Hello, Coherence Community Edition, Part 2: Building the client

How to develop React and JavaFX front-end clients that can work with a Coherence CE back-end application

by Aleks Seović

October 9, 2020

In the first article in this series, I showed how to implement a REST API that manages to-do list tasks stored in Oracle’s open source Coherence Community Edition (CE). Testing the REST API was done using a curl-based interface, but curl is not the most user-friendly interface on the planet.

This article will build on the back-end code from the previous article by implementing two clients: a React-based web UI (see Figure 1) and a desktop UI that uses JavaFX.

Figure 1. A user-friendly interface for the To Do List application

Client applications and Coherence CE
First, let’s cover a fundamental topic: How do client applications connect to Coherence CE and access data?

Coherence CE supports two types of clients: cluster member clients and remote clients.

The REST API implemented in the previous article is an example of a cluster member client, which can be either storage enabled or storage disabled. Those types of members are essentially identical, except storage-disabled members do not store any data locally.

This is what I alluded to in the last article when I mentioned that this project could separate the data store and the app server: I could’ve run Helidon web servers that serve the REST API as storage-disabled members, separate from the storage-enabled members that manage all the data. Sometimes that makes sense, because it provides better isolation between the two and it allows them to scale independently. For simplicity I’ve decided not to do that, at least for now.

The benefit of using cluster member clients is that they are fully aware of the cluster topology and partition assignment. This allows the clients to directly access any data object stored in Coherence CE, via a single network hop. The downside is that the clients need to be on the same high-speed, low-latency network as the rest of the cluster. Further, clients have the potential to destabilize the whole cluster if they start acting out and become unresponsive.

Remote clients, on the other hand, work a bit differently. They do not connect directly to all other cluster members but to a proxy server that is typically a storage-disabled member of a cluster.

Traditionally, Coherence supported two types of proxies: Coherence*Extend and the Coherence REST proxy. Now there is also a third type: Coherence gRPC.

**Coherence*Extend.** This is a proprietary, TCP-based RPC protocol that is used by Oracle’s existing Java, .NET, and C++ client implementations. Coherence*Extend has been around for a long time (since 2006), it is supported by many Oracle Coherence versions in a backward- and forward-compatible manner, and it has been proven in many mission-critical applications. On the other hand, Coherence*Extend is proprietary and synchronous, it doesn’t always play nicely with modern cloud deployments, and some of the clients implemented on top of it (.NET clients, in particular) are a bit old and do not use many of the latest features of the languages they support.

**The Coherence REST proxy.** This proxy is an implementation of a generic REST API that allows you to access from pretty much any platform or language data that is managed in Coherence CE. Unfortunately, it also comes with certain limitations due to the nature of REST and the underlying HTTP
protocol itself, it is not as full-featured as native clients, and it can be a bit cumbersome to configure.

Quite frankly, while the REST proxy had its purpose at a time, I really don't see much need or use for it any longer. It's just as easy, if not easier, to implement your own application-specific REST API that is free to use a full Java API, as in the previous article, and expose it either via Helidon integration (recommended) or via the built-in HTTP server.

**Coherence gRPC.** This is a third proxy introduced in the latest Coherence CE release (20.06).

Coherence gRPC is a proxy implementation that uses gRPC as a transport and is a viable alternative to Coherence*Extend. This proxy builds on top of Helidon gRPC Server, and it provides a number of immediate benefits: It works much better with modern cloud deployments, it is supported by various HTTP load balancers and Kubernetes ingress controllers, it is asynchronous, and the gRPC itself is supported by pretty much every relevant platform and language.

At the moment, Oracle offers only a native Java client for Coherence gRPC, which is what I’ll use to implement a JavaFX client in a bit, but with luck there will be a native Node.js/JavaScript client out soon followed by modern .NET and C++ clients as well as new Python, Golang, and Swift clients.

And if you want a client for another platform that is not officially supported, you will be able to write it yourself. The protocol buffer definitions for the services and messages Oracle supports are already publicly available, and you will be able to use existing client implementations as a guide when implementing your own.

The benefits of remote clients are that they don’t have to be on the same network, you can have as many of them as you need, they can come and go as they please without impacting cluster membership, and they can obviously be written in languages other than Java. The downside is that every request from the client has to go through the proxy, which adds an additional network hop and associated latency to each operation.

With that out of the way, it's time to implement the clients.

**Implementing a React client**

As I admitted in the previous article, I am not really a front-end developer, and the choice of React is somewhat accidental and mostly driven by what I’m (barely) familiar with. You could easily implement a similar front end using Angular, Vue.js, or any other popular front-end framework.

As a matter of fact, my colleague Tim Middleton has already implemented another web front end using Oracle JavaScript Extension Toolkit (Oracle JET), which uses the same idea of the
UI binding to a data model that is updated via events. If you are interested in it, the source code for the Oracle JET client is available on GitHub.

All of these frameworks have one thing in common: They allow you to use the standard Node.js development toolchain to build and test the UI, and once you are satisfied you can “compile” the application into a set of static HTML, JavaScript, and CSS files that can be served by any web server capable of serving static content.

Therefore, a sample application can use the same Helidon Web Server that serves the REST API to serve the static front end as well. This approach simplifies the application quite a bit; there is no separate server to deploy and manage, and there are no cross-origin resource sharing (CORS) issues to deal with, because both the front end and the REST API have the same origin.

**Setting up the React client**

To build the React client, I need to answer a few questions:

- Where is the source for the front end going to be?
- How will I build the front end as part of the existing Maven build?
- How will the “compiled” front end be packaged in order to make it available for Helidon to serve?

I’ll answer these questions in reverse order.

Helidon can serve static content either from the file system or from the classpath. The latter is much easier to manage, so I’ll package all the static files for the front end within the server JAR file and configure Helidon to serve the content from there by adding the `META-INF/microprofile-config.properties` file to the project:

```xml
<plugin>
    <groupId>com.seovic.maven.plugins</groupId>
    <artifactId>server-static-classpath</artifactId>
    <version>1.0.0</version>
</plugin>
```

That’s really all there is to it—Helidon has now been told to serve static content from the web directory in the classpath and to use `index.html` as a default welcome file.

How will I build the front end as part of the Maven build to ensure that the static content for the front end ends up where Helidon expects it? To do this, I’ll use `npm-maven-plugin` by yours truly, which embeds an npm-based build into a Maven build:

```xml
<plugin>
    <groupId>com.seovic.maven.plugins</groupId>
</plugin>
```
The code above will run the build script defined in package.json during the generate-resources phase of the Maven build, and it says the source code for the front end will be under the src/main/web directory within the server project. See Figure 2.

![Figure 2. Front-end application within the larger Maven project](image)

Note that the front-end application (and its developers) are completely unaware of the surrounding Maven structure. As far as they are concerned, the web directory highlighted above is the root of their JavaScript project.

Although that makes it easier for the front-end developers to do their job without having to learn Maven, it does mean that after the front end is built, it's necessary to copy generated static files into the structure Maven understands. Fortunately, this is trivial to do using the standard maven-resources-plugin artifact ID:
That's it. Now that the project structure is in place, it is time to proceed with the implementation.

Implementing the React client

Covering the complete front-end implementation is out of the scope of this article, because most of the UI-related code has nothing to do with Coherence CE and would quickly turn this article into a book chapter. The code is available on GitHub, along with the rest of the application, so feel free to clone the repository and check out the details if you are so inclined.

Here, the focus is on the state management aspect of the client application and on the interaction between the front-end app and the back-end REST API created in the previous article.

I will use Redux to manage state locally on the client. It's not the only option, but it does fit nicely into the event-driven architecture this application will implement. Redux uses actions that are dispatched to a reducer to update application state. Technically, Redux doesn't really update anything, because it treats state as immutable; rather, it creates a new state based on the existing state and the action received.

If this is all a bit unclear (I know it took me a while to grok it), look at the reducer for the client-side representation of a to-do list. With luck, that should make things a bit clearer.

```javascript
export default function todos(state = [], action) {
  switch (action.type) {
    case INIT_TODOS:
      // Implementation of the INIT_TODOS action
  }
}```
The function above defines a *reducer* for the state of the to-do list. Each case within the `switch` statement handles a different *action type* and returns a new state based on the current state and the *action payload* received. Whenever the state changes, the UI *reacts* to it (there is a reason the framework is called *React*) by refreshing itself accordingly.

What’s important to understand is that this state is local to the front-end app and at this point, it has nothing to do with the state being managed within the Coherence CE back end. To fix that and link the two together, it is necessary to do the following:

- Initialize the local state when the application is loaded
- Update the local state based on the events received from the server

Here’s the code to accomplish both of those tasks within the main *App.js* component of the React application:

```javascript
let initialized = false;

function init(actions) {
  actions.fetchAllTodos();
}
```
The `init` function above addresses both of those tasks. First it calls the `fetchAllTodos` action, which makes the call to the REST API and dispatches the results to the reducer:

```javascript
export const initTodos = (todos) => ({type: 'todos', payload: todos});

export function fetchAllTodos() {
  return (dispatch) => {
    request
      .get('/api/tasks')
      .end(function (err, res) {
        console.log(err, res);
        if (!err) {
          dispatch(initTodos(res.body));
        }
      });
  }
}
```

Next, it registers an event source with the event’s endpoint implemented on the server and handles each event by

```javascript
// register for server-side events
let source = new EventSource('/api/tasks/');

source.addEventListener("insert", (e) => {
  let todo = JSON.parse(e.data);
  actions.addTodo(todo.id, todo.createdAt);
});

source.addEventListener("update", (e) => {
  let todo = JSON.parse(e.data);
  actions.updateTodo(todo.id, todo.description);
});

source.addEventListener("delete", (e) => {
  let todo = JSON.parse(e.data);
  actions.deleteTodo(todo.id);
});

source.addEventListener("end", (e) => {
  console.log("end");
  source.close();

  initialized = true;
});
```

const App = ({todos, actions}) => {
  if (!initialized) {
    init(actions);
  }

  return (
    <div>
      <Header />
      <TodoInput addTodo={actions.addTodo}/>
      <MainSection todos={todos} actions={actions} />
    </div>
  )
};
dispatching it to the Redux reducer.

The actions themselves are divided into two groups: the ones that update local, Redux-managed state and the ones that make remote calls to the REST API to update the server-side state in Coherence CE.

The remaining local actions are similar to the `initTodos` action above and simply add the appropriate action type to the payload, so they can be applied by the reducer. For example, the `addTodo` action is defined as follows:

```javascript
export const addTodo = (id, createdAt, description) => {
  return {
    type: types.ADD_TODO
  }
}
```

The remote actions, on the other hand, simply make REST calls to the server without updating the Redux state directly. Instead, they rely on the event listeners registered earlier to apply any server-side state changes to the local state.

For example, the `addTodoRequest` action that is passed to the `TodoInput` component above simply sends the request and logs the response:

```javascript
export function addTodoRequest(text) {
  request
    .post('/api/tasks')
    .send({description: text})
    .end(function (err, res) {
      console.log(err, res);
    });
}
```

The actual task, in JSON format, is then received and added to the Redux state by the insert event handler defined earlier:

```javascript
source.addEventListener("insert", (e) => {
  let todo = JSON.parse(e.data);
  actions.addTodo(todo.id, todo.createdAt, todo.description);
});
```

This reactive, event-driven approach has two important consequences:

- It makes it easier to implement the actions that mutate the server-side state, because you have to worry only about sending the request to the server. You don't need to worry about synchronization of the server-side and client-side state.
- It makes the client UI react to the server-side state changes, regardless of which client made the change.
It should be obvious by now that this event-driven approach enabled by Coherence CE is significantly more efficient than what you would be able to implement with a majority of popular data stores, where you would likely have to rely on polling on the server or, worse, on the client to keep the UI up to date.

This concludes the React front-end implementation. Next, I’ll implement the JavaFX client.

Implementing a JavaFX client

To a large extent, the JavaFX client (see Figure 3) is very similar to the server-side REST API implementation from the previous article. It uses the same NamedMap API, it observes Coherence CE events, and many of the data access methods are exactly the same.

Figure 3. The JavaFX client

There are a few differences, however, which I’ll explain. But first, I need to set up the project.

Setting up the JavaFX client project

I will implement JavaFX client as another module within the Maven project, so the place to begin is with the client POM file:

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.oracle.coherence.examples</groupId>
  <artifactId>todo-list-client</artifactId>
  <version>1.0.0-SNAPSHOT</version>
  <properties>
    <coherence.groupId>com.oracle.coherence.coherence</coherence.groupId>
    <coherence.version>20.06</coherence.version>
  </properties>
```
There isn’t anything surprising or very interesting in the code above. Note that it includes Coherence Java Client and JSON serialization support, as well as the dependencies needed for JavaFX and for contexts and dependency injection (CDI) support.

However, this is not sufficient, because the client doesn’t have a proxy that will enable it to talk to the server side yet. To fix that, add the following dependencies to the server POM file:

```xml
<dependency>
    <groupId>${coherence.groupId}</groupId>
    <artifactId>coherence-grpc-proxy</artifactId>
    <version>${coherence.version}</version>
</dependency>
<dependency>
    <groupId>${coherence.groupId}</groupId>
    <artifactId>coherence-json</artifactId>
    <version>${coherence.version}</version>
</dependency>
```
To summarize, there are two new dependencies:

- The Coherence gRPC proxy, which contains the gRPC service implementation the gRPC client needs
- Coherence JSON, which is the same dependency as on the client and provides the JSON serialization support I want to use between the client and the server

That’s pretty much it. The Coherence gRPC proxy is built on top of Helidon gRPC Server, which will be added as a transitive dependency. Just like the Helidon Web Server, Helidon gRPC Server will be bootstrapped by the CDI at startup if it is present in the classpath, and any discovered gRPC services will be deployed automatically. There is nothing else to do, since the Helidon gRPC Server default configuration works just fine for this project’s purposes.

In addition to creating a Maven project, I need to enable CDI by creating a `META-INF/beans.xml` file:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://xmlns.jcp.org/xml/ns/javaee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee http://xmlns.jcp.org/xml/ns/javaee/pac"version="1.0"
bean-discovery-mode="annotated"/>

package com.oracle.coherence.examples.todo.client;

public class Task {
    private String id;
    private long createdAt;
    private String description;
    private Boolean completed;

    /**
     * Construct Task instance.
     *
     * @param description task description
     */
    public Task(String description)
There are really only two differences between the server-side and client-side implementation of the Task class. The first one is that the client class does not implement the Serializable interface, which doesn't really matter all that much. However, the second difference is important and needs to be discussed.

Although the two classes have the same set of fields and look very much alike, they are not the same class, because they are in different packages. This means I cannot use Java serialization for marshaling data between the client and the server, because that would require a shared data class to be used.

That's one reason for using JSON to marshal data between the client and the server: It allows me to deserialize compatible payloads into different classes on both ends of the pipe. The same would be true if I had chosen to use the Portable Object Format (POF), which might a better choice from a performance perspective, but I'm not really concerned about that in this sample application.

However, there is still one issue to be faced: Unlike the REST API implemented earlier, which can infer the class to use when deserializing the JSON payload from the strongly typed JAX-RS method signatures, Coherence CE's NamedMap<K,V> is a generic interface, which makes it impossible to infer the type. To solve that problem, Coherence JSON includes type information in the JSON payload during serialization by default, via the @class metaproperty.

For example, the JSON payload for a serialized server-side instance might look similar to this:

```json
{
    "@class": "com.oracle.coherence.examples.to.Serializable",
    "id": "a3f764",
    "completed": true,
    "createdAt": 1596105656378,
    "description": "Write an article"
}
```

Can you see the problem? The name of the class doesn't exist on the client embedded into the JSON payload. Fortunately, unlike Java serialization, Coherence JSON provides support for type aliasing. This makes it possible to register different classes on the server and the client under the same alias to make the JSON payload compatible with both.
To accomplish that, implement a `GensonBundleProvider` both within the client and on the server:

```java
public class JsonConfig implements GensonBundleProvider {
    @Override
    public GensonBundle provide() {
        return new GensonBundle()
            .public void configure(GensonBuild {
                builder.addAlias("Task", Task
            });
    }
}
```

Apart from the package name, which is not shown above, the classes are identical and use whichever `Task` implementation is available in the classpath.

Next, add the `com.oracle.coherence.io.json.GensonBundleProvider` file to the `META-INF/services` directory, so custom providers can be discovered by the service loader and configured with the content of `com.oracle.coherence.examples.todo.server.JsonConfig` on the server and with `com.oracle.coherence.examples.todo.client.JsonConfig` on the client.

Once the necessary JSON configuration is in place, the payload above becomes

```json
{
    "@type": "Task",
    "id": "a3f764",
    "completed": true,
    "createdAt": 1596105656378,
    "description": "Write an article"
}
```

And it can be deserialized on the client and on the server successfully using locally a registered implementation of the `Task` class.

Note that Coherence JSON uses an embedded, heavily customized version of the Genson JSON serializer. The embedded Genson serializer is in a different package to prevent conflicts in case the official Genson release is also used by an application.

For the most part, you shouldn’t have to care about this. As long as you annotate your data classes with either JSON-B or
Jackson annotations (when necessary), everything should just work, and you can use other JSON implementations for other purposes. As a matter of fact, once that's in place, the REST API uses the reference JSON-B implementation, Eclipse Yasson, which is brought in by Helidon.

With that out of the way, it's time to return to the client implementation. All of the important logic is within the TaskManager class:

```java
@ApplicationScoped
public class TaskManager
{
  /**
   * A {@link Filter} to retrieve completed
   */
  private static final Filter<Task> COMPLETED
      = Filters.equal("completed", true);

  /**
   * A {@link Filter} to retrieve active tasks.
   */
  private static final Filter<Task> ACTIVE
      = Filters.equal("completed", false);

  /**
   * Tasks map.
   */
  @Inject
  @Remote
  private NamedMap<String, Task> tasks;

  ...
}
```

The code above should look familiar: It defines static fields for the two filters that will be used in various queries and injects a NamedMap containing the tasks.

However, there is one important difference. To inject a NamedMap obtained by the Coherence Java gRPC client, I need to add the @Remote qualifier to the injection point. If I didn't do that, the Coherence CDI extension would inject the default NamedMap implementation and the JavaFX client would attempt to join the cluster, which is not what I want.

For the code above to work, I must configure the gRPC client to use the correct session.

The session defines which gRPC channel to use to connect to the server, as well as which serializer to use. So let's add the following application.yaml file to the src/main/resources directory within the client module:

```yaml
coherence:
  sessions:
    - name: default
```
serializer: json
channel: default

This file configures the client to use the JSON serializer and the default gRPC channel, so the client will attempt to connect to the gRPC server on localhost:1408.

That's fine at the moment, because that's precisely the setup I'll use for testing, but I'll have to add some additional configuration before the client can connect to the server once the server is deployed to Kubernetes, which is what will happen (sneak preview!) in the next article.

The good news is that because Helidon MP Config is used for configuration, I can easily override any of the values above (or add new ones) using system properties or environment variables. But let's leave that for the next article as well.

Note that the client is configured explicitly to use the JSON serializer, but I haven't done anything similar on the proxy. The good news is I don't have to. The proxy supports all available serialization formats, which are discovered by either CDI or the service loader, and it will use whichever serializer the client tells it to use.

In theory, the client could use a different format for each request, but in practice, each client will likely use the same format, as configured within the session above, for the duration of its connection to the proxy.

Now, let's continue with the TaskManager implementation and look at some of the data access methods:

```java
public void addTodo(String description) {
    Task todo = new Task(description);
    tasks.put(todo.getId(), todo);
}

public Collection<Task> getAllTasks() {
    return tasks.values();
}

public Collection<Task> getActiveTasks() {
    return tasks.values(ACTIVE);
}

public Collection<Task> getCompletedTasks() {
    return tasks.values(COMPLETED);
}

public void removeTodo(String id) {
    tasks.remove(id);
}
```
Everything above is very similar to the code implemented within the REST API. To implement basic create, read, update, delete (CRUD) operations against the tasks map, the code uses standard Map APIs such as put, remove, and values as well as NamedMap APIs such as invoke, invokeAll, and values overloads that accept a filter.

**Coherence aggregators.** There are two additional methods in TaskManager that use a Coherence CE feature I haven’t discussed yet: aggregators.

Coherence CE aggregators allow you to perform parallel, MapReduce-style aggregations across the cluster.

The example above uses a very basic Count aggregator, which simply returns the number of entries that satisfy the specified filter, but there are many other built-in aggregators that allow you to find minimum, maximum, and average values of a specific attribute, or even to group entries by some attribute and perform another aggregation within each group of entries. You can also implement your own custom aggregators.

The aggregators execute in parallel, which makes them very efficient and nearly linearly scalable, as long as they are implemented correctly. They will also automatically re-execute if there are cluster membership changes and data rebalancing, so the client application doesn’t have to worry about any of that.

In the example above, each member will determine the count of local entries that satisfy the specified criteria and return the partial result to the root aggregator executing on the client (or in
this case, on the gRPC proxy), which will then combine those partial results into the final result.

In this case, the final result is a scalar value, but it could be literally anything.

By the way, Coherence CE also provides a custom implementation of the Stream API introduced in Java 8, which is built on top of aggregators. When you use the Oracle Coherence Remote (see a video about it here), the stream pipeline definition will be sent to all cluster members using a custom aggregator, and it will be executed in parallel, not within a single JVM and across a handful of CPU cores but possibly across hundreds of JVMs and thousands of cores.

Finally, just as I had to observe Coherence CE events within the REST API and convert them into server-sent events (SSEs) that the web UI can consume, I need to do something similar here. The difference is that instead of converting Coherence CE MapEvents into SSE events, the code will convert them into standard CDI events. That way the JavaFX UI implementation can remain completely Coherence CE-agnostic and simply observe the CDI events as they are published:

```java
@Inject
private Event<TaskEvent> taskEvent;

/**
 * Convert Coherence map events to CDI events
 */
void onTaskEvent(@Observes @MapName("tasks")
{
    taskEvent.fire(new TaskEvent(event.getOld
}
```

That concludes the JavaFX client implementation. It's time to see if the new clients work as expected, letting you get rid of `curl` once and for all.

**Running the clients**

If you haven’t done so already, you should get the code for the example from GitHub because in this article I didn’t cover all the code that is necessary to run the application.

To access the React client, you must build and run the server, which is responsible for serving the front-end application and the REST API the front end depends on.

In the previous article, I was able to simply run the server within the IDE. But now you have to build the server using Maven to build the front end and package it into the server JAR file. As long as you have the latest version of Node.js and npm installed, this can be easily accomplished by simply running `mvn install` within the server directory.
You will first need to run `npm install` within the `server/src/main/web` directory to install the necessary front-end dependencies. You need to do this only once.

You can then run the server in the IDE, just as you did before, or from the command line by running `mvn exec:exec`. If everything goes well, you should see the server start, and after a few seconds, you should see the same Helidon log message mentioned in the first article:

```
2020.08.11 03:16:00 INFO io.helidon.microprof
```

You can now access the React front end (see Figure 4) by simply navigating to http://localhost:7001/, as the log message above suggests.

![Tasks](image)

**Figure 4.** The React client UI

Feel free to play with the application and create some tasks. If you want to make it more interesting, open the application in multiple browser windows and see how all of them are kept in sync via events as you make changes.

Finally, start the JavaFX client by running `mvn install` in the client directory, and then run `mvn javafx:run`. You should see a UI similar to what's shown in Figure 5.
Aleks Seović

Aleks Seović (@aseovic) is an architect at Oracle, where he works on Oracle Coherence, a leading in-memory data grid product, and contributes to the Helidon microservices framework. Most recently, Seović led the design and implementation of the Helidon gRPC framework, as well as CDI and Eclipse MicroProfile support in Coherence. He currently leads the implementation of Coherence native clients, GraphQL support, and Spring integration. Prior to joining Oracle in 2016, Seović led a boutique consultancy practice, where he worked with customers around the world to help them implement mission-critical applications on top of Coherence. Seović is the author of Oracle Coherence 3.5 (Packt Publishing, 2010) and frequently speaks about and evangelizes Coherence at industry conferences, Java and .NET user group events, and Coherence SIGs.
The initial list of tasks will depend on the tasks you've created earlier using the React client. Just as before, add some tasks and change the existing ones, and see how both applications stay in sync as you make changes.

Alternatively, if you don’t feel like running the code yourself, you can watch this video.

Coherence 12.2.1 - Support for the Java 8 Stream API

Conclusion

That's it for this article, and it was a long one. The result is a To Do List application that you can run locally. That's nice, but I'll be the first to admit that it's not all that useful.

In the third and final article, I'll turn this toy demo application into a highly available production application that is deployed to a Kubernetes cluster, can be easily scaled out, and can be monitored using Prometheus, Grafana, and Jaeger.

Dig deeper