Building the Example Element: A Complete Walkthrough

Your Game Code - Adding Custom Elements · https://namazustudios.com/docs/custom-code/element-example-complete-walkthrough/

A complete tour of the Example Element project, from a blank checkout to a running backend with a custom REST API and a dashboard plugin.

Embedded media: https://www.youtube.com/embed/TqkvpwvRJEc?feature=oembed&enablejsapi=1

What You'll Build

The Example Element is a reference Custom Element for Namazu Elements 3.8. Once running locally, it gives you:

- A REST API mounted at `/element/example/rest/api` with an open probe endpoint, an authenticated endpoint that greets the logged-in user, and a POST/PUT demo resource.
- A Guice-wired service layer that injects the SDK's own `UserService`.
- A dashboard UI plugin (React, built as a standalone bundle) that appears in the Elements admin dashboard sidebar.
- A packaged `.elm` archive you can deploy to any Elements instance.

This guide covers setup end-to-end, then breaks down every Maven module and every source file so you understand exactly what each piece does and why it's there.

Prerequisites

1. Java 21 (JDK)
2. Apache Maven

3. Docker with Docker Compose (used to run a local MongoDB replica set)
4. Git
5. Node.js (the build can also fetch its own pinned Node v22.14.0 via a Maven profile — see the Maven deep dive below — but a system Node install is the fastest path for local iteration)

> Note
>
> Since Elements is a Java 21 project, we recommend IntelliJ as your IDE. See the platform-specific setup guides (Windows, Mac, Linux) if you haven't set up a local Elements development environment before.

Step by Step: Setup to Running

1. Clone the Repository

```
git clone https://github.com/NamazuStudios/element-example.git
cd element-example
```

The project is a four-module Maven build:

```
element-example/
├── api/          # Exported interfaces (other Elements depend on this)
├── ui/           # TypeScript/React UI plugin source (Vite; not deployed directly)
├── element/      # Implementation module — builds the .elm archive
├── debug/        # Local development runner (not deployed)
└── services-dev/ # Docker Compose services (MongoDB) for local dev
```

2. Build Everything

```
mvn install
```

This compiles `api` and `element` in dependency order, packages `element` into a `.elm` archive, and installs everything (including the classified API jar and the `.elm` artifact) into your local Maven repository. The `ui` module's real npm build is skipped unless you pass `-Pbuild-ui` — see the Maven deep dive below for why.

3. Start MongoDB

```
docker compose -f services-dev/docker-compose.yml up -d
```

This starts a single-node MongoDB 6.0.9 instance configured as replica set `local-test` on port 27017, plus a one-shot `rs-init` sidecar container that waits for Mongo to accept connections and then runs `rs.initiate(...)` to actually form the replica set. Elements' core SDK requires a replica set (it relies on transactions/change streams), so a plain standalone `mongod` will not work.

4. Run the Element Locally

Run `debug/src/main/java/run.java` from your IDE, or from the command line:

```
mvn -pl debug exec:java -Dexec.mainClass=run
```

Note

The working directory must be the project root (`element-example/`), because `run.java` shells out to `npm` in `ui/` and `docker compose` in `services-dev/` using relative paths. In IntelliJ: Run → Edit Configurations → set the working directory to the project root.

When it runs, `run.java` does the following, in order:

1. If `ui/node_modules` doesn't exist yet, runs `npm install` in `ui/` (first run only).
2. Runs `npm run build` in `ui/`, which builds both the superuser and user dashboard plugin bundles and writes them directly into `element/src/main/ui/superuser/` and `element/src/main/ui/user/`.
3. Runs `docker compose up -d` in `services-dev/` (the same command as step 3 — safe to run again).
4. Builds an `ElementsLocalBuilder` from the SDK's local runtime, configured to build and deploy `com.example.element:element:elm:1.0-SNAPSHOT` from source.
5. Calls `local.start()` — this triggers the Maven build of the `element` module (picking up the freshly built UI bundles), then boots the full local Elements runtime with your Element deployed.
6. Calls `local.run()`, which blocks and serves requests until you stop the process.

After a short startup you'll see log output indicating the Elements runtime is listening, by default on port 8080.

5. Verify It's Working

Hit the open probe endpoint — no authentication required:

```
curl http://localhost:8080/element/example/rest/api/helloworld
# => Hello world!
```

Hit the authenticated endpoint without a session — you'll be greeted as a guest:

```
curl http://localhost:8080/element/example/rest/api/hellowithauthentication
# => Hello, Guest!
```

Create a user, log in to get a session secret, then call it again with the `Elements-SessionSecret` header — see Creating a User and User Authentication in Elements if you need a refresher on those two calls. With a valid session you should get `Hello, <your name>!` instead of `Hello, Guest!`.

Explore the generated OpenAPI spec (this Element's routes appear alongside the rest of the platform's, tagged `Example`):

```
http://localhost:8080/api/rest/openapi.json
```

Finally, log in to the dashboard at `http://localhost:8080/admin/login` as a superuser and look for "Example Element" in the sidebar — that's the React plugin bundle shipped from `ui/src/superuser/ExamplePlugin.tsx`, served from this Element's UI content tree.

Maven Structure Deep Dive

Root `pom.xml`

The root is a pure aggregator (`<packaging>pom</packaging>`) with no parent of its own. It declares the four modules, pins shared properties, and centralizes dependency versions and scopes:

```
<modules>
    <module>api</module>
    <module>ui</module>
    <module>element</module>
```

```
    <module>debug</module>
</modules>

<properties>
    <maven.compiler.source>21</maven.compiler.source>
    <maven.compiler.target>21</maven.compiler.target>
    <elements.version>3.8.14</elements.version>
    <api.classifier>${project.groupId}.api</api.classifier>
    <!-- swagger.version, guice.version, rs.api, jakarta.websocket.version,
         crossfire.version, servlet.api, logback.version also declared here -->
</properties>
```

The important piece is `dependencyManagement`, which imports the Elements SDK BOM:

```
<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>dev.getelements.elements</groupId>
            <artifactId>sdk-bom</artifactId>
            <version>${elements.version}</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>

        <dependency>
            <groupId>com.example.element</groupId>
            <artifactId>api</artifactId>
            <version>${project.version}</version>
            <scope>provided</scope>
        </dependency>

        <dependency>
            <groupId>com.example.element</groupId>
            <artifactId>api</artifactId>
            <version>${project.version}</version>
            <classifier>${api.classifier}</classifier>
            <scope>provided</scope>
        </dependency>
    </dependencies>
</dependencyManagement>
```

Importing `sdk-bom` means every child module gets the correct version **and** the correct scope for every SDK artifact automatically — child `pom.xml` files never declare a `<version>` or `<scope>` for an SDK dependency. The `api` artifact is declared twice: once as a plain dependency (used at compile time by `element`) and once with the `${api.classifier}` classifier (a second, classified jar of the same code — see below).

The `api` Module: a Classified Jar

`api/pom.xml` depends on nothing but the core SDK (scope `provided`) — by design. The API module should stay as lean as possible, since every API jar in a deployment shares a common classpath with every other Element's API jar; bloating it with third-party libraries invites classpath conflicts across unrelated Elements.

The interesting part is its build configuration:

```
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-jar-plugin</artifactId>
    <executions>
        <execution>
            <id>classified-jar</id>
            <phase>package</phase>
            <goals><goal>jar</goal></goals>
            <configuration>
                <classifier>${api.classifier}</classifier>
            </configuration>
        </execution>
    </executions>
</plugin>
```

This produces a **second** jar — `api-1.0-SNAPSHOT-com.example.element.api.jar` — in addition to the normal jar. The plain jar is what `element` compiles against; the classified jar is what gets copied into the `api/` directory inside the final `.elm` archive, which is how other deployed Elements can depend on and resolve this Element's exported interfaces at runtime.

The `element` Module: Dependencies

`element/pom.xml` declares no versions or scopes — every dependency's scope comes from the imported BOM:

```
<dependencies>
    <dependency> <!-- own API, classified jar -->
        <groupId>com.example.element</groupId>
        <artifactId>api</artifactId>
        <classifier>${api.classifier}</classifier>
    </dependency>

<dependency><groupId>dev.getelements.elements</groupId><artifactId>sdk</artifactId></dependency>
    <dependency><groupId>dev.getelements.elements</groupId><artifactId>sdk-
model</artifactId></dependency>
    <dependency><groupId>dev.getelements.elements</groupId><artifactId>sdk-
service</artifactId></dependency>
    <dependency><groupId>dev.getelements.elements</groupId><artifactId>sdk-spi-
guice</artifactId></dependency>
    <dependency><groupId>dev.getelements.elements</groupId><artifactId>sdk-jakarta-
rs</artifactId></dependency>
    <dependency><groupId>com.google.inject</groupId><artifactId>guice</artifactId></dependency>
    <dependency><groupId>jakarta.ws.rs</groupId><artifactId>jakarta.ws.rs-
api</artifactId></dependency>
    <dependency><groupId>jakarta.websocket</groupId><artifactId>jakarta.websocket-
api</artifactId></dependency>
    <dependency><groupId>io.swagger.core.v3</groupId><artifactId>swagger-
annotations</artifactId></dependency>
    <dependency><groupId>io.swagger.core.v3</groupId><artifactId>swagger-jaxrs2-
jakarta</artifactId></dependency>
</dependencies>
```

The BOM scopes `sdk`, `sdk-model`, `sdk-service`, `sdk-spi-guice`, and `sdk-jakarta-rs` as `provided` — the Elements runtime already has these on its classpath, so they're compiled against but never bundled. `guice`, the Jakarta RS/WebSocket APIs, and the Swagger artifacts are also provided by the runtime. This Element only uses the Guice SPI (`sdk-spi-guice`) — there's no non-Guice `sdk-spi` variant in use here.

Note

Only artifacts that are **not** `provided` get bundled into the `.elm`'s `lib/` directory. Always double-check the BOM's scoping if you see duplicate-class errors at runtime — it usually means something that should be `provided` got bundled twice.

The `.elm` Packaging Pipeline

Unlike older Elements projects that used the Maven Assembly Plugin to zip a `libs/` + `classpath/` directory pair, this project builds its `.elm` archive with the Maven Antrun Plugin and Maven Dependency Plugin, no assembly descriptor required. Two properties define the staging layout:

```
<elm.staging.dir>${project.build.directory}/${project.groupId}.${project.artifactId}-
${project.version}</elm.staging.dir>
<elm.element.dir>${elm.staging.dir}/${project.groupId}.${project.artifactId}</elm.element.dir>
```

i.e. `target/com.example.element.element-1.0-SNAPSHOT/com.example.element.element/`. Five executions build up that directory and then zip it, in this order:

1. `elm-copy-api-deps` (`maven-dependency-plugin`, phase `prepare-package`) — copies this project's own `${api.classifier}`-classified jar (and any transitive ones sharing the same groupId) into `<elm.element.dir>/api`.
2. `elm-copy-lib-deps` (`maven-dependency-plugin`, phase `prepare-package`) — copies every non-`provided`-scope dependency into `<elm.element.dir>/lib`, prepending the group id to each filename.
3. `elm-stage-classpath` (`maven-antrun-plugin`, phase `prepare-package`) — copies compiled classes and `src/main/resources` into `<elm.element.dir>/classpath`.
4. `elm-stage-static-content` (`maven-antrun-plugin`, phase `prepare-package`) — copies `src/main/static` into `<elm.element.dir>/static` and `src/main/ui` into `<elm.element.dir>/ui`.
5. `elm-write-manifest` (`maven-antrun-plugin`, phase `prepare-package`) — resolves `git rev-parse HEAD` (falling back to `unknown`) and writes `dev.getelements.element.manifest.properties` at the element root with `Element-Version`, `Element-Build-Time`, `Element-Revision`, and `Element-Builtin-Spis`.
6. `elm-create-archive` (`maven-antrun-plugin`, phase `package`) — ensures `api/`, `lib/`, and `classpath/` exist even if empty, then zips the whole staging directory to `<elm.staging.dir>.elm`: `<zip destfile="${elm.staging.dir}.elm" basedir="${elm.staging.dir}"/>`.

Finally, `build-helper-maven-plugin`'s `attach-artifact` goal (phase `package`) attaches that `.elm` file to the Maven project as artifact type `elm`:

```
<plugin>
    <groupId>org.codehaus.mojo</groupId>
    <artifactId>build-helper-maven-plugin</artifactId>
    <executions>
        <execution>
            <id>attach-elm</id>
            <phase>package</phase>
            <goals><goal>attach-artifact</goal></goals>
            <configuration>
                <artifacts>
                    <artifact>
                        <file>${elm.staging.dir}.elm</file>
                        <type>elm</type>
                    </artifact>
                </artifacts>
            </configuration>
        </execution>
    </executions>
</plugin>
```

This is what makes `mvn install` publish `com.example.element:element:1.0-SNAPSHOT:elm` into your local repository alongside the normal jar — exactly the coordinate `run.java` references with `.elmArtifact("com.example.element:element:elm:1.0-SNAPSHOT")`, and the coordinate you'd reference from a deployment configuration.

The resulting `.elm` archive layout:

```
com.example.element.element/
  api/            <- classified API jar(s)
  lib/            <- bundled runtime jars (non-provided scope)
  classpath/      <- compiled classes + src/main/resources
  static/         <- src/main/static
  ui/             <- src/main/ui (superuser/user plugin bundles)
  dev.getelements.element.manifest.properties
```

The `debug` Module

`debug/pom.xml` intentionally depends on nothing but the SDK's local runtime and logging:

```
<dependencies>
    <dependency><groupId>dev.getelements.elements</groupId><artifactId>sdk-
local</artifactId><scope>compile</scope></dependency>
    <dependency><groupId>dev.getelements.elements</groupId><artifactId>sdk-local-
maven</artifactId><scope>compile</scope></dependency>
    <dependency><groupId>dev.getelements.elements</groupId><artifactId>sdk-
logback</artifactId><scope>compile</scope></dependency>
    <dependency><groupId>ch.qos.logback</groupId><artifactId>logback-
classic</artifactId><scope>compile</scope></dependency>
</dependencies>
```

The module's own comment sums up its purpose: `sdk-local` is a thin wrapper around a fully configured instance of Namazu Elements, and this configuration should almost never need changes. It's never deployed — it exists purely so you can boot the whole platform, with your Element loaded from source, inside your IDE.

The `ui` Module

`ui/pom.xml` is packaged `pom` and does no work by default. Its `build-ui` profile uses `frontend-maven-plugin` to install a pinned Node version and run the npm build, for CI/release builds that shouldn't depend on the machine's own Node install:

```
<profile>
    <id>build-ui</id>
    <build>
        <plugins>
            <plugin>
                <groupId>com.github.eirslett</groupId>
                <artifactId>frontend-maven-plugin</artifactId>
                <version>1.15.0</version>
                <configuration>
                    <nodeVersion>v22.14.0</nodeVersion>
                </configuration>
```

```
                <executions>
                    <execution><id>install-node-and-npm</id><goals><goal>install-node-and-
npm</goal></goals></execution>
                    <execution><id>npm-install</id><goals><goal>npm</goal></goals></execution>
                    <execution>
                        <id>npm-build</id>
                        <goals><goal>npm</goal></goals>
                        <configuration><arguments>run build</arguments></configuration>
                    </execution>
                </executions>
            </plugin>
        </plugins>
    </build>
</profile>
```

Activate it with `mvn install -Pbuild-ui`. Day to day, developers run `npm` directly in `ui/` (or let `run.java` do it) using their own Node install — this profile exists for build environments that shouldn't assume Node is already present.

Java Source Deep Dive

All source lives under `com.mystudio.mygame`, split across the `api` and `element` modules. Here's every file, in the order you'd read them to understand how the pieces connect.

1. `package-info.java` — Declaring the Element

`element/src/main/java/com/mystudio/mygame/package-info.java`:

```
// Required annotation for an Element. Will recursively search folders
// from this point to include classes in the Element if recursive is true.
// Otherwise, you must include additional package-info.java files in child packages.
@ElementDefinition(recursive = true)
// Enables DI via Guice
@GuiceElementModule(MyGameModule.class)
// Allows injecting DAO layer from Elements Core
@ElementDependency("dev.getelements.elements.sdk.dao")
// Allows injecting Service layer from Elements Core
```

```
@ElementDependency("dev.getelements.elements.sdk.service")
package com.mystudio.mygame;

import com.mystudio.mygame.guice.MyGameModule;
import dev.getelements.elements.sdk.annotation.ElementDefinition;
import dev.getelements.elements.sdk.annotation.ElementDependency;
import dev.getelements.elements.sdk.spi.guice.annotations.GuiceElementModule;
```

- `@ElementDefinition(recursive = true)` is what makes the SDK's classloading/discovery mechanism recognize `com.mystudio.mygame` — and every sub-package, because `recursive = true` — as one Element. Without it, nothing here would be treated as an Element at all. If `recursive` were `false`, each sub-package (`rest`, `service`, `model`, `guice`) would need its own `package-info.java`.
- `@GuiceElementModule(MyGameModule.class)` tells the SDK which Guice module to install when it bootstraps this Element's private injector.
- The two `@ElementDependency` annotations declare dependencies on other Elements — the core SDK's DAO layer and service layer. This is what lets `GreetingServiceImpl` (below) `@Inject` the SDK's own `UserService` even though this Element never binds it itself.

---

2. `guice/MyGameModule.java` — the Guice Module

```
package com.mystudio.mygame.guice;

import com.google.inject.PrivateModule;
import com.mystudio.mygame.service.GreetingService;
import com.mystudio.mygame.service.GreetingServiceImpl;

public class MyGameModule extends PrivateModule {

    @Override
    protected void configure() {

        bind(GreetingService.class).to(GreetingServiceImpl.class);

        expose(GreetingService.class);
    }
}
```

- It extends `PrivateModule`, not plain `AbstractModule`. Elements gives each Element its own private Guice environment so internal bindings can't leak into, or collide with, other Elements' bindings.
- `bind(GreetingService.class).to(GreetingServiceImpl.class)` is a standard interface-to-implementation binding.
- `expose(GreetingService.class)` is required precisely **because** this is a `PrivateModule` — nothing inside one is visible outside it by default. Without this call, the service-locator lookup in `HelloWithAuthentication` (below) would fail.

---

3. `service/GreetingService.java` — the API Interface

Lives in the **api** module (`api/src/main/java/com/mystudio/mygame/service/GreetingService.java`), not `element`, so other Elements can depend on the interface without pulling in the implementation or Guice wiring:

```
package com.mystudio.mygame.service;

public interface GreetingService {

    /**
     * Attempts to fetch the current user for the session header and return an appropriate
greeting
     * @return The greeting based on if a logged-in user is found
     */
    String getGreeting();

}
```

Note that the interface itself carries no `@ElementServiceExport` annotation — in this project that annotation is applied to the implementation class instead.

---

4. `service/GreetingServiceImpl.java` — the Implementation

```
package com.mystudio.mygame.service;
```

```
import dev.getelements.elements.sdk.annotation.ElementServiceExport;
import dev.getelements.elements.sdk.model.user.User;
import dev.getelements.elements.sdk.service.user.UserService;
import jakarta.inject.Inject;


@ElementServiceExport(GreetingService.class)
public class GreetingServiceImpl implements GreetingService {

    private UserService userService;

    public UserService getUserService() {
        return userService;
    }

    @Inject
    public void setUserService(final UserService userService) {
        this.userService = userService;
    }

    @Override
    public String getGreeting() {
        // Because we set the dev.getelements.elements.auth.enabled attribute to "true" in the
HelloWorldApplication,
        // the UserService will be automatically injected with the current user. This will apply
an authentication
        // filter to every request and every service that is used in this application.
        final User currentUser = userService.getCurrentUser();
        final boolean isLoggedIn = !User.Level.UNPRIVILEGED.equals(currentUser.getLevel());
        final String name = isLoggedIn ? currentUser.getName() : "Guest";

        return "Hello, " + name + "!";
    }
}
```

- `@ElementServiceExport(GreetingService.class)` tells the SDK to expose this concrete class under the `GreetingService` service-locator key — the annotation-driven counterpart to the explicit `bind()` / `expose()` calls in `MyGameModule`. Both point at the same binding.

- `setUserService` is Guice **setter injection** (`@Inject` on a setter rather than a constructor) — a common pattern for SDK-provided dependencies.
- `UserService` comes from `dev.getelements.elements.sdk.service.user` — resolvable here only because `package-info.java` declared `@ElementDependency("dev.getelements.elements.sdk.service")`.
- The auth check is: fetch `currentUser`, then compare its level against `User.Level.UNPRIVILEGED` to tell a real logged-in user from an anonymous/guest request. **This is the only auth-level check in the project**, and it distinguishes "logged in vs. not" — it does not check for `SUPERUSER`.

---

5. `HelloWorldApplication.java` — Registering Endpoints

```
package com.mystudio.mygame;

import com.mystudio.mygame.rest.ExampleContent;
import com.mystudio.mygame.rest.HelloWithAuthentication;
import com.mystudio.mygame.rest.HelloWorld;
import dev.getelements.elements.sdk.annotation.ElementDefaultAttribute;
import dev.getelements.elements.sdk.annotation.ElementServiceExport;
import dev.getelements.elements.sdk.annotation.ElementServiceImplementation;
import io.swagger.v3.jaxrs2.integration.resources.OpenApiResource;
import jakarta.ws.rs.core.Application;

import java.util.Set;

@ElementServiceImplementation
@ElementServiceExport(Application.class)
public class HelloWorldApplication extends Application {

    @ElementDefaultAttribute("true")
    public static final String AUTH_ENABLED = "dev.getelements.elements.auth.enabled";

    @ElementDefaultAttribute("/element/example/rest/api")
    public static final String RS_ROOT = "dev.getelements.elements.element.rs.root";

    @ElementDefaultAttribute("/element/example/ws")
    public static final String WS_ROOT = "dev.getelements.elements.element.ws.root";

    @ElementDefaultAttribute("/app/static/test/path")
```

```
    public static final String STATIC_CONTENT_URI = "dev.getelements.element.static.uri";

    @ElementDefaultAttribute("/app/ui/test/path")
    public static final String UI_CONTENT_URI = "dev.getelements.element.ui.uri";

    public static final String OPENAPI_TAG = "Example";

    /**
     * Here we register all the classes that we want to be included in the Element.
     */
    @Override
    public Set<Class<?>> getClasses() {
        return Set.of(
                //Endpoints
                HelloWorld.class,
                HelloWithAuthentication.class,
                ExampleContent.class,

                // Exposes the default security rules for the API. Assumes you are using the
builtin Elements auth
                // system by setting `dev.getelements.elements.auth.enabled` to true in the
annotation above.
                OpenAPISecurityConfig.class

        );
    }

}
```

- `@ElementServiceImplementation` marks this as a concrete implementation the SDK should instantiate and manage, rather than a plain POJO.
- `@ElementServiceExport(Application.class)` is the key wiring annotation — it exposes this class under the JAX-RS `Application` service-locator key, which is how the Elements HTTP runtime discovers which `Application` subclass to mount for this Element, and at what root path.
- Each `@ElementDefaultAttribute` field declares a default value for a named configuration attribute — the string constant is the attribute key, the annotation value is its default. `AUTH_ENABLED` defaulting to `"true"` is what turns on the built-in Elements auth filter for every request and service in this Element, which is exactly what makes `UserService.getCurrentUser()` populated in

`GreetingServiceImpl`. `RS_ROOT` and `WS_ROOT` set the REST and WebSocket mount points; `STATIC_CONTENT_URI` and `UI_CONTENT_URI` are example placeholder paths, not wired to real content.

- `getClasses()` is the actual JAX-RS registration point. Note that `OpenApiResource` is imported but never added to the returned set in this project's current form — only `HelloWorld`, `HelloWithAuthentication`, `ExampleContent`, and `OpenAPISecurityConfig` are registered.

---

6. `OpenAPISecurityConfig.java` — Documenting the Auth Scheme

```
package com.mystudio.mygame;

import dev.getelements.elements.sdk.model.Headers;
import io.swagger.v3.oas.annotations.ExternalDocumentation;
import io.swagger.v3.oas.annotations.OpenAPIDefinition;
import io.swagger.v3.oas.annotations.info.Contact;
import io.swagger.v3.oas.annotations.info.Info;
import io.swagger.v3.oas.annotations.security.SecurityRequirement;
import io.swagger.v3.oas.annotations.security.SecurityScheme;
import io.swagger.v3.oas.annotations.security.SecuritySchemes;

import static dev.getelements.elements.sdk.jakarta.rs.AuthSchemes.SESSION_SECRET;
import static io.swagger.v3.oas.annotations.enums.SecuritySchemeIn.HEADER;
import static io.swagger.v3.oas.annotations.enums.SecuritySchemeType.APIKEY;

@OpenAPIDefinition(
        info = @Info(
                title = "Example Element",
                description = "An example element.",
                contact = @Contact(
                        url = "https://namazustudios.com",
                        email = "info@namazustudios.com",
                        name = "Namazu Studios"
                )
        ),
        externalDocs = @ExternalDocumentation(
                url = "https://namazustudios.com/docs",
                description = "Please see the Namazu Elements Manual for more information."
        ),
        security = {
```

```
                @SecurityRequirement(name = SESSION_SECRET)
        }
)
@SecuritySchemes({
        @SecurityScheme(
                type = APIKEY,
                in = HEADER,
                name = SESSION_SECRET,
                paramName = SESSION_SECRET,
                description = "Session secret required for authenticated endpoints")
})
public class OpenAPISecurityConfig {}
```

This class has no fields or methods — it exists solely to carry class-level OpenAPI annotations, and is registered in `getClasses()` purely so Swagger's scanner picks them up when generating the API spec. It defines the `Elements-SessionSecret` header as an API-key-style security scheme and sets it as the document-wide default. It plays **no role at request-processing time** — it's documentation metadata only, not enforcement.

---

7. `rest/HelloWorld.java` — an Open Probe Endpoint

```
package com.mystudio.mygame.rest;

import io.swagger.v3.oas.annotations.Operation;
import io.swagger.v3.oas.annotations.tags.Tag;
import jakarta.ws.rs.Consumes;
import jakarta.ws.rs.GET;
import jakarta.ws.rs.Path;
import jakarta.ws.rs.Produces;
import jakarta.ws.rs.core.MediaType;

import static com.mystudio.mygame.HelloWorldApplication.OPENAPI_TAG;

@Tag(name = OPENAPI_TAG)
@Path("/helloworld")
public class HelloWorld {

    @GET
```

```
    @Produces(MediaType.TEXT_PLAIN)
    @Operation(summary = "Hello world probe", description = "Returns a simple greeting")
    public String sayHello() {

        return "Hello world!";

    }

}
```

The simplest possible JAX-RS resource — a plain `GET` returning static text, useful as a health-probe. It declares no `@SecurityRequirement`, so it's reachable even though this Element enables auth globally: authorization here works by *services* resolving the current user (or not) rather than a filter rejecting unauthenticated requests outright.

---

8. `rest/HelloWithAuthentication.java` — the Service Locator Pattern

```
package com.mystudio.mygame.rest;

import com.mystudio.mygame.service.GreetingService;
import dev.getelements.elements.sdk.Element;
import dev.getelements.elements.sdk.ElementSupplier;
import io.swagger.v3.oas.annotations.Operation;
import io.swagger.v3.oas.annotations.security.SecurityRequirement;
import io.swagger.v3.oas.annotations.tags.Tag;
import jakarta.ws.rs.Consumes;
import jakarta.ws.rs.GET;
import jakarta.ws.rs.Path;
import jakarta.ws.rs.Produces;
import jakarta.ws.rs.core.MediaType;

import static com.mystudio.mygame.HelloWorldApplication.OPENAPI_TAG;
import static dev.getelements.elements.sdk.jakarta.rs.AuthSchemes.SESSION_SECRET;


@Tag(name = OPENAPI_TAG)
@Path("/hellowithauthentication")
public class HelloWithAuthentication {
```

```
    private final Element element = ElementSupplier
            .getElementLocal(HelloWithAuthentication.class)
            .get();

    private final GreetingService greetingService = element
            .getServiceLocator()
            .getInstance(GreetingService.class);


    @GET
    @Produces(MediaType.TEXT_PLAIN)
    @Consumes(MediaType.TEXT_PLAIN)
    @Operation(
            summary = "Greeting with login check",
            description = "Checks if the session token in the header corresponds to at least a
USER level user.",
            security = { @SecurityRequirement(name = SESSION_SECRET) }
    )
    public String sayHelloWithAuth() {
        return greetingService.getGreeting();
    }

}
```

This is the class to study for the service-locator pattern:

- `ElementSupplier.getElementLocal(HelloWithAuthentication.class).get()` resolves the `Element` instance associated with the calling class's classloader. This is necessary because JAX-RS resources are instantiated by the Jakarta RS container, not by Guice — so they can't use `@Inject` directly. This static lookup is how a plain, container-instantiated resource reaches back into its own Element's private Guice injector.
- `element.getServiceLocator().getInstance(GreetingService.class)` then pulls the singleton out of that injector. This only works because `MyGameModule` called `expose(GreetingService.class)` — remove that call and this lookup throws.
- `@SecurityRequirement(name = SESSION_SECRET)` on the `@Operation` is, again, OpenAPI documentation metadata — it tells Swagger this endpoint expects the header, but the actual "logged in or guest" branching happens inside `GreetingServiceImpl`, not in a JAX-RS filter defined here.

9. `rest/ExampleContent.java` — POST/PUT and Path Params

```
package com.mystudio.mygame.rest;

import com.mystudio.mygame.model.ExamplePostRequest;
import com.mystudio.mygame.model.ExamplePostResponse;
import com.mystudio.mygame.model.ExamplePutRequest;
import com.mystudio.mygame.model.ExamplePutResponse;
import io.swagger.v3.oas.annotations.Operation;
import io.swagger.v3.oas.annotations.tags.Tag;
import jakarta.ws.rs.*;
import jakarta.ws.rs.core.MediaType;

import java.util.Map;

import static com.mystudio.mygame.HelloWorldApplication.OPENAPI_TAG;

@Tag(name = OPENAPI_TAG)
@Path("/examplecontent")
public class ExampleContent {

    @POST
    @Consumes(MediaType.APPLICATION_JSON)
    @Produces(MediaType.APPLICATION_JSON)
    @Operation(summary = "Example POST request", description = "Example produces/consumes for
POST")
    public ExamplePostResponse examplePost(ExamplePostRequest examplePostRequest) {

        //Normally we'd create a new object in the database with a POST request, but for
demonstration
        //purposes, we'll just return an example response object
        final var response = new ExamplePostResponse();

        response.setName(examplePostRequest.getName());

        return response;
    }

    @PUT
```

```
    @Consumes(MediaType.APPLICATION_JSON)
    @Produces(MediaType.APPLICATION_JSON)
    @Operation(summary = "Example PUT request", description = "Example produces/consumes for PUT")
    public ExamplePutResponse examplePost(ExamplePutRequest examplePutRequest) {

        //Normally we'd overwrite an existing object in the database with a PUT request, but for
demonstration
        //purposes, we'll just return an example response object
        final var response = new ExamplePutResponse();

        response.setName(examplePutRequest.getName());

        return response;
    }

    @PUT
    @Path("{name}")
    @Consumes(MediaType.APPLICATION_JSON)
    @Produces(MediaType.APPLICATION_JSON)
    @Operation(summary = "Example PUT request with a path param", description = "Example
produces/consumes for PUT with a path param")
    public ExamplePutResponse examplePutWithPathParam(@PathParam("name") String name,
ExamplePutRequest examplePutRequest) {

        //Normally we'd overwrite an existing object in the database with a "name" property that
matches the "name" path
        // param with this PUT request, but for demonstration purposes, we'll just return an
example response object
        final var response = new ExamplePutResponse();

        response.setName(examplePutRequest.getName());
        response.setMetadata(Map.of("name", name));

        return response;
    }
}
```

This resource never touches a database — its own comments say so explicitly. It exists to demonstrate JSON request/response bodies, a validated request DTO, and a `@PathParam`. One of the four request/response DTOs, `ExamplePutResponse` (in `model/`), shows the shape all of them share:

```
package com.mystudio.mygame.model;

import dev.getelements.elements.sdk.model.Constants;
import io.swagger.v3.oas.annotations.media.Schema;
import jakarta.validation.constraints.NotNull;
import jakarta.validation.constraints.Pattern;

import java.util.Map;

@Schema
public class ExamplePutResponse {

    @NotNull
    @Pattern(regexp = Constants.Regexp.NO_WHITE_SPACE)
    @Schema(description = "A unique name for the object that we're creating. No spaces allowed.")
    private String name;

    @Schema(description = "The type of request being made. For example/debugging purposes.")
    private String requestType = "ExamplePutResponse";

    @Schema(description = "Any additional information to return.")
    private Map<String, Object> metadata;

    // getters/setters omitted for brevity
}
```

`@Schema` annotations document the field for OpenAPI generation, and `@NotNull` / `@Pattern` (using the SDK's own `Constants.Regexp.NO_WHITE_SPACE`) are standard Jakarta Bean Validation constraints, enforced automatically by the JAX-RS runtime before the resource method body even runs.

What's Not Demonstrated in This Repo

To avoid sending you looking for code that isn't there, note explicitly what this example does **not** include, even though these are all valid Elements SDK capabilities described elsewhere in the manual:

- No WebSocket endpoint (no `@ServerEndpoint` class), despite `HelloWorldApplication.WS_ROOT` declaring a base path for one. See WebSockets for the general pattern.
- No Morphia `@Entity`/DAO classes — nothing in this project persists to MongoDB directly.
- No `User.Level.SUPERUSER` check — the only level check is `UNPRIVILEGED` vs. logged-in.
- No use of `ElementRegistry` for cross-Element service discovery — `GreetingService` is only ever resolved locally.

The Debug Runner: `run.java`

```
import dev.getelements.elements.sdk.local.ElementsLocalBuilder;

import java.io.File;
import java.io.IOException;

/**
 * Runs your local Element in the SDK.
 *
 * Working directory must be the project root (element-example/).
 * IntelliJ: Run → Edit Configurations → Working directory → set to this project root.
 */
public class run {
    public static void main(final String[] args ) throws IOException, InterruptedException {

        // Install npm dependencies on first run, then build both segment bundles.
        // The bundles are written directly to element/src/main/ui/{superuser,user}/
        // so that the Maven build triggered by local.start() picks them up.
        final var uiDir = new File("ui");

        if (!new File(uiDir, "node_modules").exists()) {
            new ProcessBuilder("npm", "install")
                    .directory(uiDir)
                    .inheritIO()
                    .start()
                    .waitFor();
        }
```

```
        new ProcessBuilder("npm", "run", "build")
                .directory(uiDir)
                .inheritIO()
                .start()
                .waitFor();

        new ProcessBuilder("docker", "compose", "up", "-d")
                .directory(new File("services-dev"))
                .inheritIO()
                .start()
                .waitFor();

        final var local = ElementsLocalBuilder.getDefault()
                .withSourceRoot()
                .withDeployment(builder -> builder
                        .useDefaultRepositories(true)
                        .elementPackage()
                        .elmArtifact("com.example.element:element:elm:1.0-SNAPSHOT")
                        .endElementPackage()
                        .build()
                )
                .build();

        local.start();
        local.run();

    }

}
```

Note it's a bare, package-less class — the Javadoc comment doubles as its usage instructions. The four steps (npm install/build, docker compose, then `ElementsLocalBuilder`) were walked through in the setup section above; the key API points here are `.withSourceRoot()` (build against the local source tree, not a published artifact) and `.elmArtifact("com.example.element:element:elm:1.0-SNAPSHOT")` (the exact Maven coordinate, including the `elm` packaging type, produced by the `element` module's `attach-artifact` step described above).

Dashboard UI Plugin Deep Dive

Elements can inject custom pages into the Elements admin dashboard by shipping a React component bundle alongside the Java code. The dashboard discovers these at runtime via a `plugin.json` manifest — no dashboard changes required. The `ui/` module is a Vite/TypeScript project that builds these bundles.

Source Layout

```
ui/
├── package.json
├── vite.base.config.ts       # shared dev-server / library-build config factory
├── vite.superuser.config.ts  # createConfig('superuser')
├── vite.user.config.ts       # createConfig('user')
├── tsconfig.json
├── tailwind.config.ts
├── postcss.config.ts
└── src/
    ├── dev.css                # Tailwind + light/dark tokens, dev shell only
    ├── superuser/
    │   ├── ExamplePlugin.tsx  # the component shown in the dashboard
    │   ├── plugin-entry.ts    # registers the component with window.__elementsPlugins
    │   ├── dev-entry.tsx      # mounts the component for standalone dev (not shipped)
    │   └── index.html         # dev server entry point (not shipped)
    └── user/                  # same four files, simpler component
```

Build Scripts

`ui/package.json`:

```
{
  "scripts": {
    "dev:superuser": "vite --config vite.superuser.config.ts",
    "dev:user": "vite --config vite.user.config.ts",
    "build": "vite build --config vite.superuser.config.ts && vite build --config
vite.user.config.ts"
  }
}
```

React and its dev tooling are listed only under `devDependencies` — there's no runtime `dependencies` block. That's deliberate: the built bundle never ships its own copy of React.

The Dual-Mode Vite Config

`vite.base.config.ts` exports a `createConfig(segment)` factory used by both `vite.superuser.config.ts` and `vite.user.config.ts`. It branches on Vite's `command`:

```
export function createConfig(segment: string) {
  return defineConfig(({ command }) => {
    if (command === 'serve') {
      // Standalone dev server with HMR: npm run dev:superuser / dev:user
      // API calls are proxied to a running Elements instance (override with ELEMENTS_URL).
      const elementsUrl = process.env.ELEMENTS_URL ?? 'http://localhost:8080'
      return {
        plugins: [react({ jsxRuntime: 'classic' })],
        root: `src/${segment}`,
        server: { proxy: { '/api': elementsUrl, '/app': elementsUrl } },
      }
    }

    // Library/IIFE build: npm run build
    return {
      esbuild: { jsx: 'transform', jsxFactory: 'React.createElement', jsxFragment:
'React.Fragment' },
      build: {
        lib: {
          entry: `src/${segment}/plugin-entry.ts`,
          name: 'ElementPlugin',
          formats: ['iife'],
          fileName: () => 'plugin.bundle.js',
        },
        outDir: `../element/src/main/ui/${segment}`,
        emptyOutDir: false,
        minify: false,
        rollupOptions: {
          external: ['react'],
          output: { globals: { react: 'window.React' } },
        },
      },
```

```
    }
  })
}
```

- In `serve` mode, Vite runs a normal dev server with hot-module-reload against the segment's own root, proxying `/api` and `/app` to a running Elements instance so relative fetches work regardless of the dev server's port.
- In build mode, `outDir` resolves to `element/src/main/ui/{segment}` — this is the mechanism that gets the bundle into the right place for the antrun `elm-stage-static-content` step to pick up later. `emptyOutDir: false` so the build never deletes the `plugin.json` file sitting next to it.
- `external: ['react']` plus `globals: { react: 'window.React' }` rewrites every `import React from 'react'` into `var React = window.React` in the compiled IIFE — the bundle never embeds its own React, it shares the host dashboard's instance.
- `minify: false` is intentional — the shipped bundles are left readable.

The Plugin Component

`ui/src/superuser/ExamplePlugin.tsx` fetches an unauthenticated platform endpoint and renders the result:

```
import React from 'react'

interface VersionInfo {
  version: string
  revision: string
  timestamp: string
}

export function ExamplePlugin() {
  const [info, setInfo] = React.useState<VersionInfo | null>(null)
  const [loading, setLoading] = React.useState(false)
  const [error, setError] = React.useState<string | null>(null)

  async function fetchVersion() {
    setLoading(true)
    setError(null)
    try {
```

```
      const res = await fetch('/api/rest/version')
      if (!res.ok) throw new Error(`${res.status} ${res.statusText}`)
      setInfo(await res.json())
    } catch (e) {
      setError(e instanceof Error ? e.message : String(e))
    } finally {
      setLoading(false)
    }
  }

  return (
    <div className="p-6 max-w-2xl">
      <h1 className="text-2xl font-bold mb-2">Example Element</h1>
      <p className="text-muted-foreground mb-6">
        This page is served from the Example Element&rsquo;s superuser UI content directory.
      </p>
      <button onClick={fetchVersion} disabled={loading}>
        {loading ? 'Loading…' : 'Get Platform Version'}
      </button>
      {error && <div>{error}</div>}
      {info && (
        <div>
          <div>Version: {info.version}</div>
          <div>Revision: {info.revision}</div>
          <div>Built: {info.timestamp}</div>
        </div>
      )}
    </div>
  )
}
```

The `user` segment's version is simpler — a static informational panel with no fetch call at all. Both are wired to the dashboard's plugin registry the same way, in `plugin-entry.ts`:

```
import { ExamplePlugin } from './ExamplePlugin'

declare const window: Window & {
  __elementsPlugins?: {
    register(route: string, component: unknown): void
  }
```

```
}

window.__elementsPlugins?.register('example-element', ExamplePlugin)
```

Note

If your Element's REST endpoint requires authentication, the platform's convention is to send `window.__elementsApiClient.getSessionToken()` as an `Elements-SessionSecret` header on the fetch call (cookies alone aren't reliable in every dashboard context). This example's own fetch call only hits the unauthenticated `/api/rest/version` platform endpoint, so it doesn't exercise that pattern — if you want to call `/element/example/rest/api/hellowithauthentication` from a plugin, you'll need to add that header yourself, following the same shape shown for server-side auth elsewhere in this guide.

The `plugin.json` Manifest

After `npm run build`, the compiled bundle lands next to a manifest at `element/src/main/ui/superuser/plugin.json` (and the equivalent under `user/`):

```
{
  "schema": "1",
  "entries": [
    {
      "label": "Example Element",
      "icon": "Package",
      "bundlePath": "plugin.bundle.js",
      "route": "example-element"
    }
  ]
}
```

`label` is the sidebar text, `icon` is any Lucide icon name, `bundlePath` is relative to the manifest, and `route` is the unique key used both in the dashboard URL (`/plugin/{route}`) and in the `.register(route, ...)` call above — the two must match. Both the manifest and the built `plugin.bundle.js` get staged into the `.elm`'s `ui/` tree by the `elm-stage-static-content` antrun execution described earlier, and served at `/app/ui/{element-prefix}/{segment}/`.

Static & UI Content Serving

Two source directories are copied verbatim into the `.elm` archive by the Maven build, with no extra configuration required:

- `element/src/main/static/` — served at `/app/static/{prefix}/`
- `element/src/main/ui/` — served at `/app/ui/{prefix}/` (this is where the dashboard plugin bundles live)

Serving behavior for both trees — index file, custom routing rules, response headers, error pages — is controlled by the `StaticRuleEngine` reading attributes from the Element's configuration:

- `dev.getelements.static.index` / `dev.getelements.ui.index` — file served at the tree's context root (default `index.html`)
- `dev.getelements.static.rule.<name>.regex` / `dev.getelements.ui.rule.<name>.regex` — a regex rule matching file paths
- `dev.getelements.static.rule.<name>.header.<Header>.value` / the equivalent `ui` key — a response header template for matched files (supports `$filename`, `$path`, `$[0]`, `$[N]`)
- `dev.getelements.static.error.<code>` / the equivalent `ui` key — file served for a given HTTP error code
- `dev.getelements.element.static.uri` / `dev.getelements.element.ui.uri` — override the full serve URI, default `/app/static/{prefix}` / `/app/ui/{prefix}`

These attributes are normally embedded in the deployed `.elm` at `dev.getelements.element.attributes.properties` (element root, same level as `api/`, `lib/`, `classpath/`). This example project does not currently ship that file — if you add one, place it at `element/src/main/elm/dev.getelements.element.attributes.properties` and add an antrun copy step to `element/pom.xml` alongside the existing `elm-stage-*` executions to stage it into `${elm.element.dir}`.

Next Steps

- Custom Code Overview — the broader picture of how Elements load and isolate custom code
- Introduction to Guice and Jakarta in Elements — background on the two frameworks used throughout this example

- Structuring Your Element — a shorter, pattern-focused companion to this guide
- Packaging an Element with Maven — more detail on the `.elm` packaging mechanism