Principles of Operation

NIST CTS Agents

1 Table of Contents

2 List of Acronyms

CTS	Common Transactive Services
EMA	External Market Adapter
EML	Energy Mashup Lab
LMA	Local Market Agent
LME	Local Market Engine
MA	Market Agent
NIST	National Institute of Standards and Technology
REST	Representational State Transfer
SC	Supervisory Controller
TE	Transactive Energy
TEUA	Transactive Energy User Agent
TRM	Transactive Resource Management

3 Background

Transactive Resource Management (TRM) enables Actors representing systems that use or supply a resource—any commodity whose value is defined by time and delivery location— to coordinate behaviors without the need for central control. TRM-based systems engage Actors in markets to manage supply and demand of a resource over time. Markets enable emergent behavior—new behavior related to actors and relationships as actors meet their internal needs.

TRM systems are highly resilient, as Actors can join or leave the system without additional integration. TRM applications include managing power distribution, smart power grids, smart water, bandwidth sharing, placement of web and social media ads, and wastewater management.

When the resource is electric power, TRM is called Transactive Energy (TE). Transactive Energy is already used to manage the bulk power grid. TE is considered essential to developing new resilient power grids, to transform legacy power grids, and to build resource-constrained grids.

Actor-based architectures enable hyper-scalable applications that are easy to design, build, and maintain. Actor Interactions are limited to defined messages, so they support diversity of participants and technologies. Market transaction messages create self-optimizing systems of suppliers, consumers, and distribution.

This project allows Transactive Energy User Agents (TEUA) to interact through Markets. TEUAs interacts with a Market Agent/Actor (MA) that encapsulates market behavior. While the project uses a Bilateral Market model, the Market Agent incorporates a Market Modular Interface to support other market models.

A bilateral market is a classification for a type of market that allows trades between two exclusive parties. Examples of bilateral markets include a double auction and order book trading.

3.1 Applications

We expect that this project will make it easier for communities, facility owners, and device makers to apply TE. NIST looks to use these agents in simulations to model TE for regulators and legislators. A complete implementation of the Common Transactive Services will be highly visible and widely used.

4 Technical Description

The NIST-CTS Project is a standards-based implementation of the Common Transactive Services and a Market Agent and a Transactive Energy Agent. The architecture drawing shows terminology and relationships.

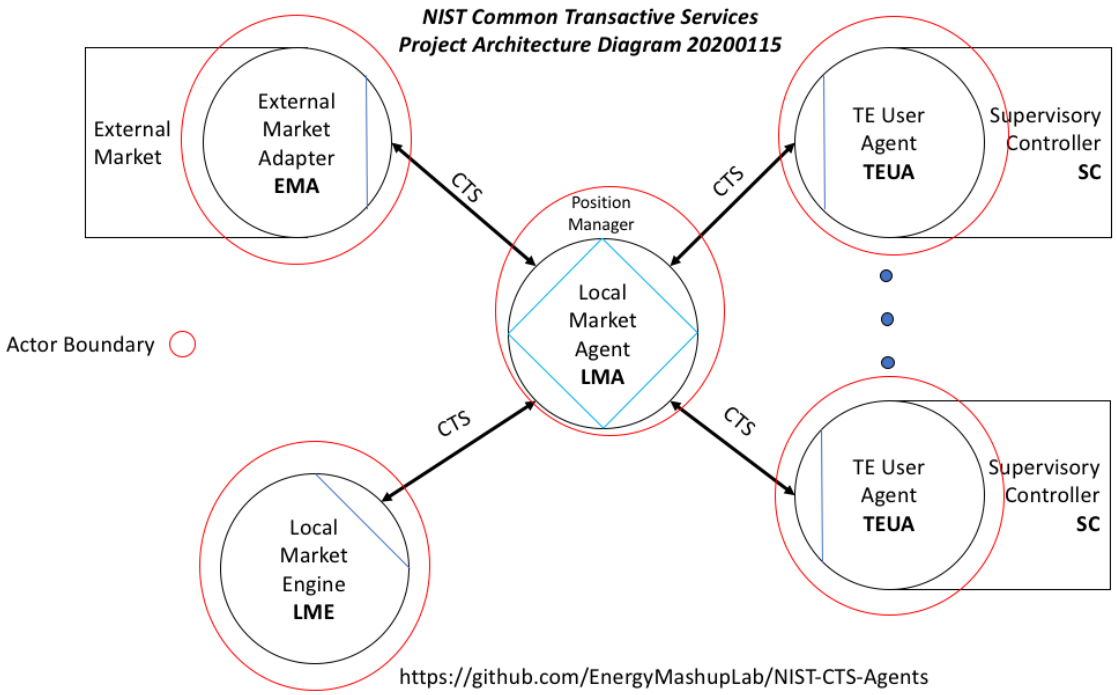

Figure 1 Project Architecture Diagram

The project has a number of components and information in a number of subfolders under ../dev. We use *ei2j* (Energy Interoperation to Java) as shorthand for the integration function.

Markets:

- The Local Market Engine (LME) is the matching engine that coordinates buy and sell tenders
- Markets are connected using the CTS
- A bilateral market

Local Market Agent (LMA): interacts with the local market and with Transactive Energy User Agents and External Market Adapters using the CTS including

- Market Position Management (see note)
- The Ledger is the record of completed transactions.
- Price Adjustment hooks, enabling market economics experiments
- Uses ei2j capabilities for CTS connections
- Links to external markets via the External Market Adapter (EMA) which is an extension of the TEUA

External Market Adapter (EMA): an extension of the TEUA, interacts with the Local Market Agent and a single external market. Functions include

- Market Position Management
- The Ledger records completed transactions.
- Price Adjustment hooks, enabling market economics experiments and presentation of markup on wholesale prices
- Uses ei2j capabilities for CTS connections

Transactive Energy User Agent (TEUA): which interacts with the MA and provides integration capabilities for device and facility management

- Uses ei2j capabilities for CTS connections
- Integrates with the Supervisory Controller (SC)
- Maintains the Ledger, the record of cleared (not pending) transactions (see note)
- Provides information on committed market positions to the SC (see note)

Utilities:

- Common Transactive Services (CTS) implementation
- ei2j-Energy Interoperation to and from Java, includes CTS implementation
- Logging (traces) and input for live and simulation meter and other data
- Ledgers keep records of tenders and transactions.

Note: A ledger is a list in time order of committed transactions. A position is cumulative committed transactions. A trace of messages includes transactions proposed but never cleared. Ledgers are saved to a file or possibly sent over a network connection as the design matures.

The Market Position Manager is a function that tracks completed (cleared) transactions contained in a ledger to determine committed market positions. Market position information is needed by the TEUA (on behalf of the SC), and is maintained by the LMA as transactions are created and cleared.

The TEUA makes information on existing market positions available to the SC, which in turn can use the information to determine the difference between committed position and projected needs. The SC can then transact only for what is needed to align current committed position with projected needs, tendering to buy or sell as appropriate.

All transactions and clearing flow through the LMA, which through the MPM function will update the Market Position for use by the TEUA.

4.1.1 Actors vs. Agents

The difference between Actors and Agents can be a fine one. An actor is an intelligent resource that has the capacity to initiate, manage, and/or control activities of given types. An actor can respond to a message it receives by: making local decisions, creating more actors, sending more messages, or it can determine how to respond to the next message received.

An agent may be a particular instantiation of an actor. Some distinguish the two by whether systems can share direct access to external data. In this case, an agent would be able to access the external data but the actor would not. This project does not wish to delve into these semantics and generally uses the terms interchangeably.

5 RESTful Web Services

Representational State Transfer, better known as REST, determines a pattern for distributed systems to exchange messages and information. REST uses the common pattern of the web (http) to exchange "documents": GET, POST, UPDATE and DELETE. In particular, this project uses GET to request information and POST to send information. UPDATE and DELETE are not used in this system. We specify system details, such as which TEUA, by using a pattern for the address (or URL) of the "document".

Each URL is called a request while the data sent back to you is called a response.

A request is made up of four things:

- The target endpoint
- The headers
- The data/body

5.1.1 Postman

Postman is a collaboration platform for API development. This software helps the team to design, build and test this project's APIs.

You can get all the tenders by going to the following URL in Postman:
http://localhost:8080/tenders/allTenders

You can screen a specific tender by going to the following URL and select GET.
http://localhost:8080/tenders/search/{id}

You can create a new tender by going to the following URL and select POST.
http://localhost:8080/tenders/add

Similarly, you can delete the tender by going to the assigned URL

5.1.2 Controller Class

The Spring Controller Class is simply a class using REST controller annotation

Spring Boot annotations for handling different HTTP request types:

- @RequestMapping — For handling any request type
- @GetMapping — GET request
- @PostMapping — POST request
- @PutMapping — PUT request
- @DeleteMapping — DELETE request

Example:

```
@RestController
@RequestMapping("/tenders")

public class EiCreateTenderType {
        private static final Logger logger =
LogManager.getLogger(EiCreateTenderType.class);
        /*
        *   public actorID counterPartyID; public EiTenderType eiTender; public
actorID
        *   partyID; public refID requestID
        */

        @Autowired
        EiTenderType tenderDao;
        @RequestMapping("/")
        public String home() {
            return "Hello World";
        }

        @PostMapping("/add")
        public EiTenderModel createTender(@Valid @RequestBody EiTenderModel bks) {
            logger.info(bks.toString());

            return tenderDao.save(bks);
        }
    }
```

158

159 Path variables are variables in the request URL and are annotated with @PathVariable. The actual value
160 of the request mapping and the HTTP method determine the target method for the
161 request. @RequestBody will bind the parameters of the method to the body of the HTTP request,
162 whereas @ResponseBody does the same for the response and return type.

163 5.1.3 Model Class

164 Model Class in our case is for Tender which will have tenderID, emixBase, transactionID, status and date
165 as its properties.

166 '`@Table(name="EiTender")`' is used for creating a table with name EiTender in the MySQL database.

167 '`@Column(name="tenderID")`' is used to create a column with a particular name 'tenderID' in
168 the EiTender table in the database.

169 @JoinColumn annotation helps us specify the column we will use for joining an entity association or
170 element collection.

171 Finally, @NotNull annotation is used to apply Not Null Constraint on a column.

172 Getter and Setter methods are used to set and get the required information from Entity Model.

173 Example:

```
@Entity
@Table(name="EiTender")
@EntityListeners(AuditingEntityListener.class)

public class EiTenderModel {

        //private static final java.sql.Date CurrentDateTimeProvider = null;
        @Id
        @NotNull
        private long tenderID; // Primary key, we can have only one tender with the
same ID
        @NotBlank
        private String emixBase;

        public String status = "Created";
        private LocalDate Date = gdate();

        public LocalDate gdate() {
            LocalDate localDate = LocalDate.now();

            return (localDate);
        }
    }
```

5.2 URI Structure for REST Service Operations

POST operations have a RequestBody (the message that is POSTed to the listed URI) and a ResponseBody (the message body that is returned to the actor doing the POST). This provides the standard Energy Interoperation messages - a POST RequestBody contains an EiCreateTender, while the POST ResponseBody contains the correlated EiCreatedTender.

For this project the principal authors of the base standards flattened the type hierarchy for only the product (energy) and information elements we use. This approach maintains standards conformance and allows for

- A simpler to use and understand type system
- Simpler Java class definitions for standard payloads
- A conformance statement at the end of the project

NIST-CTS-Agents uses JSON rather than XML for message payloads. The project uses Jefferson serialization and deserialization between Java and JSON.

LMA

/createTender	POST	@ResponseBody is an EiCreatedTender object
/createTransaction	POST	@ResponseBody is an EiCreatedTransaction object
/cancelTender	POST	@ResponseBody is an EiCanceledTender object
/party	GET	@ResponseBody is an ActorId containing actor's partyId

LME

/createTender	POST	@ResponseBody is an EiCreatedTender object
/cancelTender	POST	@ResponseBody is an EiCanceledTender object
/party	GET	@ResponseBody is an ActorId containing actor's partyId

TEUA and EMA

/teua — {id - sequential integer assigned on creation}

/{id}/CreateTransaction	POST	@ResponseBody is an EiCreatedTransaction object
/{id}/CreateTender	POST	@ResponseBody is an EiCreatedTender object. For user entity integration
/{id}/party	GET	@ResponseBody is an ActorId containing actor's partyId

Refer to the Architecture Diagram for more detail on the RESTcontrollers.

5.3 LMA Pseudocode

Logical Description:

1. Receive CreateTender service request (from a TEUA) log in transport message list
2. Respond to the TEUA with a CreatedTender log
3. Adapt and send a [rewritten] CreateTender to LME log in transport message list
4. When LME matches and clears it will send LMA back a CreateTransaction log and ledger,
5. also log in transport message list
6. Send CreatedTransaction back to LME log and ledger
7. Send [rewritten] CreateTransaction to requestion UA log and ledger for UA
8. Receive CreatedTransaction from TEUA log and ledger; update in ledger to note acknowledgment

POST methods:

1. POST action (request from SC, CreateTender for full requirements for a time period)
    a. Query position
    b. Compare, subtract and POST remaining requirements as CreateTender to LMA. Log
    c. RETURN/POST CreatedTender message to SC
2. POST action (request from LMA, CreatedTender)
    a. Log
    b. Inform SC (callback or POST)
3. POST action (request from LMA, CreateTransaction for cleared transaction)
    a. Enter in my Ledger
    b. Add to my position. Should already be in my position stored at LMA
    c. RETURN/POST CreatedTransaction to LMA

NOTES: Hook in LMA POST method for possible rewrite. Only the hook so can still experiment with rewrite rules.

5.4 LME Pseudocode

Uses only generated libraries from Parity, specifically parity.libraries.book and .market, and consumes classes from util.

Logical Description:

1. Receive a CreateTender service request from LMA log in transport message list
2. Respond to the LMA with a CreatedTender log
3. Enter the Tender in the Order Book
4. When tenders match and clear send LMA a CreateTransaction log and ledger also log in
5. transport message list
6. Receive CreatedTransaction from LMA log and ledger
7. LMA will send CreateTransaction to requesting UA

POST Methods:

1. POST action CreateTender (request from LMA, tender for UA net requirements for a time period)
    a. book.market.add(orderID, details) which adds to bid/ask data structures as relevant
    b. RETURN/POST CreatedTender messaged to LMA.
2. Spontaneous calls from internal MarketListener on match
    a. Accept callback
    b. market.execute(orderID, quantity, price) which clears from bid/ask data structures as relevant
    c. POST CreateTransaction to LMA
3. POST action for CancelTender (request from LMA)
    a. market.delete(orderID)
    b. RETURN/POST CanceledTender to LMA
4. POST CreatedTransaction
    a. Log

NOTES:

1. The CTS IDs (inherited from idType in EI) should be used in the OrderBook (the Parity ID is a long)
2. CTS does not rewrite tenders in place, so CancelTender == market.delete (Parity Cancel adjusts quantity)

5.5 TEUA Pseudocode

Logical Description:

1. Receive a service request to buy/sell energy (from the SC) and net against position for that time period
2. log in transport message list for debugging
3. Build net CreateTender payload log with all fields of the Tender (party, counterparty, Tender, etc)
4. Send net CreateTender to LMA (log in transport message list)
5. Wait to receive a CreatedTender (log receipt) and later a CreateTransaction (log and ledger updates)
6. Update my position (in LMA)
7. Respond to the CreateTransaction with a CreatedTransaction (log)

POST methods:

1. POST action CreateTender (request from SC, for full requirements for a time period)
    a. Query position
    b. Compare, subtract, and POST remaining requirements as CreateTender to LMA (log)
    c. Return CreatedTender message or indication to SC (CTS or callback/return on simple method invocation or POST)
2. POST action CreatedTender (from LMA)
    a. Log
    b. Respond to SC (callback or POST)
3. POST action CreateTransaction (request from LMA, CreateTransaction for cleared transaction)
    a. Enter in my logical Ledger (correct to place in LMA ledger for query, logically specific to this TEUA)
    b. Add to my postion should already be in my position as stored at LMA
    c. POST CreatedTransaction to LMA

NOTES:

1. The SC might invoke a method OR use a RESTful web service; the latter is possible if in the same code space/JVM as the TEUA.
2. Position manger is attached to LMA, not to each UA.
3. The position manager can be a separate RESTful service. It needs to be queried by other than the LMA.

6 Position Manager

The payload for the position manager is stored in the org.theenergymashuplab.cts.controller.payloads package. The REST call methods that access the position manager are stored in this package.

Position Manager Controller Methods:

- createPosition: Will add position into the table.
- getPositionHistoryToId: Fetch position from table with respect to the sellerId.
- getPositionHistoryFromId: Fetch position from table with respect to the buyerId.
- getPositionHistory: Fetch position from table with respect to the count.
- getStatus: Fetch status from table given the id of respective position id.

The model for the position manager is stored in org.theenergymashuplab.cts.model package. This model represents the database table structure .

The position repository is stored in org.theenergymashuplab.cts.repository package. This is where the native queries are stored.

7 Logging

Please note that Log4j 2 functions are included through the Spring Boot Starter for Log4j 2, spring-boot-starter-log4j2. This project will continue using that version until we need to update either Log4j 2 or Spring Boot.

The Apache Log4j 2 2.11.2 library is used for logging. This version was published February 2019, and is available in the Maven Central Repository in log4j-core 2.11.2 and log4j-api 2.11.2. The base library is available at the Apache Log4j 2 archives

In the NIST-CTS-Agents repository, the configuration file is in /dev/src/main/resources/log4j2.xml.

Logs are stored on your local system in the dev/logs folder. Ledgers are stored on your local system in the dev/ledger folder. Ledgers (see the project README) contain committed transactions, and may be used to build an actor's position.

7.1 Logging Levels

Log4j 2 supports conventional and standard logging levels as well as custom levels. The standard levels are, from most specific to least specific (and lowest IntLevel to highest) is:

- OFF (most specific, no logging)
- FATAL (most specific, little data)
- ERROR
- WARN
- INFO
- DEBUG
- TRACE (least specific, a lot of data)
- ALL (least specific, all data)

More information on the logging levels is available from the Apache Log4j 2 documentation.

7.2 Log4j2.xml

There are two types of rolling logs, one is for normal logging and the other is for ledgers. All logs described here are tab-separated text to simplify analysis with Excel and other spreadsheets (rather than for best human readability).

All the of logs from org.theenergymashuplab go in the logs folder dev/logs of your local system and print on the console.

The levels set are TRACE (used rarely) and INFO; see the source code and dev/src/main/resources/log4j2.xml. The level that is outputted can be changed in log4j2.xml.

Logs created by org.theenergymashuplab.cts.controller.payloads.EiCreateTransactionType are set to level INFO and will go to the ledger folder.

The trace level to be logged can be changed in log4j2.xml.

All the logs generated by hibernate (for MySQL interactions) will be printed on console and go to the application general log.

8 Built With

If you are using Spring Tools Suit 4 for Eclipse you do not need to download Apache Tomcat 9. Only download Apache Tomcat 9 if you are using the regular Eclipse IDE for development.

For Windows, MacOSX and Linux systems you will need:

- Java 8 JDK
- Maven 3.x
- Spring Tool Suite 4 for Eclipse
- MySQL 5.1 or later
- MySQL Workbench 5.1 or later

MySQL Workbench may require a MacOS system update.

9 Running

Importing Maven Projects from Git into Eclipse

Step 1: Select the folder where you want to create the local storage (on your local machine). Right click and select 'Git Bash Here'.

Step 2: Use the command 'git clone' to create a clone of the repository

Step 3: In Eclipse select the File menu and then select Import

Step 4: Select Existing Maven Projects and click on next. (Note that Eclipse from the Spring Boot download and direct download work the same)

Step 5: Browse to the local NIST-CTS-Agents folder (cloned from Github) and click finish. The project is now imported in Eclipse.

Step 6: Right click on project, go to Run As and select Maven Build.

You will get a Build success message in the console. Note that the Build button will use the most recent detailed build instructions during the current execution of the Eclipse environment

Step 7: Configure Tomcat Server

Detailed notes at this site

Step 8: Select tomcat server from the list of those installed on your local machine and click next.

Spring Tool Suite 4.5 does not have this wizard selector; instead for Step 8 and Step 9 right click on the project and select a server.

Step 9: Move the project to configure it to the server *Picture shows after selecting the project and clicking Add*

Step 10: Create the Database

Database initialization was done using MySql Community Edition and Workbench 8.0.18 on Mac OS X 10.15.2. SpringTool Suit 4 and JDBC will initialize the database and schemas with no manual intervention when using the developement enviroment mentioned before.

If you are not using this enviroment, you may have to create the database and schemas manually. Username is set as eml@localhost and the password is set as capstone123.

Manual Database Initialization:

1. In MySql Workbench, create a new user "eml" with the password "capstone123".
2. Create a new "Schema" (database show with the disk icon) called nist_cts_eml
3. Give user capstone permission to access nist_cts_eml from localhost
4. Remember to apply all changes and refresh the nist_cts_eml schema.
5. Run the project application in SpringToolSuite
6. In MySql Workbench, refresh all for the schema nist_cts_eml.

Step 11: Right click on the project, go to Run As, and click on Spring Boot App. You can also use Run As → Run on Server.

Step 12: The project is now running. Open the browser and go to localhost:8080 to view the project

10 Testing

JUnit is an open source unit testing framework for Java. It is useful for Java developers to write and run repeatable tests for small chunks of code.

When using JUnit in Spring, there are several features added that many developers are not aware of.

First, if you are including the Spring Context in your tests, it becomes an Integration Test, no longer a Unit Test.

To integrate Spring with JUnit, you need spring-test.jar Specifying dependencies in pom.xml.

Example:

```
<!--Spring boot starter web-->
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-test</artifactId>
    <scope>test</scope>
    <exclusions>
        <exclusion>
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
        </exclusion>
    </exclusions>
</dependency>
<dependency>
    <groupId>org.junit.jupiter</groupId>
    <artifactId>junit-jupiter-engine</artifactId>
    <scope>test</scope>
</dependency>
```

10.1 Creating Unit Test Classes

For the unit tests to run a batch job, the framework must load the job's ApplicationContext. The annotations @RunWith(SpringRunner.class) and @ContextConfiguration are used to trigger this behavior.

@RunWith(SpringRunner.class) indicates that the class should use Spring's JUnit facilities

@ContextConfiguration indicates which resources to configure the ApplicationContext with. In this application, we use @Autoconfigure with @WebMvcTest, imported from org.springframework.boot.test.autoconfigure.web.servlet.WebMvcTest, which will disable full auto-configuration and instead apply only configuration relevant to MVC tests, i.e. @Controller @ControllerAdvice, @JsonComponent, @Converter/@GenericConverter, @Filter, @WebMvcConfigurer and @HandlerMethodArgumentResolver beans but not @Component, @Service or @Repository beans.

By default, tests annotated with @WebMvcTest will also auto-configure Spring Security and MockMvc, including support for HtmlUnit WebClient and Selenium WebDriver. For more fine-grained control of MockMVC the @AutoConfigureMockMvc annotation can be used. Typically @WebMvcTest is used in combination with @MockBean or @Import to create any collaborators required by your @Controller beans.

Example:

```
@RunWith(SpringRunner.class)
@WebAppConfiguration
@WebMvcTest(EiCreateTenderType.class)
public class EiCreateTenderTypeTest {
@Autowired
private MockMvc mvc;

@MockBean
private EiCreateTenderType eCTT;

@Test
public void home() throws Exception {
    mvc.perform(get("http://localhost:8080/tenders/"))
            .andExpect(status().isOk());
}

@Test
public void add() throws Exception {
    EiTenderModel bks = new EiTenderModel();
    bks.setTenderID(12334);
    bks.setEmixBase("434fsdfssdq2mn3123mnxcvxc");
    bks.setTransactionID(4234234);
    Map<String, String> map = new HashMap<>();
    map.put("tenderID", "12334");
    map.put("emixBase", "434fsdfssdq2mn3123mnxcvxc");
    map.put("transactionID", "4234234");
    mvc.perform(post("http://localhost:8080/tenders/add"))
            .andExpect(status().isOk());
}
```

The above code structure is a Text fixture. A test fixture is a context where a Test Case runs. Typically, test fixtures include:

- Objects or resources that are available for any test case.
- Activities that make these objects/resources available such as:
    - Allocation (setup)
    - De-allocation (teardown)

If you are looking to load your full application configuration, you should consider @SpringBootTest.

An example from */src/test/java/com/eml/energy/ EnergyApplicationTests.java*:

```
@RunWith(SpringRunner.class)
@SpringBootTest
public class EnergyApplicationTests {

    @Autowired
    private EiCreateTenderType ctt;
```

11 Authors

- **William Cox** - *Architecture* - Cox Software Architects LLC
- **Toby Considine –** *Architecture* – TC9 Inc

See also the list of [contributors] who have contributed to this project.

12 License

This project is licensed under the Apache 2.0 License, and is Copyright 2019-2020 The Energy Mashup Lab.

For incoming (contributed) licenses see https://github.com/EnergyMashupLab/EML_Licenses

13 Standards Used

The TEMIX profile of OASIS Energy Interoperation. Energy Interoperation is the profile base of [OpenADR 2] standardized as [IEC 62746-10-1] (https://webstore.iec.ch/publication/26267)

- Informative UML models for Energy Interoperation/CTS payloads as shown in the EI Standard
- ISO 17800 Facility Smart Grid Information Model (https://www.iso.org/standard/71547.html)
- Adapter methods for integrating with Independent System Operator Wholesale Markets and other energy markets are based on [IEC 62746-10-3:2018] (https://webstore.iec.ch/publication/59771)