Java is a trademark of Sun Microsystems, Inc.

Java™ Champions

JavaOne℠

Energy, CO2 Savings with Java™ Platform, Enterprise Edition and More: Project GreenFire

Adam Bien, Consultant
adam-bien.com
blog.adam-bien.com

What Is GreenFire?

- GreenFire is a a “heating tuner”, actually a heating regulator
- Beyond heating regulation, GreenFire is also an interesting automation platform to control ventilation and parts of the house automation
- GreenFire is opensource: greenfire.dev.java.net
- It is already 2 years in production. Energy savings: about 20-50%

What Is GreenFire?

- I was not very happy with the way how the heating regulation worked.
- ...I begun to regulate the heating manually (what was a lot of plumbing)
- The first prototype was started with Ruby On Rails in early 2006, but the problem were the conventions and the DB choice (only few databases were supported that time)
- The second prototype (with hardcoded rules) was built at a weekend with EJB 3 preview

GreenFire - And How It Started

- It was one of the very first Java EE 5 apps in 2006
- GreenFire is continually (mis)used to evaluate some Java EE patterns and best practices as well
- It is also a nice application to test JavaFX™ capabilities

Context / What is Solar Heating?

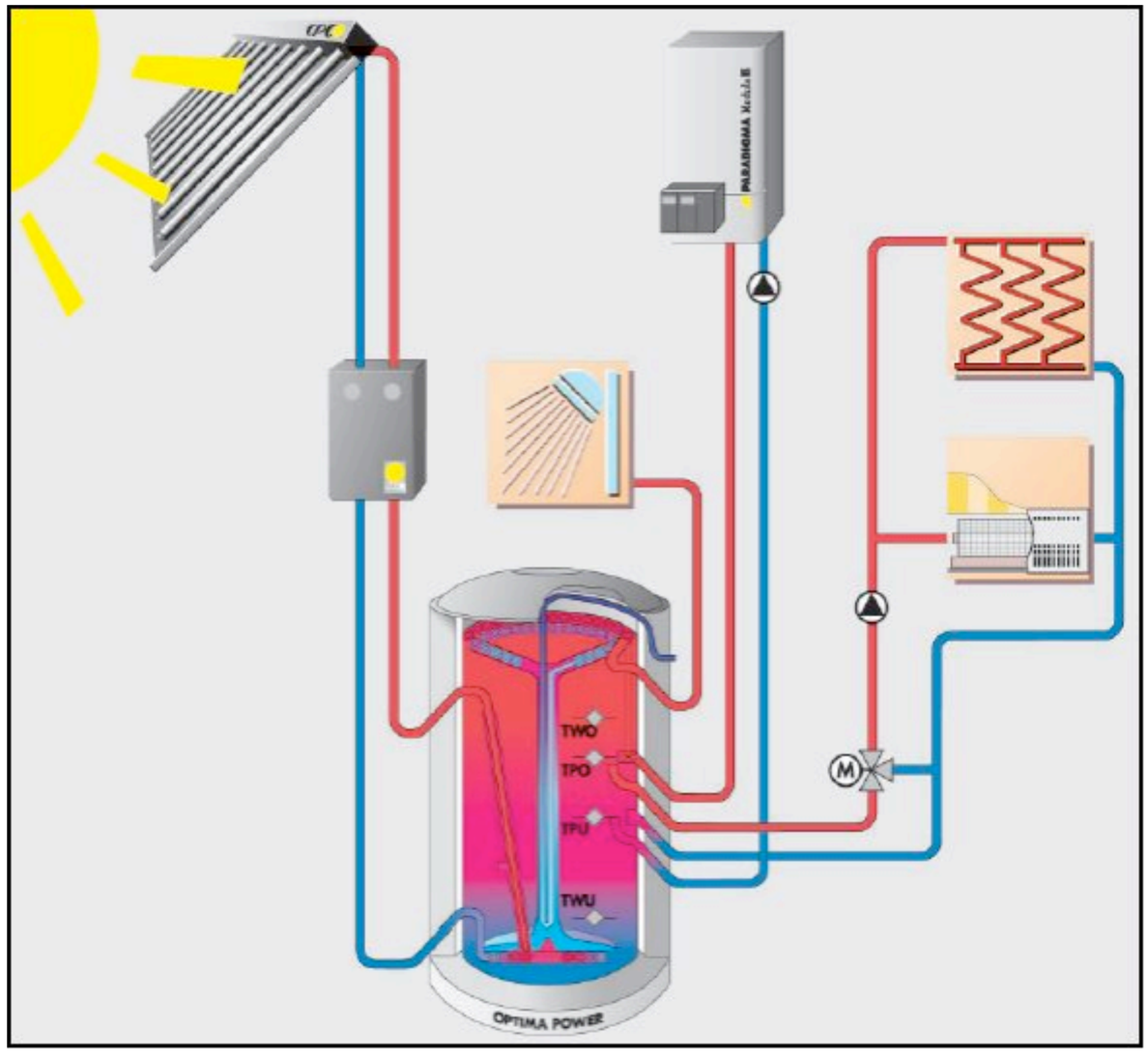
- "Is the usage of solar energy to provide process, space or water heating"
- Solar energy is cheap comparing it to oil/gas/wood-pellet ..and CO2 neutral.
- But: conventional energy sources are easier to control and manage.
- However, it is hard to predict, whether, when and how long the sun will shine (at least in Germany / Bavaria ☺).

GreenFire - The Intention

- Situation/context dependent prioritization of „bio“ resources (solar over wood-pellets etc.)
- Highest possible energy and CO2 savings
- Intelligent control of circulation-pumps, heating and ventilation
- Monitoring, reports and remote control
- …and FUN (JavaFX, SunSPOTs, Groovy etc.)

Real (house) hardware - “Deployment Diagram” :-)

- TWO: upper water temp
- TPO: upper heating temp
- TPU: lower heating temp
- TWO: lower heating temp



How It Works?

- Modern heating systems use buffers to store the (solar) energy:
  - The buffer is the heart of the heating control.
  - The buffer's temperature is important for the decision, whether the primary heating source should be activated or turned off
- Even modern systems only use a timer driven approach to control the heating. The current solar-power, weather predictions, internal house temperature are not taken into consideration :-(.

Use (Problem) Cases - Wrong Prioritization

- Buffer is cold, the sun shines, ...and the timer starts the primary energy source (oil/wood pellet)

- Buffer is cold, you are cold, so you are going to make fire in the wood burning stove (it is connected with the buffer). .. your heating doesn't know that (more precisely - it ignores that) and starts the primary heater again (this energy is wasted)

Use (Problem) Cases - Wrong Prioritization

- Autumn/Spring, the buffer is warm and the sun shines.To avoid overheating the control decides to turn the solar collectors off, instead of turning on the heating in the basement (another kind of cooling).

- You are enjoying the wood burning stove in the evening, however it produces too much heat (= warm water). The house heating could be automatically turned on for cooling, and turned off again at a predefined point automatically.

Use (Problem) Cases - Wrong Prioritization

- The weather forecast for the next day is good (=sun). But the primary source (oil/wood pellet) heats the buffer in the night. Morning sun is too weak to heat the already warm buffer. The energy is wasted again.
- In case the sun shines – it is usually warm enough – the house heating as well as the heater could be turned off.
- …and many, many ideas.

Technology decisions: why EJB 3?

- EJB 3 technologies are POJOs with only few annotations (@Stateless, @Local).
- The container cares about state, concurrency and dependency management (it “injects” other EJB technologies , PersistenceContext, JMS queue or the DataSource)
- The DI (Dependency Injection) relies on convention, rather than configuration – so XML is no more necessary

Why EJB 3?

- Applications built with EJB technology are leaner than without:
  - Dependency Injection makes ServiceLocators, factories and even constructor invocations superfluous
  - Additional frameworks or libraries are optional (mostly not needed)
- The integration with Java EE 5 platforms is superb. To create a simple “CRUD” application only few lines of code are necessary

Why EJB 3?

- Transactions and concurrency issues are already solved for you:
  - No need to use ThreadLocals, Singletons or another hacks to associate the EntityManager with a transaction
  - The Entities inside a transaction remain consistent
  - EJB 3 technology can be easily injected into a Java Servlet API, JSP™ framework or Managed Beans – the concurrency and transactions are correctly handled by the application server

Why EJB 3?

- Transactions are propagated to all methods within the same thread, even scripts (Groovy, JavaScript etc.)
- EJB 3 are well integrated with JMS (sender and receiver)
- JSR-223 scripts can be loaded and executed directly into a Session Bean
- EJBs have to expose their monitoring data via JMX (JSR-77). Monitoring is an important requirement for a heating regulator

Why EJB 3?

- EJB 3 Timer Service is perfect for the implementation of a heart beat
- The EntityManager is correctly “synchronized” by the EJB container in multi-threaded environment
- Good JSR-311 (REST) and JSR-181 (SOAP) integration
- EJB components are portable and can be deployed to any Java™ EE 5 application server without additional effort

Why EJB 3/JPA?

- JPA entities are just annotated POJOs
- Superb integration with scripting components - attached entities can be directly manipulated
- The same entity can be used for XML serialization and persistence. Its Don't Repeat Yourself (DRY)
- DAOs and DTOs are an option - and not a necessity (important for smaller applications)
- EJB 3 and JPA in particular are easy unit-testable

Java EE

- GreenFire is a Java EE application:
  - workflow is implemented as EJB 3.0 Session Beans
  - business logic is implemented in Groovy and loaded via JSR-223 in a Session Bean
  - the current state and the decisions are persisted with JPA

GreenFire Glassfish v2 deployment

Name: GreenFire

Virtual Servers: server

Associates an internet domain name with a physical server

Description:

Status: ☑ Enabled

Java Web Start: ☑ Enabled

Location: ${com.sun.aas.instanceRoot}/applications/j2ee-apps/GreenFire

Object Type: user

Libraries:

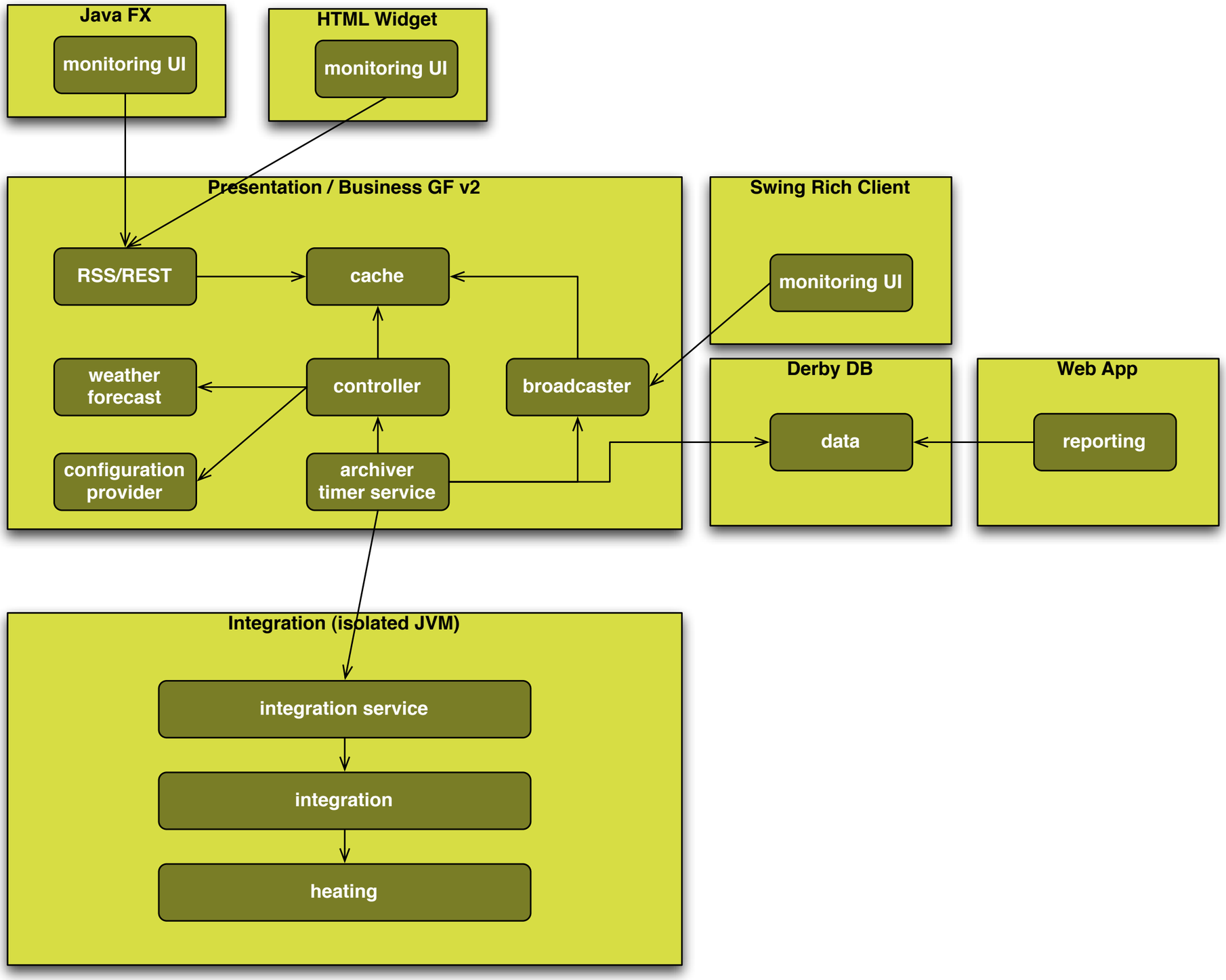
Sub Components (5)

Name	Type
HeatingControlBrain-ejb.jar	EJBModule
HeatingMessenger.jar	EJBModule
WeatherForecastProvider.jar	EJBModule
HeatingManagement.war	WebModule
HeatingControlRSS-war.war	WebModule

Java EE (contd.)

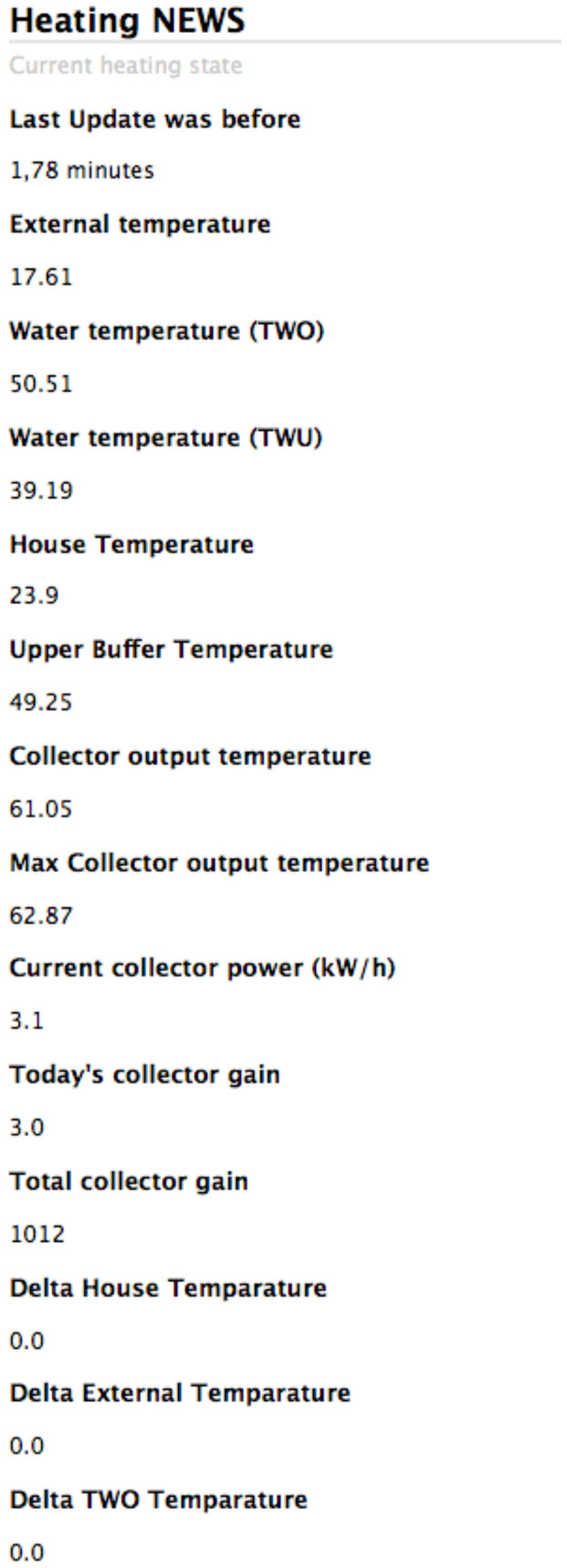
- the “heartbeat” is implemented with EJB timer service
- reports (currently BIRT) are accessing the database directly, via JDBC
- the sensor information is published via JMS topic internally (inside the EAR), and with shoal.dev.java.net externally
- RSS is just Servlet (JMS listener)
- the same data is also published with JSR-311 (REST, JAX-RS)

Components Overview



Heating News RSS Feed

- Updated every 5 minutes via JMS
- Data is distributed via Topic (openmq / Glassfish)
- RSS Feed is only one subscriber



GreenFire Status in iPhone

- Can be accessed from mobile phones directly
- A restful interface is available as well (useful for Java FX

Portability

- GreenFire was initially developed on JBoss 4.0
- Porting to Glassfish™ v1, then v2 was painless (only copy+paste of the EAR)
- It runs currently on Glassfish v2.1
- No proprietary extension are used
- The UI is based on Java FX Script, RSS, HTML and Swing

IDE and Environment

- Greenfire was developed with plain Netbeans 5.5.1/6.7 without any extensions (reason: lazyness, better compatibility between developers)

- Greenfire is tested with (www.paradigma.de). However only a small part of the integration layer is dependent on the proprietary heating-protocol. All other parts are heating-equipment independent. Actually every heating system with an accessible interface, should be easy to integrate.

Packaging and Prerequisites

- GreenFire is an ear, which consists of several ejb-jar modules and some WAR applications.

- For the installation only a datasource (for archiving) and a JMS-topic (for publishing) are needed.

- The Java EE 5 application talks to an integration unit, which is realized as a lean RMI-server.

...and how it works

- Every 5 minutes the timer service (the archiver, actually the heartbeat) completes the following tasks:
  - It gathers the data (heating + configuration, weather forecast) and passes it to the „brain“
  - The brain is asked what to do (heating on/off, etc.)
  - The decision is passed to the integration layer and executed
- All the data is stored in the Derby DB and can be used in reports.

- The data in the Derby DB is used for reports and monitoring independently (it stores every 5 minutes all the data since about 3 years – and performs still very well -> no problems).

- The UI is accessing Derby DB to access the information – not directly the heating system (decoupling – the serial port seems not to be thread safe)

The “driver”

```
@Stateless
public class HeatingStateArchiverBean implements Archiver {
  //constant declaration omitted
    @EJB
    private HeatingNewsBroadcasterLocal heatingNewsBroadcasterBean;
    @EJB
    private HeatingDecider heatingDecider = null;
    @PersistenceContext
    protected EntityManager entityManager;
    @Resource
    private SessionContext sessionContext;
    private ParadigmaRemoteAccess paradigmaAccess;

    @PostConstruct
    public void initialize() {
        try {
            this.paradigmaAccess = (ParadigmaRemoteAccess) Naming.lookup(URL + ParadigmaRemoteAccess.NAME);
        } catch (Exception e) {
            logger.throwing(HeatingStateArchiverBean.class.getName(), "initialize", e);
            throw new IllegalStateException("Cannot connect to ParadigmaRemoteAccess " + e, e);
        }
    }

    @Timeout
    @TransactionAttribute(TransactionAttributeType.REQUIRES_NEW)
    public void archive(Timer timer) {
        HeatingStateItem heatingStateItem = fetchData();
        String xmlData = serializeToString(heatingStateItem);
    }

    private HeatingStateItem fetchData() {
        HeatingStateItem heatingStateItem = new HeatingStateItem();
        try {
            this.paradigmaAccess.initialize();
            //copying from integration DTOs to JPA entities
            heatingStateItem.setHeatingMode(this.paradigmaAccess.getHeatingMode().value());
            persist(heatingStateItem);
            HeatingMode heatingMode = this.heatingDecider.decide(heatingStateItem);
            if (heatingMode != null) {
                logger.info("Decider has decided: " + heatingMode + " changing the heating state to");
                this.paradigmaAccess.setHeatingMode(heatingMode);
            } else {
                logger.info("Decider has not decided.");
            }
        } catch (Exception e) {
```

The decider

```
@Stateless
public class HeatingDeciderBean implements HeatingDecider {

    @EJB
    private ConfigurationMgr configurationMgr = null;
    @EJB
    private AuditMgr auditMgr;
    @EJB
    private HeatingRuleEngine heatingRuleEngine = null;

    public HeatingMode decide(HeatingStateItem currentState){
        ConfigurationItem configurationItem = configurationMgr.getCurrentConfiguration();
        if(!configurationItem.isActive()){
            logger.info("Profile is deactivated. Returning null...");
            return null;
        }

        HeatingMode decision =  this.heatingRuleEngine.checkRules(configurationItem, currentState);
        logger.info("Rule engine decided: " + decision);
        logger.info("Writing audit to persistent store");
        this.auditMgr.writeAudit(currentState, configurationItem, decision);
        return decision;
    }
}
```

The “rule engine”

```groovy
@Stateless
public class GroovyRuleEngine implements HeatingRuleEngine {
    //some obvious constants omitted
    private static final HashMap<Integer, HeatingMode> MODE_MAP = new HashMap<Integer,HeatingMode>();
    private final static String ENGINE_NAME = "groovy";

    private ScriptEngine scriptEngine = null;

    @PostConstruct
    public void initializeScripting(){
        ScriptEngineManager engineManager = new ScriptEngineManager();
        this.scriptEngine = engineManager.getEngineByName(ENGINE_NAME);
        if (this.scriptEngine == null) {
            throw new IllegalStateException("Cannot create ScriptEngine: " + ENGINE_NAME);
        }
    }

    public HeatingMode checkRules(ConfigurationItem configurationItem,HeatingStateItem heatingStateItem){
        Bindings binding = this.scriptEngine.createBindings();
        binding.put("logger", logger);
        binding.put("configurationItem", configurationItem);
        binding.put("heatingStateItem", heatingStateItem);
        InputStream is = null;
        InputStreamReader isr = null;
        try {
            logger.info("Trying to executing script engine");
            is = getScriptAsStream();
            isr = new InputStreamReader(is);
            this.scriptEngine.eval(isr, binding);
            logger.info("Script executed !");
        } catch (Exception e) {
            throw new IllegalStateException("Exception during evaluating script: " +e,e);
        }finally{ /* closing the stream*/       }
        int heatingMode = (Integer) binding.get("suggestedMode");
        return MODE_MAP.get(heatingMode);
    }

    InputStream getScript(){
        try {
            URL url = new URL(getBaseUrlName() + getScriptName());
            return  url.openStream();
```

The Script (variable initialization)

```
tpoShouldHigh = configurationItem.getTPOHigh()
tpoShouldLow = configurationItem.getTPOLow()

twoShouldHigh = configurationItem.getTWOHigh()
twoShouldLow = configurationItem.getTWOLow()
solPowerLow = configurationItem.getSolPowerLow()

tpoIs = heatingStateItem.getTPO()
twoIs = heatingStateItem.getTWO()
tpuIs = heatingStateItem.getTPU()

heatingModeIs = heatingStateItem.getHeatingMode()
solPowerIs = heatingStateItem.getMomentaneLeistung()


OFF = configurationItem.getHeatingModeOff();
ON = configurationItem.getHeatingModeOn();
```

Groovy Script - Control Logic

```
if(heatingModeIs != 0){ // Change only if heating is not in AUTO mode

    // the buffer shouldn't be hotter than 72
    if(tpoIs > 72 || twoIs > 72){
          suggestedMode = ON
          logger.info("Too hot (70) suggested change: " + suggestedMode)
          return;
    }
    if(solPowerIs < solPowerLow){
        suggestedMode = OFF
        logger.info("The solar power is too low. Suggested change: " + suggestedMode);
        return;
    }
    if(tpoIs > tpoShouldHigh || twoIs > twoShouldHigh){
      suggestedMode = ON
      logger.info("Too hot suggested change: " + suggestedMode)
    }else if(twoIs < twoShouldLow || tpoIs < tpoShouldLow){
      suggestedMode = OFF
      logger.info("Too hot suggested change: " + suggestedMode)
    }else{
      suggestedMode = heatingStateItem.getHeatingMode()
      logger.info("Nothing to do. Just keeping the current mode: " + suggestedMode)
    }
}else{ // Heating is in the AUTO state. But cooling is needed anyway
    if(tpoIs > 68 || twoIs > 68){
      suggestedMode = ON
      logger.info("Temperature greater than 68: " + suggestedMode)
    }else{
        suggestedMode = heatingModeIs
        logger.info("Heating is in AUTO state. Keeping the state.")
    }
}
```

Java FX - XML Parsing

- GreenFire's state is also available via REST / XML
- The Java FX Script class HeatingState is a DTO - which is used as a model

```
<report>
    <timestamp>1239779266217</timestamp>
    <heating-mode>AUS</heating-mode>
    <ta>20.99</ta>
    <ti>23.4</ti>
    <tpo>50.51</tpo>
    <two>52.6</two>
    <twu>38.79</twu>
    <tpu>46.79</tpu>
    <tsa>40.69</tsa>
    <tsa-max>51.97</tsa-max>
    <current-power>0.0</current-power>
    <daily-gain>1</daily-gain>
    <total-gain>1075</total-gain>
</report>
```

```
public class HeatingState {
    public var mode: String;
    public var ta: Double on replace oldValue {
     println("\nALERT! ta has changed!");
     println("Old Value: {oldValue}");
     println("New Value: {ta}");
    };

    public var ti: Double;
    public var tpo: Double;
    public var two: Double;
    public var twu: Double;
    public var tpu: Double;
    public var tsa: Double;
    public var tsaMax: Double;
    public var currentPower: Double;
    public var dailyGain: Integer;
    public var totalGain: Integer;
    public var timestamp: Long;
```

Java FX - XML Parsing

- With Java FX parsing of XML data is fairly easy
- A HttpRequest can be directly connected with the parser, which invokes a function:

```
public function processResults(is: InputStream) {
    def parser = PullParser {
        documentType: PullParser.XML;
        input: is;
        onEvent: parseEventCallback
    };
    parser.parse();
    is.close();
}

def parseEventCallback = function(event: Event) {
    if (event.type == PullParser.START_ELEMENT) {
        processStartEvent(event)
    } else if (event.type == PullParser.END_ELEMENT) {
        processEndEvent(event)
    } else if (event.type == PullParser.END_DOCUMENT) {
        println("parsed");
    }
}

function processEndEvent(event: Event) {
    def eventText = event.text.trim();
    def qname = event.qname.name.trim();
    if (qname == "report" and event.level == 0) {
    } else if (event.level == 1) {
        if (qname == "twu") {
            heatingState.twu = Double.parseDouble(eventText);
        } else if (qname == "ta") {
            heatingState.ta = Double.parseDouble(eventText);
        } else if (qname == "ti") {
            heatingState.ti = Double.parseDouble(eventText);
        } else if (qname == "total-gain") {
            heatingState.totalGain = Long.parseLong(eventText);
        } else if (qname == "tpo") {
            heatingState.tpo = Double.parseDouble(eventText);
        } else if (qname == "tpu") {
            heatingState.tpu = Double.parseDouble(eventText);
        }//remaining attributes
    }
}

function run(){
    def httpRequest = HttpRequest {

        location: "http://localhost:8080/GreenFireRestMock/report.xml"
        method: HttpRequest.GET

        onInput: function(io:java.io.InputStream) {
            try {
                processResults(io);
            } finally {
                io.close();
            }
        }
```

Data distribution

- The current data is broadcasted every five minutes using JMS internally (inside the application server)
- The shoal.dev.java.net / fishfarm.dev.java.net is used to broadcast the data in the local network
- Broadcasted data can be accessed with every device (currently multi-media system, PCs, working on Java ME) over W-LAN

Design Decisions

- The rules for the heating can be changed without redeploying and especially recompiling the application.
- For the implementation the Fluid Kernel, Persistent Anemic Objects, Lookup Utility etc.. pattern were used (see: http://p4j5.dev.java.net).
- Initially Groovy was chosen (JDK 1.5 time, no JSR-223) for the implementation of the rules. In the next release JavaScript is going to be used (is already shipped with Java 6, faster and totally sufficient for GreenFire‘s purposes).

- Especially in the first phase, the rules had to be often changed for fine tuning
- The rules for the heating can be changed without redeploying and especially recompiling the application
- For the implementation the Fluid Kernel pattern was chosen
- Drools, JavaScript, JRuby etc. could be used for the rule validation as well

Lessons Learnt

- Java EE 5 is pragmatic and productive even for small projects. The first attempt was started with Ruby On Rails – it was not as efficient as expected (no built in monitoring, no support for free, opensource database that time).
- Mocking is great – but can cause a lot of trouble as well (who thinks about two‘s complement accessing hardware in Java).
- It is worth to try out even strange ideas

Interesting

- GreenFire reflects real world projects surprisingly well:
  - robust integration of legacy resources
  - batch processing (timer service)
  - impedance mismatch between reports/services and domain driven design (JPA)
  - unit tests challenges (mocking, integration tests)
  - fast and robust deployments
  - GreenFire's patterns were introduced into many real world Java EE projects

Conclusion

> Hacking Java EE 5 is good for the environment!

Resources

- greenfire.dev.java.net
- shoal.dev.java.net
- fishfarm.dev.java.net
- glassfish.dev.java.net
- blog.adam-bien.com
- www.javafx.com

JavaOne℠

Thank You

Adam Bien
blog.adam-bien.com

abien@adam-bien.com