Workflow Business Case: Policy Updates with Workflow Orchestration, Case Management with BPMN and Custom UI

Doc Writer – paul.lungu@camunda.com – Version 1.0, 2021-05-10

Table of Contents

Chubb PoC Goals

✓ **Identify gaps in Camunda** (Hopefully the PoC itself provided the context for this goal)

✓ **Pros and Cons of forms with Camunda** (Discussed need for custom UI integration, forms integration with Camunda is flexible and allows the use of third party form frameworks. See Example UI Integration)

✓ **Integration with custom UI** (See Example UI Integration and the Custom Traige Tasks)

✓ **Case managemet with Camunda and BPMN** (See Example UI Integration and the Case Management use-case)

✓ **Integration with Kafka** (See Kafak Integration)

✓ **Business Data Integration** (See Handling Business Data in Workflows)

✓ **Reporting, metrics and capturing history events** (Demoed Optimize see Integrations for more on metrics and capturing events)

✓ **Email integration with intake framework** (This was completely mocked with Postman due to the complexity of the integration)

Important Links

- See the code in github (https://github.com/plungu/camunda-chubb-poc-starter). You can downlaod and run the project.
- See more about Reporting and tracing with Optimize (https://camunda.com/products/camunda-platform/optimize/reports/)
- See more about Enterprise Features (https://camunda.com/enterprise/)
- See more about Enterprise Cockpit Features (https://camunda.com/products/camunda-platform/cockpit/#features)
- See more about Customize Modeler with Element Templates (https://github.com/camunda/camunda-modeler/tree/master/docs/element-templates)
- See more about Customize Modeler with Plugins (https://github.com/camunda/camunda-modeler-plugins)
- See more about Transactions in Process (https://docs.camunda.org/manual/latest/user-guide/process-engine/transactions-in-processes/)
- See more about Camunda Engine Plugins (https://github.com/camunda/camunda-bpm-examples/tree/master/process-engine-plugin)
- See more about Camunda Spring Event Bridge (https://docs.camunda.org/manual/7.14/user-guide/spring-boot-integration/the-spring-event-bridge/)
- See more about Understanding Camunda History (https://docs.camunda.org/manual/7.12/user-guide/process-engine/history/)
- See more about Accessing Historical Data (https://camunda.com/best-practices/reporting-about-processes/#_accessing_strong_historical_data_strong)
- See more about Understanding Camunda Database (https://docs.camunda.org/manual/latest/user-guide/process-engine/database/database-schema/)
- See more about Understand External Tasks (https://docs.camunda.org/manual/latest/user-guide/process-engine/external-tasks/)
- See more about Modeling and Collaboration with Cawemo (https://cawemo.com/)

Use Case

Summary

Post Buying Process use-case is demonstrating updating a insurance policy based on certain events that can trigger a policy update.

The Web Based UI demonstrates the Triage of a Policy Update allowing the Triage role to see tasks based on skills and entitlement.

A single Triage task combined with an Event Sub-process is used to simulate a case management pattern with BPMN.

Additionally, DMN based business rules are utilized to route tasks to the correct business unit and assign the correct user roles to tasks.

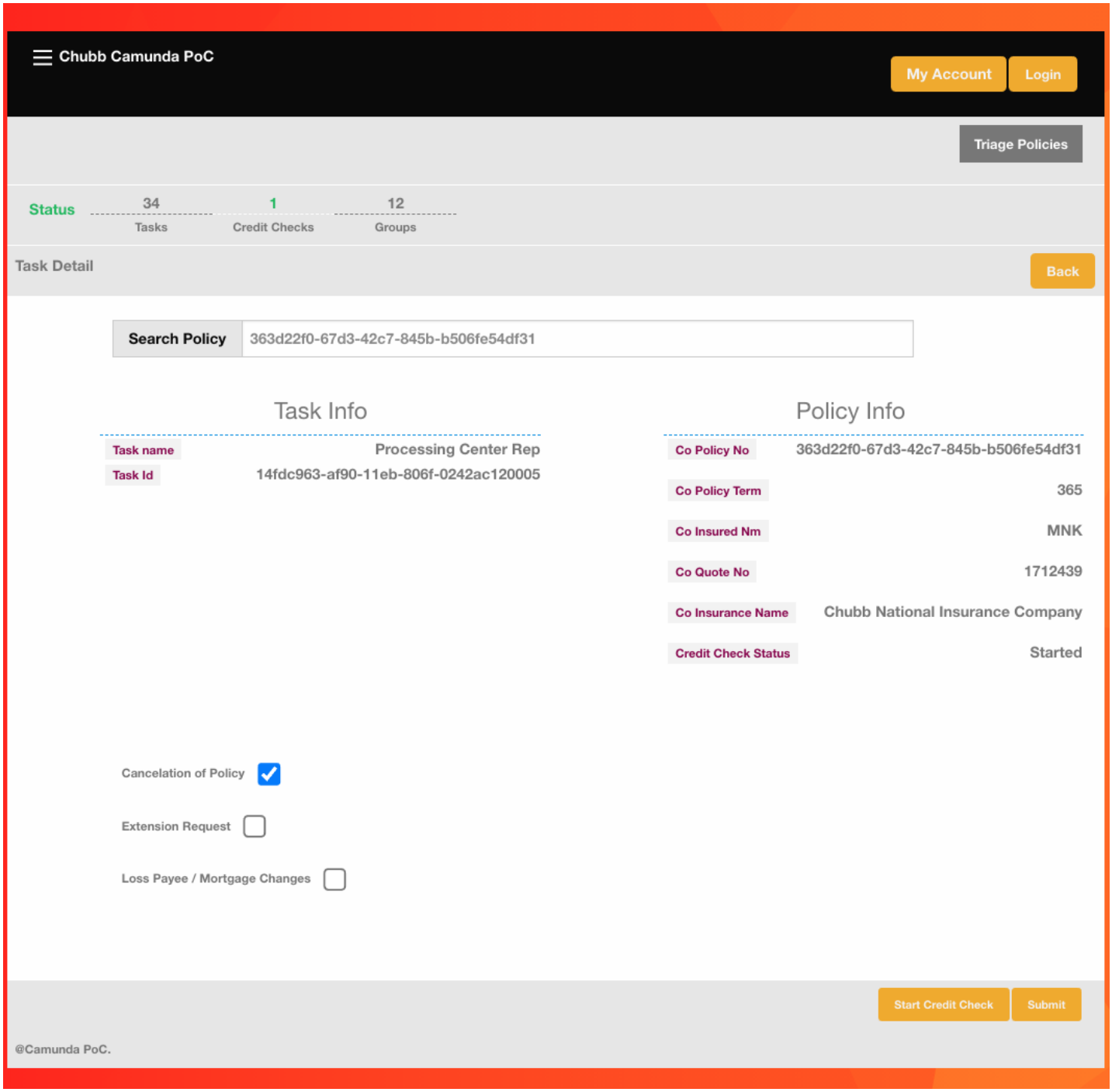
History data should be sent to Kafka to potentially store in a BI solution.

Workflow

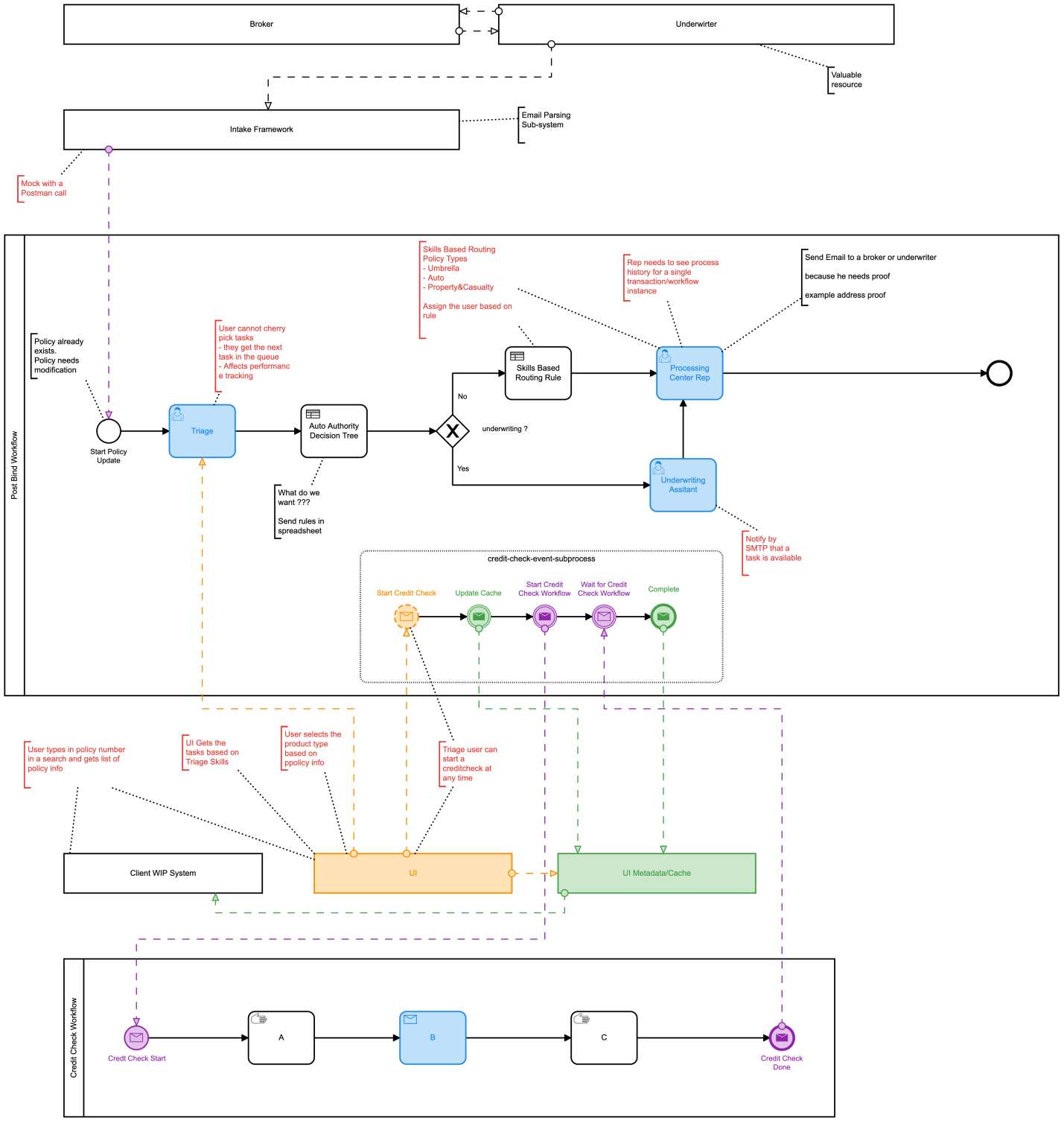
Underwriters and Brokers interact to potentially trigger a policy update. This is normally done through email interactions. When a policy update is needed an underwirter will send an email to a shared email box. The Intake Framework with parse and determine the intent of the email then start a policy update workflow (post-bind-workflow) when appropriate.

> **NOTE** The intake framework is simulate by a Postman call to start the workflow. See the `postman` directory for the request to start the workflow.

The triage task is then activated. This enables a triage user to view the task in the Custom UI and prepare the task for Underwriting and Processing. Additionally, a case management feature can be used by the triage user potentially starting a credit-check.

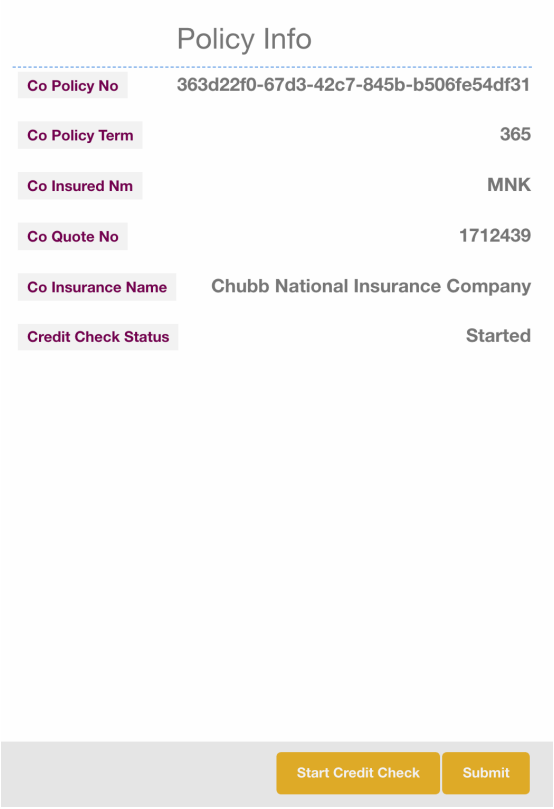


Once the triage user submits the task business rules are used to assign and route the tasks to the correct business unit and users.



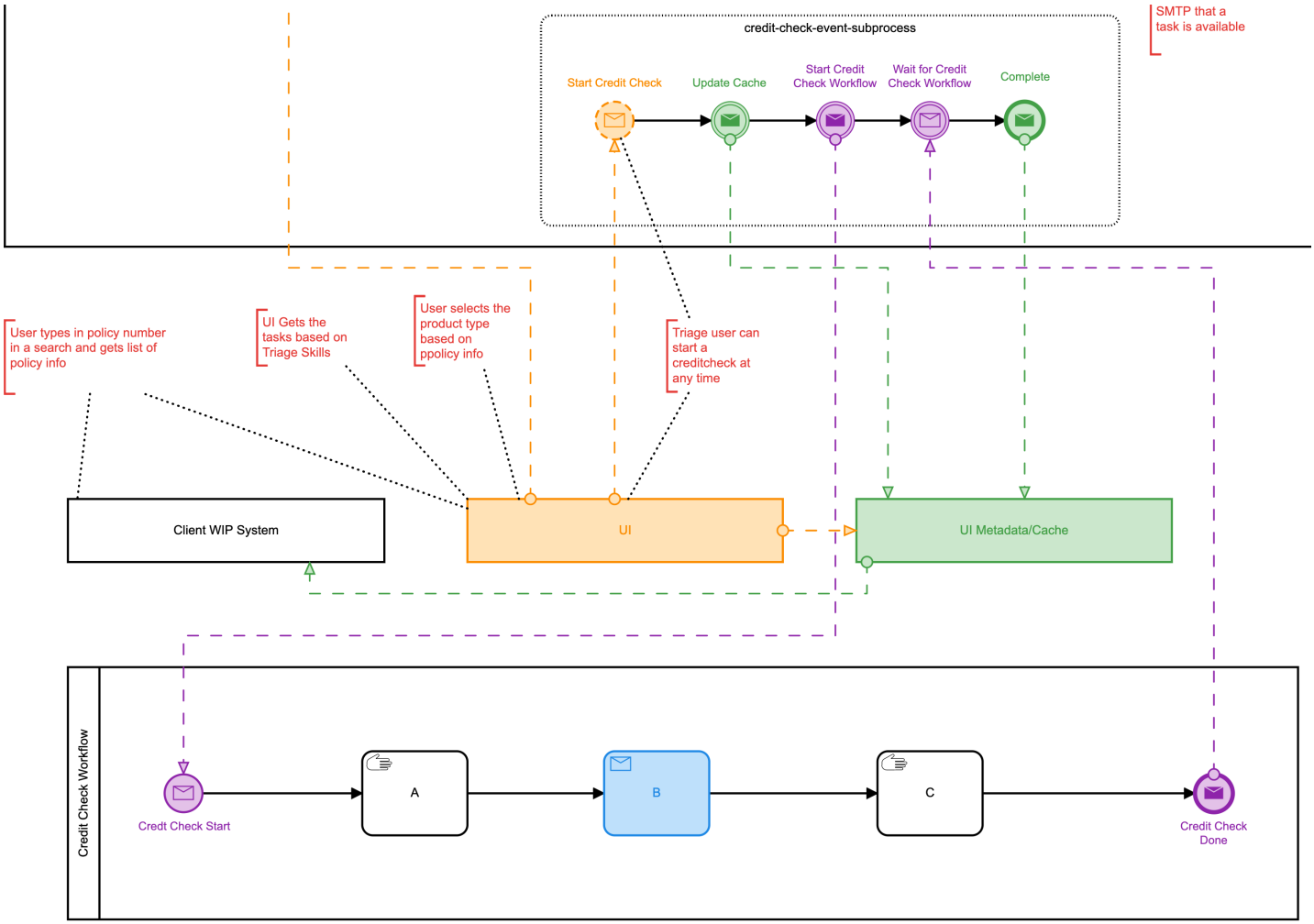
Case management with Custom UI and BPMN

This section of the process illustrates case management with BPMN. The Triage user is able to start a credit check from the UI. The use clicks the `Start Credit Check` button which triggers the Credit Check SubProcess.



The Credit Check sub-process is broken into two phases. An Event subprocess is utilized to capture the click from the UI and allows us to monitor the status of the credit check from the parent process. Additionally, using the Event sub-process doesn't interfere with execution of the parent process. In other words the Triage user could also submit the task and continue the normal flow of the process while the credit check continues.

A separate BPMN workflow is also utilized for the Credit Check. It does the actual work of orchestrating the credit check. This allows for reuse of the credit check workflow and concise separation of concerns.



Integrations Summary

The integrations with the UI are REST based. Starting, retrieving, completing tasks and workflows are done through the Camunda REST API.

Technical integrations like email integration is done through Camunda delegates and listeners. When a user task is started an email is sent to the user.

Additionally, Kafka integration is done through spring-eventing-bridge and spring-clould-streams framework for the purpose of sending Camunda event information to other systems i.e. BI reporting solutions. See Patterns for Integration and Kafka Integration

More on all these topics in later sections. Jump to section UI Integration Spring Eventing Bridge

Running the use-case

IMPORTANT if you want to use Camunda Enterprise you will need to download the enterpise license. See the section on setting up Camunda Enterprise

Option 1. Use spring-boot at the command line

```
mvn spring-boot:run -Dspring.profiles.active=servicerequest,integration,cors
```

Option 2. Use docker-compose

This should allow you to simple run the in a self contained manner. You must have docker installed on the machine.

```
docker-compose up
```

Option 3. Build the jar

Build the .jar with maven

```
mvn clean package -DskipTests
```

Run the .jar with java

java -Dspring.profiles.active=servicerequest,integration,cors -jar ./target/camunda-poc-starter.jar

Option 4. Use Kuberneties and Delcaritive Infra

This is still in progress.

Using Profiles

Profiles can be specified at the command line when the application starts. The notation is as follows.

```
-Dspring.profiles.active=servicerequest,integration,cors
```

Or you can use the application.properties file to specify the profile.

```yaml
spring.profiles.active: servicerequest,integration,cors
```

YAML

IMPORTANT Using Camunda Enterprise - it's necessary to request a Camunda Enterprise License.

See it running

```
Visit `http://<server>:<port>/sr` to access the React app


Visit `http://<server>:<port>/camunda/app/cockpit` to access Camunda Cockpit
```

Using Enterprise Camunda

IMPORTANT Using Camunda Enterprise - it's necessary to request a Camunda Enterprise License (https://camunda.com/download/enterprise/).

Update the settings.xml file in the project home with the credentials that you get from Camunda. This will allow you to pull the enterprise artifacts from our nexus repository.

NOTE You may need to configure a proxy to reach our repository. See the example in the settings.xml.

Testing with Postman

Use the postman collections in the `postman` folder.

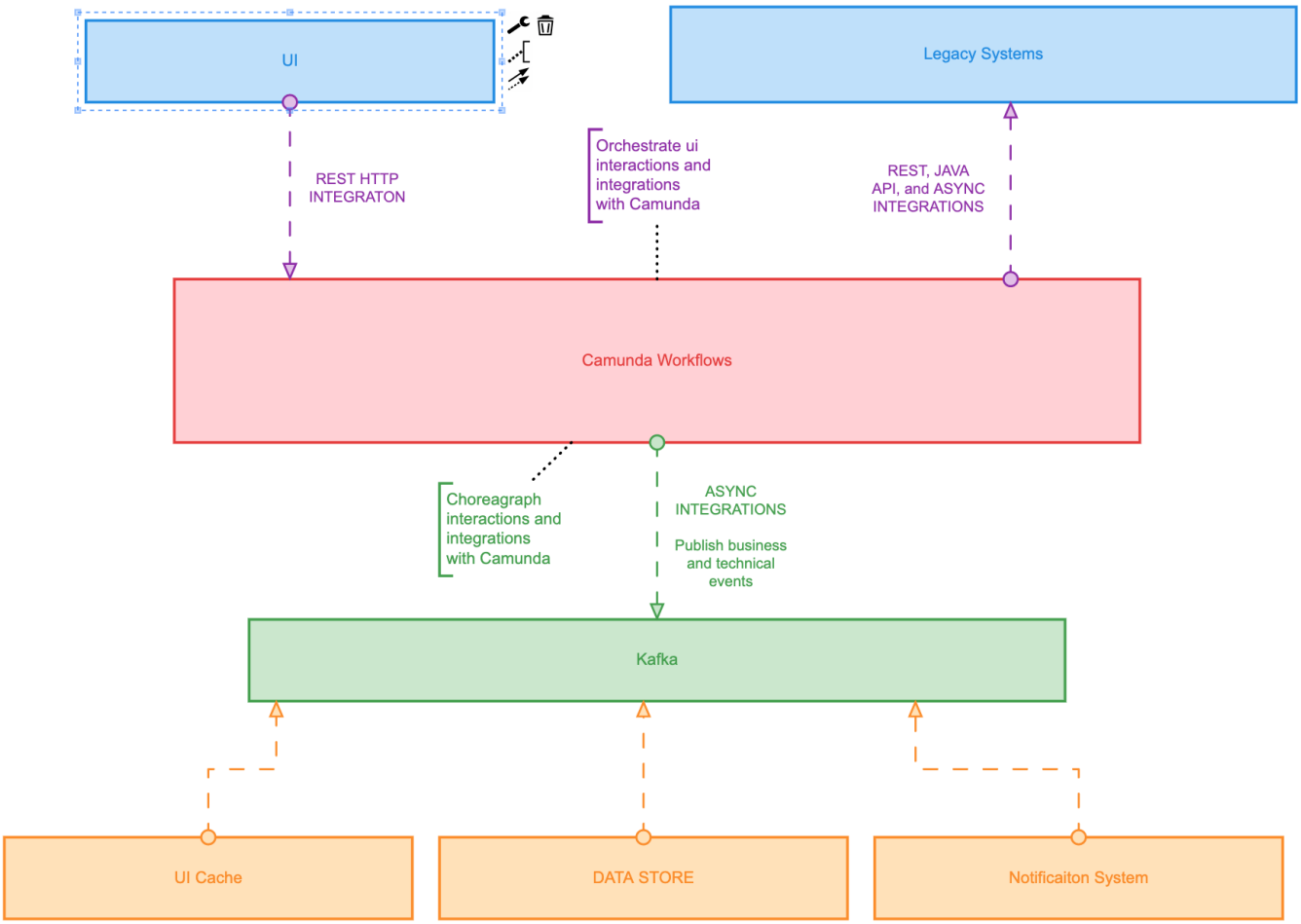
With post-man you can move through the processes simulating REST requests to the app.

- starting the processes
- completing user tasks
- correlating messages
- loading test data

Architecture

The architecture diagram below generically demonstrates the possibilities and patterns interacting with the workflow. It's worth noting this is not a top-down model for interacting with the workflow, merely one possible approach and patterns that can be generally applied.

Aync and synchronous interaction with the engine is possible and should be considered carefully based on the business and technical requirements of the use-case.



- The green bar signifies kafka. Events are published to kafka from the workflow. The implementation for the publishers can be in . This pattern works well as we can utilize the Delegate and the Send task to control the execution of the workflow and potentially ack kafka and handle incidents when Publishing fails.

- Green lines signify publishing of asynchronous messages from the workflow.
- The orange boxes signify components that sub-scribe to kafka and update other components based on the Events that they receive.
- Orange lines signify subscription from external systems to Kafaka topics for various purposes.
- The blue boxes signify components that do specific work and are potentially interact with the subscription components in yellow. Or could interact more synchronously with the workflow, though often through an abstraction layer.
- Purple lines and events signify point-to-point synchronous HTTP/REST interactions with the workflow and other components.

NOTE An important takeaway is that different patterns for interactions with the workflow are possible and often desirable. It can make sense to have a orchestrated approach in some cases and a choreographed approach in other cases within the same workflow or broader use-case.

NOTE This is a typical pattern for micro-service architectures though the level of abstraction between components will vary from use-case to use-case.

Abstraction Layers

Another important requirement is to integrate with other systems such as UI, entitlement, authorization and authentication, as well as other legacy systems. To meet this requirement it's often desirable to have an abstraction layer around Camunda. This abstraction layer can include several different technologies include custom REST API's, Kafka, API aggregation tools like MuleSoft, GraphQL, caching technologies such as Hazelcast and others depending on your use-case. There really is no limitations with Camunda.

Architecture Patterns

COMMAND

Spring REST controllers can be used to post data to the workflow. Simple and concise API's are defined should be deifned for interaction with the process. See `WorkflowController.java` The API always takes an object and returns and HTTP Status. The RESTful endpoint context mapping is associated to Commands that can happen in the system. Such as START, APPROVE and REJECT and intended to work in a point-to-point(request/response) synchronous fashion.

The controllers only write POST/PUT data CREATING and UPDATING only.

TIP The intention is to create a CQRS(Command Query Responsibility Segregation) (https://martinfowler.com/bliki/CQRS.html#:~:text=CQRS%20stands%20for%20Command%20Query,you%20use%20to%20read%20information) pattern for interaction. This can increase scalability while reducing complexity in distributed systems. The takeaway is to create data-stores that are optimized for different types of interactions primarily writing and reading.

QUERY

A separate API and logically separate data-store is used to query business data. Updates to this data-store always happen in an asynchronous fashion. For example when a Approval task in the work flow completes the workflow publishes an UPDATE-SERVICE-REQUEST event. The subscriber reads the event and decides what to do. In certain cases it updates the object in the data-store. Other components can now read from the data-store such as the UI.

We can guarantee the data is published to the data store with the workflow. See the section above on the JavaDelegates that implement the publishing functionality.

Spring Data JPA (https://spring.io/blog/2011/02/10/getting-started-with-spring-data-jpa) is the technology used for the ServiceRequest data. Spring Data allows for an easy way to create API's that are easy for a UI to query. Also an easy way to combine data into a useful form for the UI to consume.

Patterns for Integration

It's important to consider how to implement integrations with a workflow. Two basic patterns exist, integrate within the workflow execution or outside the workflow execution. The sections below provide details about the two patterns.

Access to the workflow execution context

The workflows in this project utilize the workflow execution to perform actions and interact with other systems and control the flow of custom code execution through the use of Java Delegate and Execution Listeners.

Java Delegates as shown below are marker interfaces that allow the executing class to be passed the execution context know as the DelegateExecution object. This object contains many functions that allow the Delegate to interact with the engine and data in the process. This is also a natural extension point that can be used to integrate with other systems. An example would be calling an email service to send email based on workflow variables.

```
@Component("emailDelegate")
public class EmailDelegate implements JavaDelegate {

    private final Logger LOGGER = Logger.getLogger(LoggerDelegate.class.getName());

    private EmailService emailService;

    @Autowired
    public EmailDelegate(EmailService emailService) { this.emailService = emailService; }

    @Override
    public void execute(DelegateExecution execution) throws Exception {

        String emailTo = "paul.lungu@camunda.com";
        String emailSubject = "Camunda PoC Demo Email";
        String emailbody = (String) execution.getVariable( s: "message");

        emailService.sendSimpleMessage(emailTo, emailSubject, emailbody);
    }
}
```

Execution Listeners function in much the same way as a JavaDelegate and the interfaces can be used interchangeably. Conceptually the execution listener is intended to be placed in the process in a more subtle way. **Execution Listeners** are placed in the execution on events that are part of the workflow execution. See the docs to understand more about Delegation Code (https://docs.camunda.org/manual/latest/user-guide/process-engine/delegation-code/#java-delegate)

Read more about placing listeners here (https://docs.camunda.org/manual/7.12/user-guide/process-engine/transactions-in-processes/#understand-asynchronous-continuations). Also, in the same doc (https://docs.camunda.org/manual/7.12/user-guide/process-engine/transactions-in-processes/) read about wait states and transaction bounderies to provide more contect on the operation of the engine.

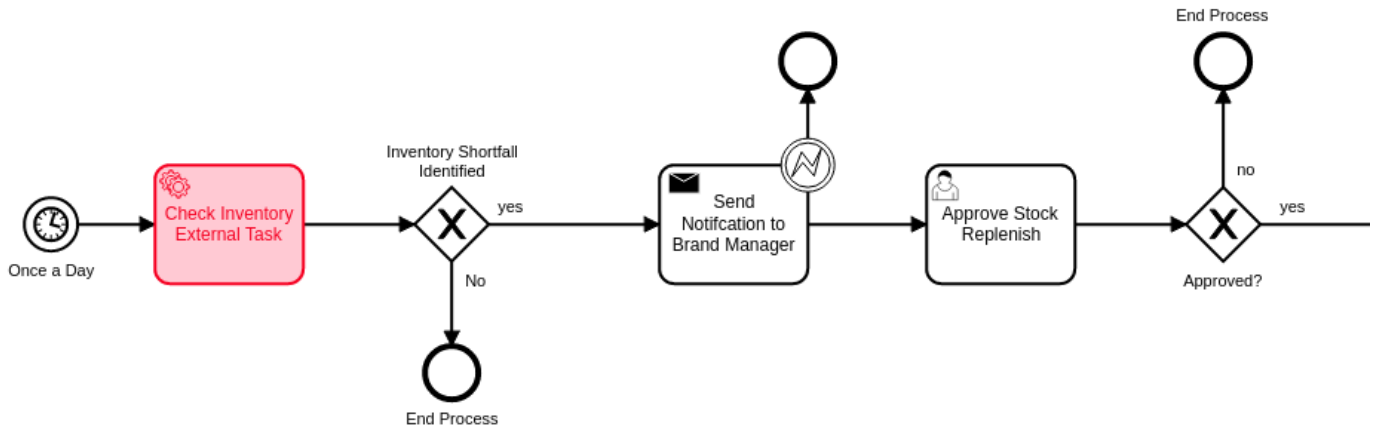
```java
@Component("OutstandingTaskEmailExecutionListener")
public class OutstandingTaskEmailExecutionListener implements ExecutionListener {

    @Override
    public void notify(DelegateExecution execution) throws Exception {

        // hardcode test
        execution.setVariable( s: "emailTo",  o: "chris.allen@camunda.com");
        execution.setVariable( s: "emailSubject",  o: "Outstanding Order Task");
        execution.setVariable( s: "emailBody",  o: "This is a reminder that you have PM order tasks available to you.");
        execution.setVariable( s: "emailFrom",  o: "info@foo.com");
        execution.setVariable( s: "emailBcc",  o: "chris.allen@camunda.com");

    }
}
```

External Task Pattern for Polyglot Programming



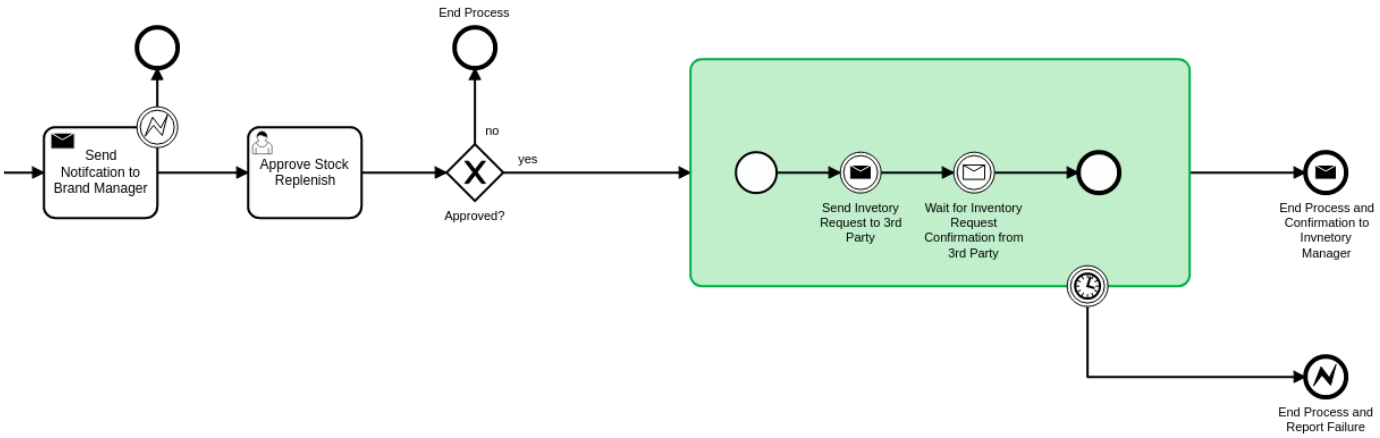
The red activity above illustrate the external task pattern configuration. The single task is configured as a topic. The control is inverted and a calling worker will application will ask for work from the engine through the external task API. The external task API is quite powerful allowing the caller to request work in bulk and complete tasks in bulk. Also it allows for creating incidents and errors in the engine.

Additionally, the worker application can be written in any technology as it is independet of the Camunda engine.

See the external task docs (https://docs.camunda.org/manual/develop/user-guide/ext-client/) for more about the API.

See the Best Practices (https://camunda.com/best-practices/invoking-services-from-the-process/#_understanding_and_using_strong_external_tasks_strong) for insights on external tasks.

Send and Recieve



The **Send and Receive** or bup/sub pattern is used in cases where simplicity in interaction with API's is preferred to the power and complexity of the external task api. It also provides some flexibility in the modeling approach as the send activity and receive activity do not need to be in sequence. Other activities can be used in parallel with the send and receive activities.

It's often required to use a small amount of Java code, see Java Delegate section, with the **Send** task which provides flexibility in implementation and allows integration with other messaging systems. The publishing Send task code should likely be dumb and only do very specific publishing function.

The **Receive** task can be correlated through the REST API and any technology can be used to implement the subscribing application. The subscriber would likely live outside the engine application context.

Kafka Integration

The spring-boot app is using spring cloud streams.

https://spring.io/projects/spring-cloud-stream

The app has a has a single publisher and a single subscriber for the service-request-events topic.

```
spring.cloud.stream.bindings.publishServiceRequest.destination=service-request-events
spring.cloud.stream.bindings.subscribeServiceRequest.destination=service-request-events
```

IMPORTANT | See the `com.camunda.poc.starter.kafka.integration` package/folder for impl of publishers and subscribers.

A single subscriber is implemented `KafkaEventSubscriber.java` ; it simply gets the message from the topic and serialized into memory. Then it saves/caches the Service Request into the local db based on the event type.

A single publisher is implemented `KafkaEventPublishingDelegate.java` , but other publishers may be needed in your use case. This publisher is also a Cmaunda JavaDelegte wired into the bpmn model and executed during the process execution. This is a powerful pattern as it provides control of the workflow execution and allows handling errors, incidents and more.

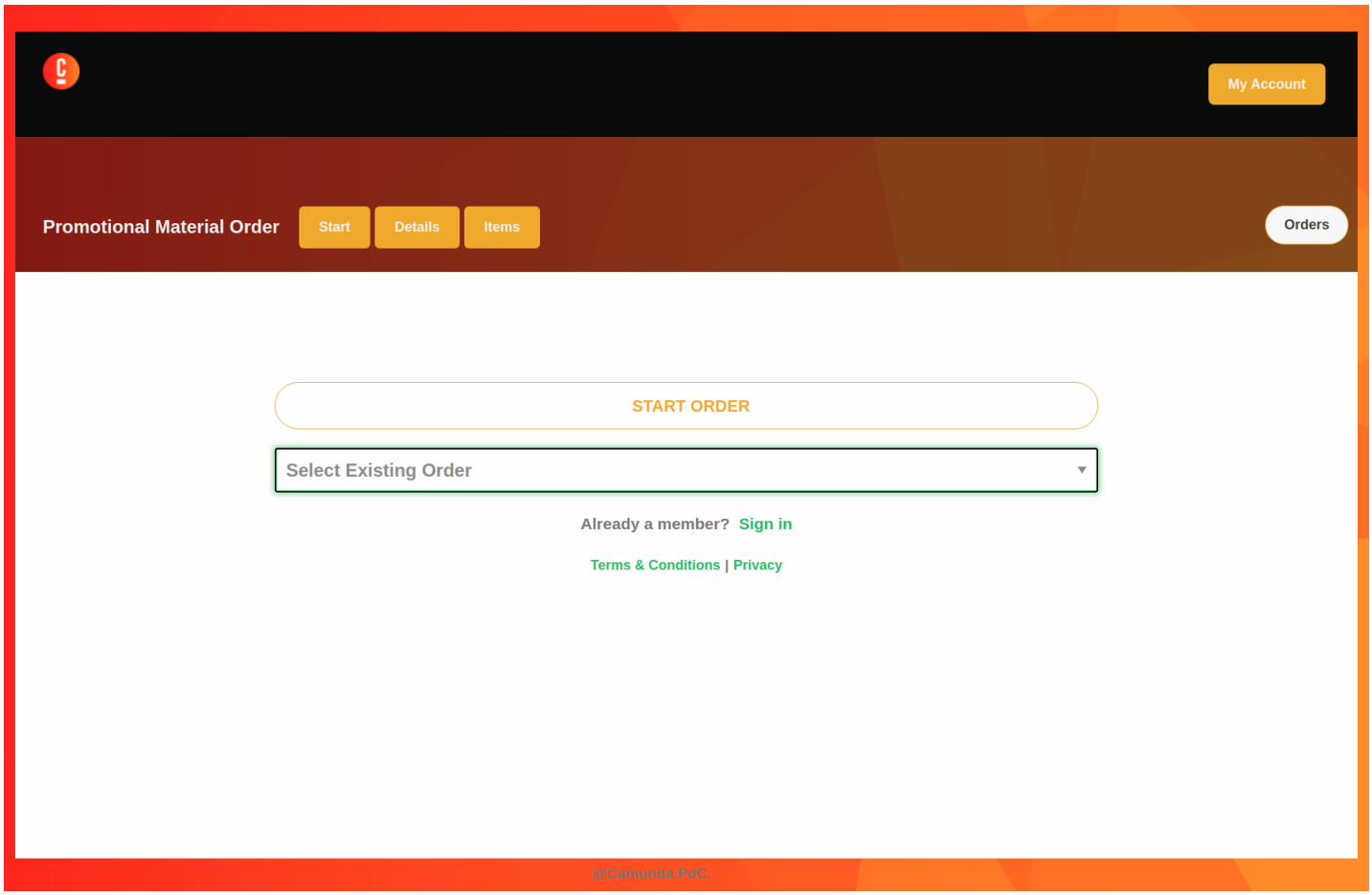
The publisher publishes an Event with event meta-data such as the Name and Type as well as parameters such as workflow state and business data.

The Event meta-data helps other components understand what to do with the event.

Camunda-Spring Event Listener Bridge

UI Integration

ReactJS Integration



The Maven frontend-maven-plugin configured in pom.xml is used to build the ReactJS app. The plugin creates a bundle.js file which ends up in `src/main/resources/static/built/bundle.js`. The static directory makes static resources such as JS and HTML available to the java app.

The Java application boot-straps the ReactJS App through Thymeleaf a java/spring frontend framework. The templates directory `src/main/resources/templates/app.html` has a HTML file app.html which calls the React app through a `<script />` tag loading the HTML into the react div `<div id="react"></div>`

Thymeleaf ties the Java frontend together using a Spring controller. `src/main/java/com/camunda/poc/starter/use-case/service.request/controller/ui/UIServicerequestController.java`. Mapping the app context to /sr and calling the app.html.

The React Components are organized under the `src\main\js\reactjs` folder into a use-case folder then subdivided by component.

Webpack and package.json define the structure and dependencies for the React App that allow and build the app into the bundle.js which is later placed in the static directory as explained previously. Allowing our frontend to load the plain EcmaScript.

Foundation CSS is used for styling https://get.foundation/sites/docs/index.html

NOTE	The react app is built in this manner, wrapped in spring-boot app, for convenience and consistency. This makes dev-ops, demoing and getting started easier and may not be appropriate for other technology stacks and dev-ops scenarios.

Handling Business Data and UI Integration

TIP	See the pattern described in the Camunda Best Practices Handling Data In Processes (https://camunda.com/best-practices/handling-data-in-processes/) and Enhancing Tasklist with Business Data (https://camunda.com/best-practices/enhancing-tasklists-with-business-data/)

Often and for numerous reasons we need to consolidate data from different sources. In this app I use JPA and Spring REST with some of springs features to build a custom API. Primarily for making integration with the UI easier. Here are few reasons why I take this approach.

- Reduce queries the UI does to the backend
- Make it easier to build UI components
- Create abstraction layer that can be used to integrate other technical and business requirements like reporting and security.
- Have a source of truth for process meta-data

NOTE	Also keep in mind I want to keep every-thing self contained for PoC purposes. Think in logical terms and that these components could be another technology or several other technologies depending on the specific needs.

Camunda REST API

Camunda has a powerful rest API. This code repository has many examples of interacting with the CAMUNDA REST API from the custom UI and using postman. See the `postman` folder in the project home.

Import the postman collection and take a look at the queries to start the workflow and correlate messages.

see more CAMUNDA REST API (https://docs.camunda.org/manual/latest/reference/rest/)

JPA Spring Data Repositories

A separate API and logically separate data-store is used to query order data.

We can guarantee the data is updated in the data store with the workflow. See the section above on the JavaDelegates that implement the publishing functionality.

Spring Data JPA (https://spring.io/blog/2011/02/10/getting-started-with-spring-data-jpa) is the technology used for business data. Spring Data allows for an easy way to create API's that are easy for a UI to query. Also an easy way to combine data into a useful form for the UI to consume.

Developing with this PoC Starter Project

Setting up React for Dev

Configure the api endpoint. This is the backend URI for the spring-boot server where the react app gets data

In the .env file in the project home directory change the environment variables to match the spring-boot server context.

NOTE If running the react app as a standalalone and not on localhost configure the API_HOST and API_POST environment vars as follows inserting your host and port for the spring-boot server. You should only need to do this if you cannot access the spring-boot server on localhost and you plan to run the React App standalone.

```
API_HOST=http://127.0.0.1
API_PORT=8080
API_ROOT=api
```

IMPORTANT You will need to use the cors profile in this setup and potentially modify the cors config in the spring-boot app.

Run node and server.js by starting a node server in the home directory of the project. You may need to run `npm install` first.

```
nodemon server.js
```

Run the web-pack watch in the project home so you can update the bundle as you modify reactjs

```
webpack -w
```

IMPORTANT You may need to install nodejs, nodemon and webpack depending on your environment setup

NOTE If you are developing only front-end components than you can user docker-compose and follow the React setup above. And simply use docker-compose to run the server

```
cd to the docker-compose directory and run


docker-compose up
```

Running the server for Dev

For development on the backed run the spring-boot app on command line

NOTE

You can enable spring-dev-tools to build front and back-end component in dev mode providing faster restarts and live-reload.

run the app in dev mode by uncommenting spring-dev-tools in pom.xml

WARNING

spring-dev-tools affects the way Camunda serializes objects into process vars and will cause serialization errors in some cases. So it is commented out in pom.xml by default.

run the following with the appropriate profiles

```
mvn spring-boot:run mvn spring-boot:run -Dspring.profiles.active=<use-case>,gui,email,cors
```

Other Tools

Mail Slurper

Use mail slurper consuming mail messages sent by the workflow. This is an easy way to test email integrations.

```
cd into /dev-tools/mailslurper-1.14.1-<dist>
```

```
execute ./mailslurper
```

Mail slurper is configured by editing the `/dev-tools/mailslurper-1.14.1-linux/config.json` The app is configured to use mail slurper in the `application-mail.properties`

Kafka

NOTE

A simple Kafka congfig is packaged into docker-compose. See `docker-compose` directory in the project home folder. Also you can run docker compose as follows.

```
docker-compose up
```

kafka image docs https://hub.docker.com/r/bitnami/kafka/

Use the downloaded Kafka Distro if you prefer. I have included the distro in the `kafka` folder in the project home. See kafka docs to run it.

NOTE

I use the consumer in the kafka distro during dev to see when messages are published.

```
./bin/kafka-console-consumer.sh --topic service-request-events --bootstrap-server localhost:9092
```

Version 1.0
Last updated 2021-05-10 19:06:21 -0600