Building the Kotlin Example Element: A Complete Walkthrough

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

A complete tour of the Kotlin Example Element project, from a blank checkout to a running backend with a custom REST API and a dashboard plugin — written entirely in **Kotlin** (with one Java file, and there's a good reason for that). If you're looking for the Java version of this same walkthrough, see Building the Example Element: A Complete Walkthrough; the two projects share the same module layout and REST/Guice patterns; the differences called out here are specifically the Kotlin ones.

Prefer video? Watch the Kotlin quickstart walkthrough here:

Embedded media: https://www.youtube.com/embed/6kLWRMex-ug?feature=oembed&enablejsapi=1

What You'll Build

The Kotlin Example Element is a reference Custom Element for Namazu Elements 3.8, structured identically to the Java example but implemented in Kotlin. Once running locally, it gives you:

- A 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 (every command below was run and verified against a live local instance), then breaks down every Maven module and every source file — Kotlin, and the one Java 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, with the bundled Kotlin plugin enabled (it is by default in current IntelliJ releases). 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-kotlin.git
cd element-example-kotlin
```

The project is a four-module Maven build, the same shape as the Java example:

```
element-example-kotlin/
├── 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 initiates the replica set. Elements' core SDK relies on multi-document transactions, which require a replica set — a plain standalone `mongod` will not work.

4. Run the Element Locally

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

```
mvn -pl debug exec:java -Dexec.mainClass=Run -Dexec.classpathScope=compile
```

Notes on the Run Command

`debug/pom.xml` has no `exec-maven-plugin` binding of its own, so both flags on the command line matter. `-Dexec.mainClass=Run` is required because `run.kt` is a top-level file (no wrapping class) annotated `@file:JvmName("Run")`, which is what makes the compiled facade class `Run` rather than the default `RunKt`. `-Dexec.classpathScope=compile` is also required: the SDK's local-runtime deployment classes this entrypoint calls into (e.g. `dev.getelements.elements.sdk.deployment.*`) come from artifacts scoped `provided`, and `exec:java`'s default classpath scope is `runtime`, which excludes `provided` dependencies — running with the default scope fails with a `NoClassDefFoundError`. The working directory must also be the project root (`element-example-kotlin/`), because `run.kt` 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.kt` 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/app/rest/example-element/helloworld
# => Hello world!
```

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

```
curl http://localhost:8080/app/rest/example-element/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!`.

Try the POST/PUT demo resource:

```
curl -X POST http://localhost:8080/app/rest/example-element/examplecontent
  -H "Content-Type: application/json" -d '{"name":"test-name"}'
# => {"name":"test-name","requestType":"ExamplePostResponse","metadata":null}

curl -X PUT http://localhost:8080/app/rest/example-element/examplecontent/pathname
  -H "Content-Type: application/json" -d '{"name":"test-name"}'
# => {"name":"test-name","requestType":"ExamplePutResponse","metadata":{"name":"pathname"}}
```

Note

Every endpoint in this project mounts under `/app/rest/example-element/` — the path is built from the platform's standard `/app/rest/{prefix}` convention using the `dev.getelements.elements.app.serve.prefix` attribute (default `example-element`, set in `HelloWorldApplication`), not from the `RS_ROOT`/`WS_ROOT` constants also declared there. Those two constants set `dev.getelements.elements.element.rs.root`/`...ws.root`, which is separate configuration and does not change the externally observed mount path — don't be misled by their default values (`/element/example/api`, `/element/example/ws`) into expecting the API there.

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 — including a Kotlin-specific one — 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>
    <kotlin.version>2.1.0</kotlin.version>
    <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>

<build>
    <pluginManagement>
        <plugins>
            <plugin>
                <groupId>org.jetbrains.kotlin</groupId>
                <artifactId>kotlin-maven-plugin</artifactId>
                <version>${kotlin.version}</version>
                <configuration>
                    <jvmTarget>21</jvmTarget>
                </configuration>
            </plugin>
        </plugins>
    </pluginManagement>
</build>
```

The `kotlin-maven-plugin` version and `jvmTarget` are pinned once here in `pluginManagement` so every child module's declaration of the plugin (they all need one, since Kotlin isn't a default Maven language) inherits the same version and target without repeating it. The rest of `dependencyManagement` imports the Elements SDK BOM, exactly as in the Java example, plus one extra managed dependency:

```
<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>

        <dependency>
            <groupId>org.jetbrains.kotlin</groupId>
            <artifactId>kotlin-stdlib</artifactId>
            <version>${kotlin.version}</version>
        </dependency>
    </dependencies>
</dependencyManagement>
```

The managed `kotlin-stdlib` dependency has no `<scope>` pinned here — each module scopes it independently (`provided` in `api` and `debug` 's own declarations are unscoped/compile, matching what each module actually needs), which is why you'll see the scope vary slightly module to module below.

The `api` Module: a Classified Kotlin Jar

`api/pom.xml` depends on nothing but the core SDK and the Kotlin standard library (both 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. In this project, `api` 's entire source is a single dependency-free Kotlin interface (see the source deep dive below) — it doesn't even need the SDK to compile.

The build configuration points the source directory at Kotlin and runs the Kotlin compiler, then produces the same "classified jar" as the Java example:

```
<build>
    <sourceDirectory>${project.basedir}/src/main/kotlin</sourceDirectory>
    <plugins>
        <plugin>
            <groupId>org.jetbrains.kotlin</groupId>
            <artifactId>kotlin-maven-plugin</artifactId>
            <executions>
                <execution>
                    <id>compile</id>
                    <goals><goal>compile</goal></goals>
                </execution>
            </executions>
        </plugin>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-jar-plugin</artifactId>
            <executions>
                <execution>
                    <id>classified-jar</id>
                    <phase>package</phase>
                    <goals><goal>jar</goal></goals>
                    <configuration>
                        <classifier>${api.classifier}</classifier>
                    </configuration>
                </execution>
            </executions>
        </plugin>
    </plugins>
</build>
```

This produces a **second** jar — `api-1.0-SNAPSHOT-com.example.element.api.jar` — in addition to the normal jar, exactly like the Java example. The plain jar is what `element` compiles against; the classified jar is what gets copied into the `api/` directory inside the final `.elm` archive.

The `element` Module: Dependencies

`element/pom.xml` declares the same dependency set as the Java example, plus `kotlin-stdlib` (unscoped — it's needed at both compile and runtime here, unlike in `api`):

```
<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>
    <dependency><groupId>org.jetbrains.kotlin</groupId><artifactId>kotlin-
stdlib</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. Because `kotlin-stdlib` is **not** `provided` here, it ends up in the `.elm`'s `lib/` directory — the Elements runtime itself has no reason to ship Kotlin's standard library, so this Element must bring its own.

Compiling Kotlin and Java Together

This is the one build detail with no equivalent in the Java example. Almost everything in `element` is Kotlin, but `package-info.java` (below) has to stay Java, because package-level annotations use `ElementType.PACKAGE`, a target Kotlin does not support. `element/pom.xml` handles this with two separate compiler executions, run in a specific order:

```
<plugin>
    <groupId>org.jetbrains.kotlin</groupId>
    <artifactId>kotlin-maven-plugin</artifactId>
    <executions>
        <execution>
            <id>compile</id>
            <goals><goal>compile</goal></goals>
            <configuration>
                <sourceDirs>
                    <sourceDir>${project.basedir}/src/main/kotlin</sourceDir>
                    <sourceDir>${project.basedir}/src/main/java</sourceDir>
                </sourceDirs>
            </configuration>
        </execution>
    </executions>
</plugin>

<!--
    Compiles Java sources (e.g. package-info.java) after Kotlin compilation.
    package-info.java is required for package-level annotations such as @ElementDefinition,
    which use ElementType.PACKAGE — a target that Kotlin does not natively support.
-->
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-compiler-plugin</artifactId>
    <executions>
        <execution>
            <id>default-compile</id>
            <phase>none</phase>
        </execution>
        <execution>
            <id>java-compile</id>
            <phase>compile</phase>
```

```
            <goals><goal>compile</goal></goals>
            <configuration>
                <sourceDirs>
                    <sourceDir>${project.basedir}/src/main/java</sourceDir>
                </sourceDirs>
            </configuration>
        </execution>
    </executions>
</plugin>
```

- The Kotlin plugin's `compile` execution is given **both** source directories. It compiles the actual Kotlin files and, since `package-info.java` references `MyGameModule` (a Kotlin class) in `@GuiceElementModule(MyGameModule.class)`, Kotlin needs to see that Java file exists during its own compilation pass to resolve the reference — but Kotlin's compiler does not generate a `.class` file for `package-info.java` itself.
- The Maven Compiler Plugin's built-in `default-compile` execution is disabled (`<phase>none</phase>`) so it doesn't try to recompile everything with plain `javac`. A second, explicit execution named `java-compile` is bound to the same `compile` phase, restricted to just `src/main/java` — in practice, just `package-info.java`.
- Because the Kotlin plugin is declared first in `<plugins>`, it runs before the compiler plugin within the same `compile` phase. By the time `java-compile` runs `javac` against `package-info.java`, `MyGameModule.class` already exists on the output classpath, so the reference resolves.

Note

If you add more Java files to your own Kotlin Element, this is the pattern to keep: Kotlin source under `src/main/kotlin`, any package-level-annotation-only Java under `src/main/java`, and both compiler executions pointed at the right directories in the right order. You almost never need more Java than a single `package-info.java`.

The `.elm` Packaging Pipeline

Like the Java example, this project builds its `.elm` archive with the Maven Antrun Plugin and Maven Dependency Plugin — no assembly descriptor. The staging properties are identical:

```
<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/`. The Maven Dependency Plugin runs **three** `copy-dependencies` executions here, not two — the extra one is Kotlin-specific:

1. `elm-copy-api-deps` — copies this project's own `${api.classifier}`-classified jar into `<elm.element.dir>/api`.
2. `elm-copy-kotlin-stdlib-api` — copies every `org.jetbrains.kotlin`-groupId dependency (i.e. `kotlin-stdlib`) into `<elm.element.dir>/api` as well, alongside the API jar.
3. `elm-copy-lib-deps` — copies every non-`provided`-scope dependency into `<elm.element.dir>/lib`, prepending the group id to each filename (this also picks up `kotlin-stdlib` a second time, for `element`'s own runtime classpath).

```
<execution>
    <id>elm-copy-kotlin-stdlib-api</id>
    <phase>prepare-package</phase>
    <goals><goal>copy-dependencies</goal></goals>
    <configuration>
        <outputDirectory>${elm.element.dir}/api</outputDirectory>
        <includeGroupIds>org.jetbrains.kotlin</includeGroupIds>
        <prependGroupId>true</prependGroupId>
    </configuration>
</execution>
```

Why kotlin-stdlib Goes in api/ Too

The `api/` directory's classes load in a separate, shared classloader visible to every deployed Element (that's the whole point of exporting an API). If your Element exports Kotlin types — data classes, enums, sealed classes — through its classified API jar, and the platform or another Element reflects over those types at startup (Swagger scanning response bodies is the common case), that shared API classloader needs `kotlin-stdlib` on it too, or you get `NoClassDefFoundError: kotlin/jvm/internal/Intrinsics` at startup. This project's own `api` module (`GreetingService`) doesn't actually need this — it's a bare interface with no Kotlin-specific runtime dependency — but the

execution is harmless to keep (it only adds about 1 MB) and saves you from rediscovering this the hard way the first time you add a Kotlin data class to your own `api` module.

The remaining antrun executions (stage classpath, stage static/UI content, write the manifest, zip the archive) and the `build-helper-maven-plugin` `attach-artifact` step are identical to the Java example — see Packaging an Element with Maven for the full breakdown of that shared mechanism. The resulting archive layout is the same, plus the extra `kotlin-stdlib` jar under `api/`:

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

This is exactly the coordinate `run.kt` references with `.elmArtifact("com.example.element:element:elm:1.0-SNAPSHOT")`, and the coordinate you'd reference from a deployment configuration or `mvn deploy`.

The `debug` Module

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

```
<build>
    <sourceDirectory>${project.basedir}/src/main/kotlin</sourceDirectory>
    <plugins>
        <plugin>
            <groupId>org.jetbrains.kotlin</groupId>
            <artifactId>kotlin-maven-plugin</artifactId>
            <executions>
                <execution>
                    <id>compile</id>
                    <goals><goal>compile</goal></goals>
                </execution>
            </executions>
```

```
        </plugin>
    </plugins>
</build>

<dependencies>
    <dependency><groupId>org.jetbrains.kotlin</groupId><artifactId>kotlin-
stdlib</artifactId><scope>compile</scope></dependency>
    <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 comments sum 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 (or via `mvn exec:java`, per the run command above).

The `ui` Module

`ui/pom.xml` is identical in shape to the Java example's — it's a TypeScript/Vite project, so the backend language of the Element it's paired with is irrelevant to it. It's packaged `pom` and does no work by default; its `build-ui` profile uses `frontend-maven-plugin` to install a pinned Node version (v22.14.0) and run the npm build, for CI/release environments that shouldn't depend on the machine's own Node install. Activate it with:

```
mvn install -Pbuild-ui
```

Day to day, developers run `npm` directly in `ui/` (or let `run.kt` do it) using their own Node install — this profile exists for build environments that shouldn't assume Node is already present. See the Dashboard UI Plugin Deep Dive section below for the full build pipeline.

Kotlin Source Deep Dive

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

1. `package-info.java` — Declaring the Element (the One Java File)

`element/src/main/java/com/mystudio/mygame/package-info.java` — note the path: `src/main/java`, not `src/main/kotlin`:

```
// 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;
```

This file is byte-for-byte the same idea as the Java example's — it just can't be Kotlin. Kotlin has no `package-info.kt` equivalent because package-level annotations require `ElementType.PACKAGE` as a target, which the Kotlin annotation model doesn't support. Every Kotlin Element project needs this one Java file (see the Maven deep dive above for how the build compiles it alongside the Kotlin sources).

- `@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.

- `@GuiceElementModule(MyGameModule.class)` tells the SDK which Guice module to install when it bootstraps this Element's private injector — pointing at the Kotlin class below.
- 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.kt` — 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

class MyGameModule : PrivateModule() {

    override fun configure() {
        bind(GreetingService::class.java).to(GreetingServiceImpl::class.java)
        expose(GreetingService::class.java)
    }

}
```

- Same pattern as the Java example — `PrivateModule`, not plain `AbstractModule` — just Kotlin's `: PrivateModule()` constructor-call syntax and `::class.java` in place of Java's `.class`.
- `expose(GreetingService::class.java)` 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.kt` — the API Interface

Lives in the **api** module (`api/src/main/kotlin/com/mystudio/mygame/service/GreetingService.kt`), and is the *entire* contents of that module:

```
package com.mystudio.mygame.service

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
     */
    fun getGreeting(): String

}
```

Zero imports, zero SDK dependency — this is what "keep the API module lean" looks like taken to its logical conclusion. Like the Java example, `@ElementServiceExport` is applied to the implementation, not this interface.

---

4. `service/GreetingServiceImpl.kt` — 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)
class GreetingServiceImpl : GreetingService {

    private lateinit var userService: UserService

    @Inject
    fun setUserService(userService: UserService) {
        this.userService = userService
    }

    override fun getGreeting(): String {
        // 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.
        val currentUser: User = userService.getCurrentUser()
        val isLoggedIn = currentUser.level != User.Level.UNPRIVILEGED
        val name = if (isLoggedIn) currentUser.name else "Guest"
        return "Hello, $name!"
    }

}
```

- `@ElementServiceExport(GreetingService::class)` exposes 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.
- `private lateinit var userService: UserService` paired with a separate `@Inject`-annotated setter is Kotlin's version of Guice setter injection. `lateinit` is required here because Kotlin non-nullable properties must otherwise be initialized in the constructor — this tells the compiler "trust me, this will be set before it's read," which holds because Guice calls `setUserService` during injector construction, before any request reaches `getGreeting()`.
- `currentUser.level` and `currentUser.name` are Kotlin property syntax calling the underlying Java getters (`getLevel()`, `getName()`) on the SDK's `User` model class — Kotlin automatically exposes JavaBean-style getters/setters as properties.
- The auth check is identical to the Java example: compare `currentUser.level` against `User.Level.UNPRIVILEGED` to distinguish a real logged-in user from an anonymous/guest request. This is the only auth-level check in the project — it does not check for `SUPERUSER`.

---

5. `HelloWorldApplication.kt` — 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 jakarta.ws.rs.core.Application

@ElementServiceImplementation
@ElementServiceExport(Application::class)
class HelloWorldApplication : Application() {

    companion object {

        @JvmField
        @ElementDefaultAttribute("true")
        val AUTH_ENABLED: String = "dev.getelements.elements.auth.enabled"

        @JvmField
        @ElementDefaultAttribute("example-element")
        val APPLICATION_PREFIX: String = "dev.getelements.elements.app.serve.prefix"

        @JvmField
        @ElementDefaultAttribute("/element/example/api")
        val RS_ROOT: String = "dev.getelements.elements.element.rs.root"

        @JvmField
        @ElementDefaultAttribute("/element/example/ws")
        val WS_ROOT: String = "dev.getelements.elements.element.ws.root"

        @JvmField
        @ElementDefaultAttribute("/app/static/test/path")
        val STATIC_CONTENT_URI: String = "dev.getelements.element.static.uri"

        @JvmField
        @ElementDefaultAttribute("/app/ui/test/path")
        val UI_CONTENT_URI: String = "dev.getelements.element.ui.uri"

        const val OPENAPI_TAG: String = "Example"
    }

    /**
     * Here we register all the classes that we want to be included in the Element.
     */
    override fun getClasses(): Set<Class<*>> = setOf(
```

```
        // Endpoints
        HelloWorld::class.java,
        HelloWithAuthentication::class.java,
        ExampleContent::class.java,

        // 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.java
    )

}
```

- The attribute-key constants live in a Kotlin `companion object` rather than as `static final` fields directly on the class, since Kotlin classes have no native concept of static members.
- Each one is also marked `@JvmField`. Without it, a Kotlin `val` in a companion object compiles to a *private* backing field plus a generated getter method — the SDK's annotation processor reads fields directly via reflection, not getters, so it would find nothing. `@JvmField` tells the compiler to expose the property as a plain public static field instead, exactly like the Java example's `public static final String AUTH_ENABLED = "...";`.
- `OPENAPI_TAG` uses `const val` instead — since it has no `@ElementDefaultAttribute` annotation to preserve field access for, a Kotlin compile-time constant (which *does* compile to a plain static final field automatically) works without `@JvmField`.
- `AUTH_ENABLED` defaulting to `"true"` is what turns on the built-in Elements auth filter for every request and service in this Element — exactly what makes `UserService.getCurrentUser()` populated in `GreetingServiceImpl`. As noted in the setup section above, `APPLICATION_PREFIX` (not `RS_ROOT`) is what actually determines the externally observed `/app/rest/{prefix}/...` mount path.
- `getClasses()` is a Kotlin expression-body override returning a `Set<Class<*>>` — Kotlin's star-projection `*` stands in for Java's unbounded wildcard `Class<?>`.

---

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

```
package com.mystudio.mygame

import dev.getelements.elements.sdk.jakarta.rs.AuthSchemes.SESSION_SECRET
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 io.swagger.v3.oas.annotations.enums.SecuritySchemeIn.HEADER
import 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(
    value = [SecurityScheme(
        type = APIKEY,
        `in` = HEADER,
        name = SESSION_SECRET,
        paramName = SESSION_SECRET,
        description = "Session secret required for authenticated endpoints"
```

```
    )]
)
class OpenAPISecurityConfig
```

Functionally identical to the Java example — a class with no fields or methods, registered in `getClasses()` purely so Swagger's scanner picks up its class-level annotations describing the `Elements-SessionSecret` API-key security scheme. Two Kotlin syntax notes:

- Annotation array attributes use Kotlin's `[ ... ]` collection-literal syntax (`security = [SecurityRequirement(...)]`) instead of Java's `{ ... }` array-initializer syntax.
- `in` is a reserved keyword in Kotlin (used for ranges and `for` loops), but it's also the actual parameter name on `@SecurityScheme`. Wrapping it in backticks — `` `in` = HEADER `` — lets Kotlin use it as an identifier anyway.

---

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

```
package com.mystudio.mygame.rest

import com.mystudio.mygame.HelloWorldApplication
import io.swagger.v3.oas.annotations.Operation
import io.swagger.v3.oas.annotations.tags.Tag
import jakarta.ws.rs.GET
import jakarta.ws.rs.Path
import jakarta.ws.rs.Produces
import jakarta.ws.rs.core.MediaType

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

    @GET
    @Produces(MediaType.TEXT_PLAIN)
    @Operation(summary = "Hello world probe", description = "Returns a simple greeting")
    fun sayHello(): String = "Hello world!"

}
```

The simplest possible JAX-RS resource — a plain `GET` returning static text via a one-line expression body, 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. Note it references `HelloWorldApplication.OPENAPI_TAG` directly as a companion-object member, Kotlin's equivalent of Java's `static` field access — no `import static` needed.

---

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

```
package com.mystudio.mygame.rest

import com.mystudio.mygame.HelloWorldApplication
import com.mystudio.mygame.service.GreetingService
import dev.getelements.elements.sdk.Element
import dev.getelements.elements.sdk.ElementSupplier
import dev.getelements.elements.sdk.jakarta.rs.AuthSchemes.SESSION_SECRET
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

@Tag(name = HelloWorldApplication.OPENAPI_TAG)
@Path("/hellowithauthentication")
class HelloWithAuthentication {

    private val element: Element = ElementSupplier
        .getElementLocal(HelloWithAuthentication::class.java)
        .get()

    private val greetingService: GreetingService = element
        .serviceLocator
        .getInstance(GreetingService::class.java)

    @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)]
    )
    fun sayHelloWithAuth(): String = greetingService.getGreeting()

}
```

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

- `ElementSupplier.getElementLocal(HelloWithAuthentication::class.java).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.serviceLocator.getInstance(GreetingService::class.java)` then pulls the singleton out of that injector — `element.serviceLocator` is Kotlin property syntax calling the Java interface's `getServiceLocator()` getter. This only works because `MyGameModule` called `expose(GreetingService::class.java)` — remove that call and this lookup throws.
- Both lookups happen in property initializers (`private val element = ...`), run once when the resource is constructed, rather than in a Kotlin `init` block or lazily — same eager-initialization behavior as the Java example's `final` fields.

---

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

```
package com.mystudio.mygame.rest

import com.mystudio.mygame.HelloWorldApplication
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

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

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

        // 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
        val response = ExamplePostResponse()
        response.name = examplePostRequest.name
        return response
    }

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

        // 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
        val response = ExamplePutResponse()
        response.name = examplePutRequest.name
        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")
    fun examplePutWithPathParam(@PathParam("name") name: String, examplePutRequest:
ExamplePutRequest): ExamplePutResponse {

        // 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
        val response = ExamplePutResponse()
        response.name = examplePutRequest.name
        response.metadata = mapOf("name" to 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`. Note the two methods both named `examplePost` — this compiles because Kotlin (like Java) allows overloading by parameter type, and JAX-RS dispatches by HTTP method + path + media type at request time, not by the method name Kotlin sees at compile time.

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

@Schema
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.")
    var name: String? = null

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

    @Schema(description = "Any additional information to return.")
    var metadata: Map<String, Any>? = null

}
```

`@Schema` annotations document each 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 runs. Kotlin's `var name: String? = null` compiles to a private field plus public getter/setter, same shape as the Java example's explicit getters/setters, just without writing them out — Jackson (the runtime's JSON library) serializes and deserializes these via the generated accessors with no special Kotlin configuration needed (see the note on Jackson below).

Kotlin-Specific Considerations

Pulling together the Kotlin-only details scattered through the sections above, plus two more that don't have a single obvious home:

- **`package-info.java` must stay Java** — package-level annotations require `ElementType.PACKAGE`, which Kotlin doesn't support. See the Maven deep dive above for the dual-compiler-execution setup this requires.
- **`@JvmField` on annotated companion-object properties** — required wherever an SDK annotation processor reads a field via reflection (like `@ElementDefaultAttribute` in `HelloWorldApplication`), since a plain Kotlin `val` in a companion object is a private field behind a getter, not a public static field.
- **`@file:JvmName("Run")` on top-level entry points** — needed if you want a clean, predictable class name for a Kotlin file with a top-level `fun main()`, since Kotlin otherwise names the generated facade class after the file (`RunKt` for `run.kt`). See the debug runner section below.

- **`kotlin-stdlib` must ship inside the `.elm`** — both in `lib/` (for `element`'s own runtime classpath) and in `api/` (in case any exported Kotlin type gets reflected over from another Element's classloader). The Elements runtime has no reason to provide Kotlin's standard library itself.
- **Do not add `KotlinModule` to Jackson** — the Elements runtime already configures a shared `ObjectMapper`. Registering Jackson's `KotlinModule` yourself causes classloader conflicts with that shared instance. Plain Kotlin `data class` / `class` DTOs (like the model classes above) serialize correctly via their generated getters with no special configuration — as demonstrated by the live JSON responses in the setup section above.

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. See Direct Database Access and Batch Configuration and this repository's own `MORPHIA.md` for the `Transaction` / `Datastore` / `@ElementTypeRequest` patterns you'd use to add it.
- 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.kt`

```
@file:JvmName("Run")

import dev.getelements.elements.sdk.local.ElementsLocalBuilder
import java.io.File

/**
```

```
 * Runs your local Element in the SDK.
 *
 * Working directory must be the project root (element-example-kotlin/).
 * IntelliJ: Run → Edit Configurations → Working directory → set to this project root.
 */
fun main() {

    // 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.
    val uiDir = File("ui")

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

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

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

    val 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()

}
```

Where the Java example wraps its entrypoint in a bare, package-less `public class run` with a `static void main(String[] args)`, Kotlin doesn't require a wrapping class at all — `run.kt` is a top-level file with a single top-level `fun main()`. Two Kotlin-specific things to note:

- `@file:JvmName("Run")` at the top of the file is a file-level annotation that renames the generated facade class from the Kotlin default (`RunKt`, derived from the filename) to `Run`. This is why the run command above passes `-Dexec.mainClass=Run` rather than `RunKt` — and it's a good habit generally, since `<FileName>Kt` is not a name you'd want to type or see in a stack trace.
- No `package` declaration — the file sits directly under `debug/src/main/kotlin/` with no sub-package, so both `Run` and `RunKt` would resolve at the default (unnamed) package if you ever needed to reference either directly.

Functionally, the four steps (npm install/build, docker compose, then `ElementsLocalBuilder`) are identical to the Java example and were walked through in the setup section above; the key API points 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 Element's code — this mechanism has nothing to do with the backend language, so everything in this section is identical between the Kotlin and Java example projects. 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 `/app/rest/example-element/hellowithauthentication` from a plugin, you'll need to add that header yourself.

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/`) — verified live at `http://localhost:8080/app/ui/test/path/superuser/plugin.json` in the setup above:

```
{
  "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.

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)

> Note
>
> In this project, `{prefix}` for these two trees is not the same as the REST API's `example-element` prefix. `HelloWorldApplication` overrides the full serve URI directly via `dev.getelements.element.static.uri` (default `/app/static/test/path`) and `dev.getelements.element.ui.uri` (default `/app/ui/test/path`) — both clearly placeholder values, not wired to anything meaningful. Verified live: `curl http://localhost:8080/app/static/test/path/index.html` returns the contents of `element/src/main/static/index.html`, and `curl http://localhost:8080/app/ui/test/path/superuser/plugin.json` returns the manifest shown above. For your own Element, either drop these two attribute overrides (to get the standard `/app/static/{prefix}` / `/app/ui/{prefix}` behavior shown in the list above) or set them to something meaningful.

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}` (overridden in this project, per the note above)

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

Watch the Kotlin quickstart video again if any step above didn't click: https://www.youtube.com/watch?v=6kLWRMex-ug

- Building the Example Element: A Complete Walkthrough — the Java version of this same project, useful for comparing the two side by side
- 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 shared by both the Kotlin and Java examples
- Direct Database Access and Batch Configuration — for adding real MongoDB/Morphia persistence, not demonstrated in this project