTask = this.props.tasks[selectedRow.rowIndex-1];
```

```
    this.selection.row = selectedRow;
    this.selection.task = selectedTask;
    this.setState({reRenderToggle : -this.state.reRenderToggle});
  }

  adjustInstanceDataBeforeRender() {

    if (this.selection.row && this.selection.task.userId !== this.props.userId){
      this.selection.row = null;
      this.selection.task = null;
    }
  }

  render() {
    this.adjustInstanceDataBeforeRender();

    let tableJSX = this.buildTable();

    return (
      <div>
        {tableJSX}
        {this.selection.task === null ?
          <NewTaskForm formId="formID"
            userId={this.props.userId}
            taskStatusSet = {this.props.taskStatusSet}>
          </NewTaskForm>  :

          <UpdateTaskForm
            task ={this.selection.task}
            taskStatusSet = {this.props.taskStatusSet}>
          </UpdateTaskForm>
        }
      </div>);
  }

}
```

## Component: New Task Form

This component allows our web-user to create a new task. The form submit logic sends a corresponding POST to our server. Earlier, we tested the Java server logic for the POST via Postman

http://localhost:8081/users/postTask

```
import React from 'react';
import {Task} from "./entities/task.js"
import {EntityDataPostPut} from './rest/entityDataPostPut.js';
import {EntityPostPutPacket} from './rest/entityPostPutPacket.js';

export const NewTaskForm = (props) => {

   const findDefaultOption= () => {
      let dflt = null;
      for (let i = 0; i < props.taskStatusSet.length; i++) {
         if (props.taskStatusSet[i].isDefault){
            dflt = props.taskStatusSet[i].status;
            break;
         }
      }

      if (!dflt && props.taskStatusSet.length > 0){
         dflt = props.taskStatusSet[0].status;  // first in array!
      }
      return dflt;
   }

   let dfltStatus=findDefaultOption();
   let item=null;
   let dueDate=null;
   let status=dfltStatus;
   let myForm=null;

   const postAckHandler = (ack) => {
      console.log("ACK -> " + ack.toString());
      if (ack.ok){
        alert("Task creation was successful");
      } else {
        alert("Task creation failed: " + ack.errorMessage);
      }
    }
```

```
const postErrorHandler= (e) => {
    console.log("ERROR -> " + e);
    alert("Task creation failed: " + e);
}

const itemHandler = (event) => {
    item = event.target.value;
}

const dueDateHandler = (event) => {
    dueDate = event.target.value;
}

const statusHandler = (event) => {
    status = event.target.value;
}

const submitHandler = (event) => {

    event.preventDefault();

    if ( item === null || item.length === 0 || dueDate === null || status === null ){
        alert("Please enter all Task related data");
        return;
    }
    let task = new Task(null);
        task.userId = props.userId;
        task.item  = item;
        task.dueDate = dueDate;
        task.status = status;

    let postPacket = new  EntityPostPutPacket();
        postPacket.method = 'POST';
        postPacket.url = 'http://localhost:8081/users/postTask';
        postPacket.dataToSend = task;
        postPacket.errorHandler = postErrorHandler;
        postPacket.ackHandler = postAckHandler;

    let entityPost = new EntityDataPostPut(postPacket);
    entityPost.sendData();

    myForm.reset();
    status=dfltStatus;
}
```

```
    const formCapture= (theForm) => {
        myForm = theForm;
        if (myForm) {
            myForm.reset();
        }
    }

    const buildStatusOptions= () => {
        let I = 0;
        const jsxArr =  props.taskStatusSet.map((taskStatus) => {
              return (<option key={I++}
value={taskStatus.status}>{taskStatus.status}</option>) });
        return jsxArr;
    }

    return (
        <form style = {{margin:"15px"}} onSubmit={submitHandler} ref={formCapture}>
        <p >Enter a new Task and SUBMIT</p>

        <div style= {{display:"flex", flexDirection:"row"}}>
            <label style = {{ width: "93px"}}>Task</label>
            <input
            type='text'
            onChange={itemHandler}/>
        </div>

        <div style= {{display:"flex", flexDirection:"row"}}>
            <label style = {{ width: "93px"}}>Due Date</label>
            <input
            type='date'
            onChange={dueDateHandler}/>
        </div>

        <div style= {{display:"flex", flexDirection:"row"}}>
            <label style = {{ width: "93px"}}>Status</label>
            <select  onChange={statusHandler} defaultValue={dfltStatus}>
            {buildStatusOptions()}
            </select>
        </div>

        <input type='submit'/>

        </form> );

}
```

Component: Update Task Form

This component allows our web-user to update a task. The form submit logic sends a corresponding PUT to our server. Earlier, we tested the Java server logic for the PUT via Postman

http://localhost:8081/users/putTask

A task update is limited to the due date and/or the status.

```
import React from 'react';
import {Task} from "./entities/task.js"
import {EntityDataPostPut} from './rest/entityDataPostPut.js';
import {EntityPostPutPacket} from './rest/entityPostPutPacket.js';
import {TaskUpdateData} from './rest/taskUpdateData.js';

export const UpdateTaskForm = (props) => {
   console.log("Child top of function ...." + props.task.toString() );

   let myForm=null;
   let newDueDate = props.task.dueDate;
   let newStatus = props.task.status;

   const postAckHandler = (ack) => {
     console.log("ACK -> " + ack.toString());
     if (ack.ok){
       alert("Task update was successful");
     } else {
       alert("Task update failed: " + ack.errorMessage);
     }
   }

   const postErrorHandler= (e) => {
      console.log("ERROR -> " + e);
      alert("Task update failed: " + e);
   }

   const dueDateHandler = (event) => {
      newDueDate = event.target.value;
   }

   const statusHandler = (event) => {
      newStatus = event.target.value;
   }
```

```
const submitHandler = (event) => {

    event.preventDefault();

    if ( newDueDate === null || newStatus === null ){
        alert("Please enter all Task related data");
        return;
    }
    let taskToUpdate = props.task;

    if ( taskToUpdate.dueDate ===  newDueDate &&
         taskToUpdate.status === newStatus){
        alert("Task related data has not changed");
        return;
    }

    let updatedTask = new Task();
        // We do not allow these two to change
        updatedTask.userId = taskToUpdate.userId;
        updatedTask.item   = taskToUpdate.item;
        // date or status can change
        updatedTask.dueDate = newDueDate;
        updatedTask.status = newStatus;

    let putPacket = new  EntityPostPutPacket();
        putPacket.method = 'PUT';
        putPacket.url = 'http://localhost:8081/users/putTask';
        let updateData = new TaskUpdateData();
        updateData.taskBeforeUpdate = taskToUpdate;
        updateData.taskAfterUpdate = updatedTask;
        putPacket.dataToSend = updateData;
        putPacket.errorHandler = postErrorHandler;
        putPacket.ackHandler  = postAckHandler;

    let entityPut = new EntityDataPostPut(putPacket);
    entityPut.sendData();
    myForm.reset();
   }

const formCapture= (theForm) => {
    myForm = theForm;
    if (myForm) myForm.reset();
}
```

```
    const buildStatusOptions= () => {
        let I = 0;
        const jsxArr =  props.taskStatusSet.map((taskStatus) => {
              return (<option key={I++}
                          value={taskStatus.status}>{taskStatus.status}</option>)
        });
        return jsxArr;
    }

    const buildJSX = () =>{
        let JSX =
            <form style = {{margin:"15px"}} onSubmit={submitHandler} ref={formCapture}>
            <p>Update Task and Submit</p>

            <div style= {{display:"flex", flexDirection:"row"}}>
            <label style = {{ width: "115px"}}>Task</label>
            <input style = {{ overflow: "auto" , width: "270px"}}
            readOnly
            type='text'
            defaultValue = {props.task.item} />
            </div>

            <div style= {{display:"flex", flexDirection:"row"}}>
            <label style = {{ width: "115px"}}>Update Due Date</label>
            <input
            type='date'
            defaultValue = {props.task.dueDate}
            onChange= {dueDateHandler} />
            </div>

            <div style= {{display:"flex", flexDirection:"row"}}>
            <label style = {{ width: "115px"}}>Update Status</label>
            <select  onChange={statusHandler} defaultValue={props.task.status}>
            {buildStatusOptions()}
            </select>

            </div>
            <input type='submit'/>
            </form>

        return JSX;
    }
    return ( buildJSX());
}
```

# Issues and enhancements

## Volume one Issues

### Propagating changes

In case you have not noticed our Full stack system is not complete. When a web-user inserts or updates a task the result of the transaction is not displayed by our users or task pages. Our web-user must perform a reload on her browser to see any changes. Humm!

Here is code snippet. We receive the ack or nack, inform the web-user and go our merry way!

```
const postAckHandler = (ack) => {
      console.log("ACK -> " + ack.toString());
      if (ack.ok){
        alert("Task update was successful");
      } else {
        alert("Task update failed: " + ack.errorMessage);
      }
    }
```

In volume two we will discuss this issue and fix it! In the meantime, consider the **logic** of having the React application update its cached user and or task data in the App component and re-rendering from the top down when it receives an ack

Here is a 'what if.' What if there are multiple web-users concurrently using our application. What happens to the above **logic**?

### Database connections

You decide if this is an issue. If you think it is then fix it!

Our java server services all our web-users with a single DB connection. Our java Web Controller class protects the single connection by forcing threads to 'wait in line' for the connection. Multiple concurrent web-users implies the chance of multiple java threads competing for the connection!

Db Manager code snippet

```
private Object connectionLock;
...
public List<Task> selectTasks() throws Exception {
        List<Task> list;
        synchronized (connectionLock) {
                list = TaskFetcher.selectTasks(connection);
        }
        return list;
}
```

Questions:

- How **many** concurrent  web-users are expected – worst case scenario?
- Is the **number** in a requirement spec for our application?
- How fast can a web-user type and click- worst case scenario?!

If there is an issue a thread pool would come in handy!

### Other Issues

For other issues you encounter, please contact me!

## Volume two enhancements

### Propagating changes

Updates and Inserts must be made visible to all concurrent web-users. The changes should be made visible a.s.a.p.

### Reports generation

We will add report generation functionality to our application. For example,  a 'manager' may want to see all tasks with expired due dates that are still active. And export the report to a spread sheet.

## Other enhancements

Whatever I can think of before starting out with volume two! For other enhancements you would like to see, please contact me!

**The End**