Dive into RIFE

Geert Bevin
CTO
Uwyn bvba

gbevin@uwyn.com
http://uwyn.com
http://rifers.org

Who am I

- **Geert Bevin**
- **CTO of Uwyn,** a small custom application development company (http://uwyn.com)
- **founder of RIFE** (http://rifers.org)
- **creator and contributor of many RIFE projects:** RIFE/Crud, RIFE/Jumpstart, RIFE/Continuations, Bamboo (forum), Bla-bla List (RIA todo list), Drone (information bot), Elephant (blog)

Agenda

- **Essentials**
- A quick look
- Metaprogramming
- Q&A

What is RIFE?

Full-stack component framework to quickly and consistently develop and maintain Java web applications

What is RIFE?

Full-stack component framework to quickly and consistently develop and maintain Java web applications

- **Most things you need are inside one Jar**
- **Functionalities benefit from cross-layer integration**
- **Consistent approach throughout all layers**
- **Very easy to setup, configure and upgrade**

What is RIFE?

Full-stack **component framework** to quickly and consistently develop and maintain Java web applications

- **Provides a standardized structure for your application**
- **Manages the life-cycle of your application**
- **Reusable components for your business logic**
- **Declare their interactions independently**
- **Each framework layer is usable separately**

What is RIFE?

Full-stack component framework to **quickly** and **consistently** develop and maintain Java web applications

- **Integrated layers allow you to quickly get results with a minimal amount of code**
- **Best practices are enforced in a pleasant way, providing many additional features and a consistent approach throughout all applications**
- **Components can easily be reused in many contexts**

What is RIFE?

Full-stack component framework to quickly and consistently **develop** and **maintain** Java web applications

- **Creating maintainable applications is our first goal**
- **A lot of attention goes to code-level developer comfort**
- **Frustration reduction by instant changes and reloads**
- **Creative solutions for difficult problems**
- **Embraces established standards** (XHTML, HTTP , SQL, ...)

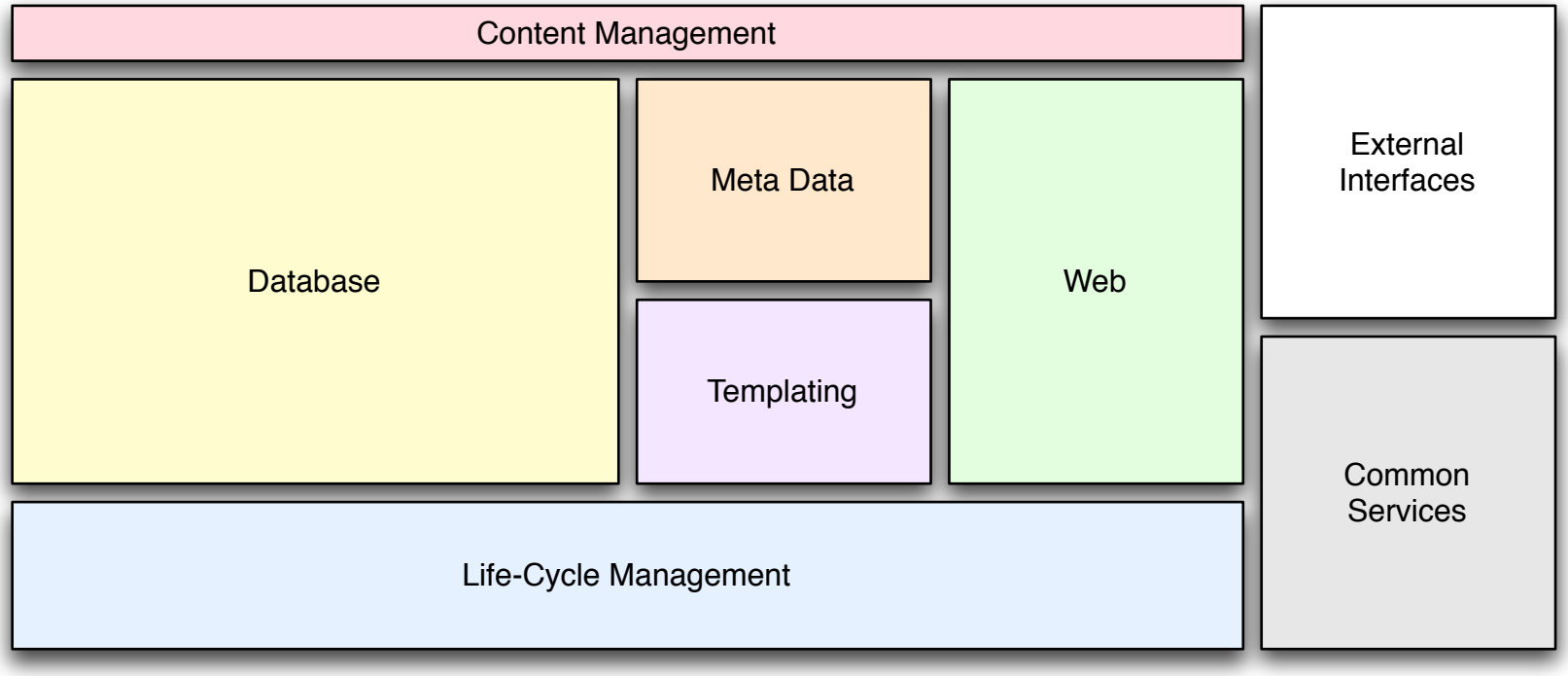
What is RIFE?

Full-stack component framework to quickly and consistently develop and maintain Java **web applications**

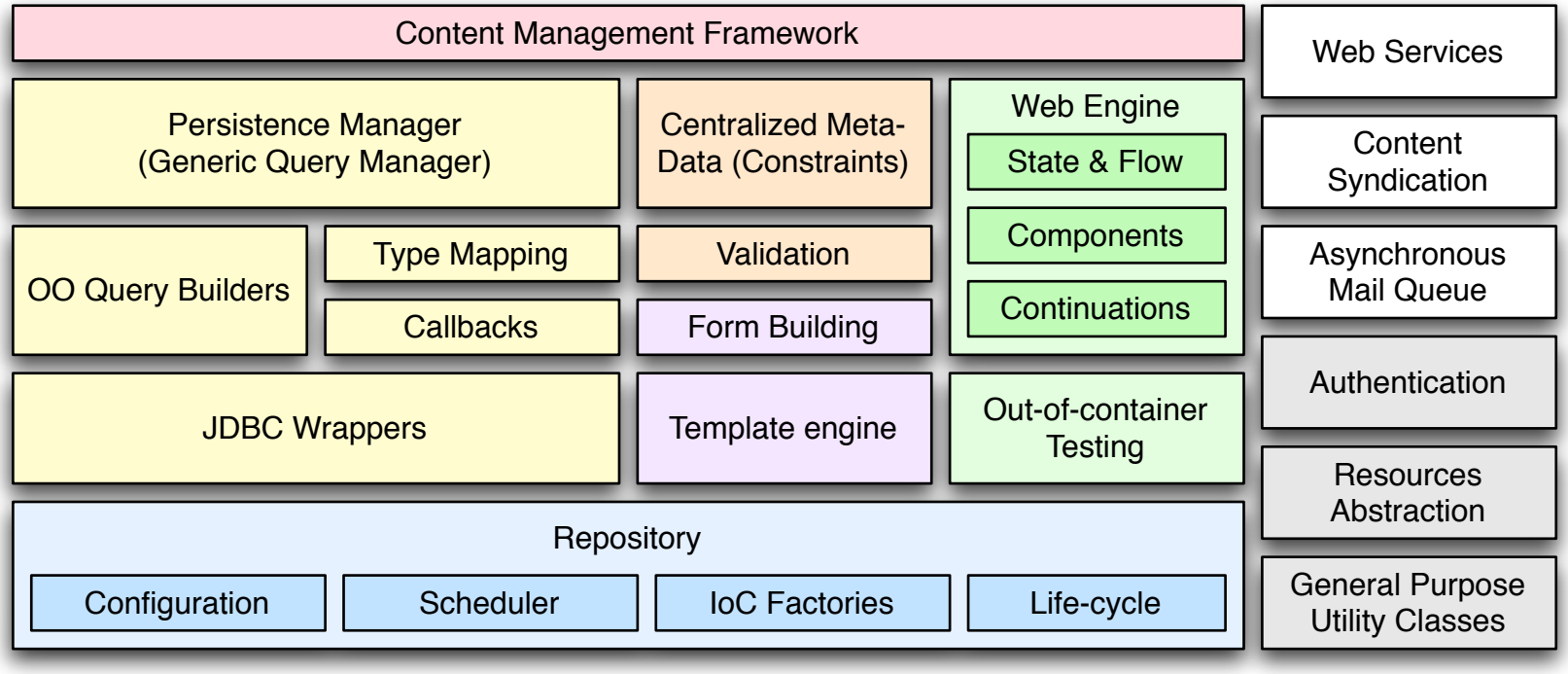
- **Geared to developing web applications and doesn't abstract away too much**
- **Everything besides the web engine is designed to be independently usable** (eg. fat clients)
- **Attention to the whole life-cycle of your application**

What's inside RIFE?

What's inside RIFE?



What's inside RIFE?



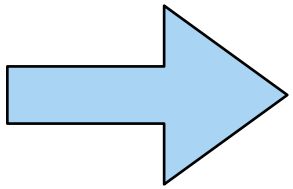
Agenda

- Essentials
- **A quick look**
- Metaprogramming
- Q&A

Dive right in with RIFE/Jumpstart

What is RIFE/Jumpstart?

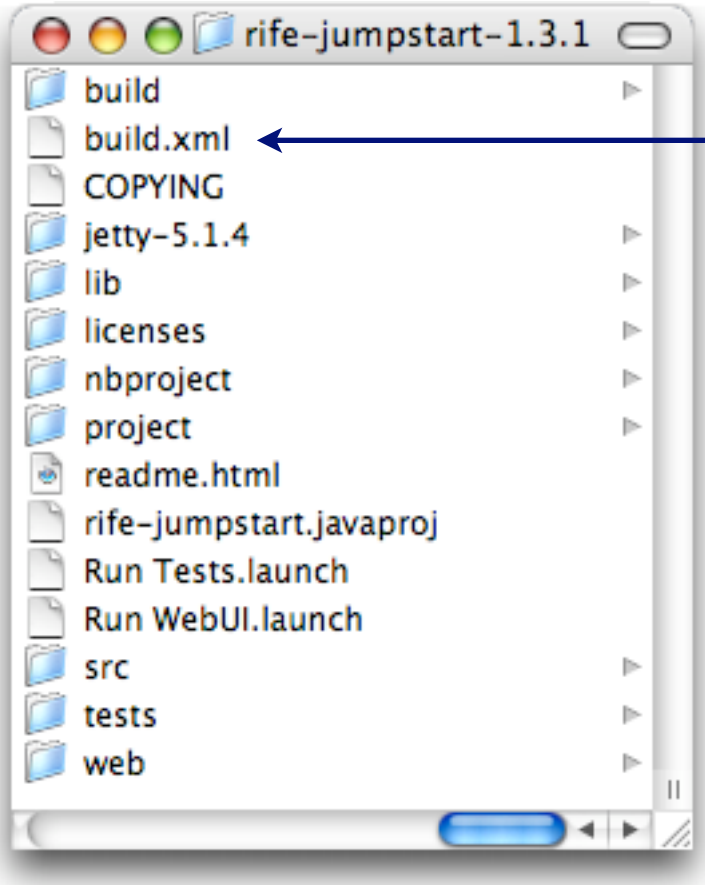
- **Source archive**
- **Makes it easy to start with new RIFE applications**
- **Contains everything you need**
- **Immediate support for most common development environments** (X-develop, Netbeans, Eclipse, IDEA, Ant)
- **Unzip, open project file, run, load, edit, reload**



Get started in a couple of minutes, download included

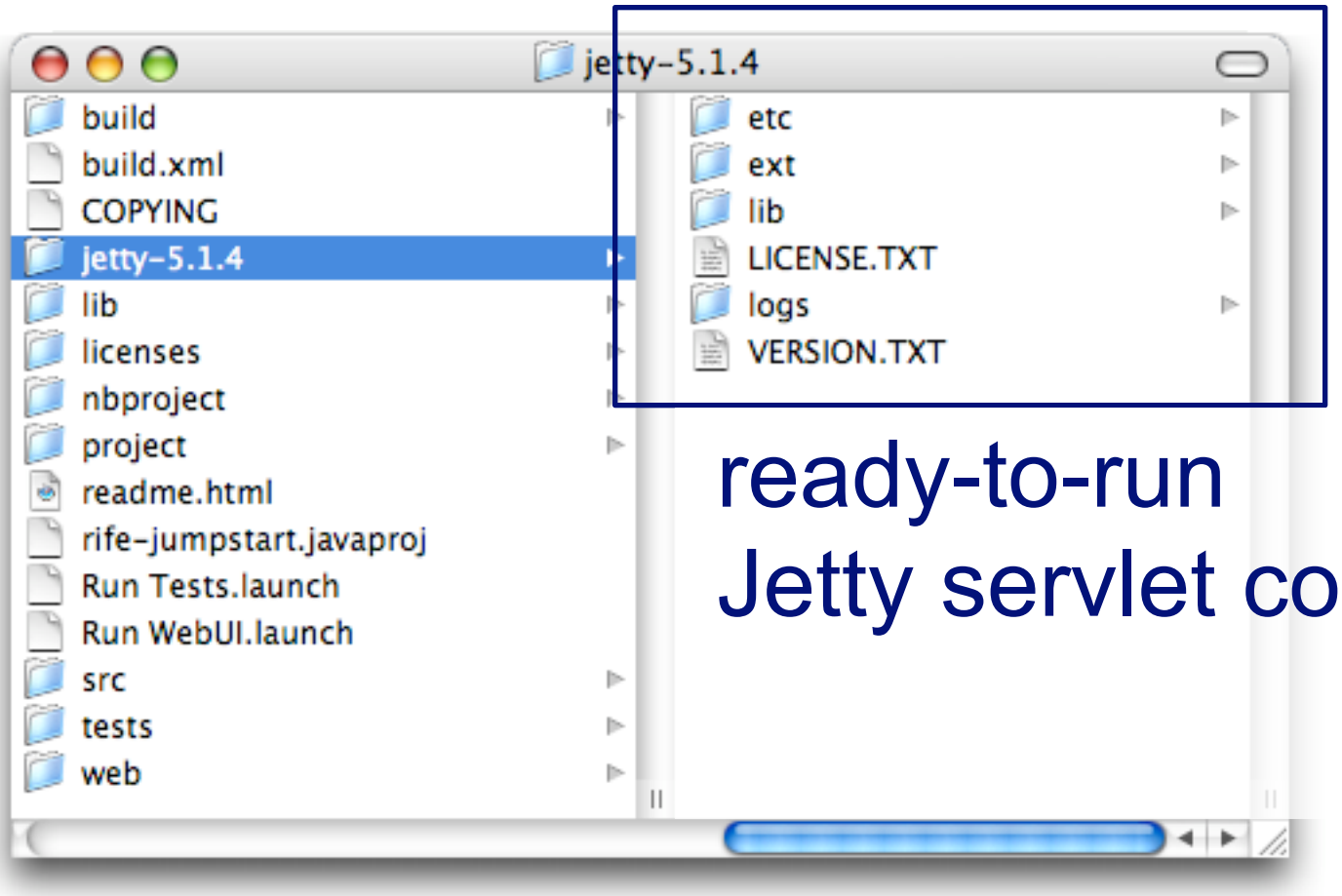
What's inside RIFE/Jumpstart?

What's inside RIFE/Jumpstart?



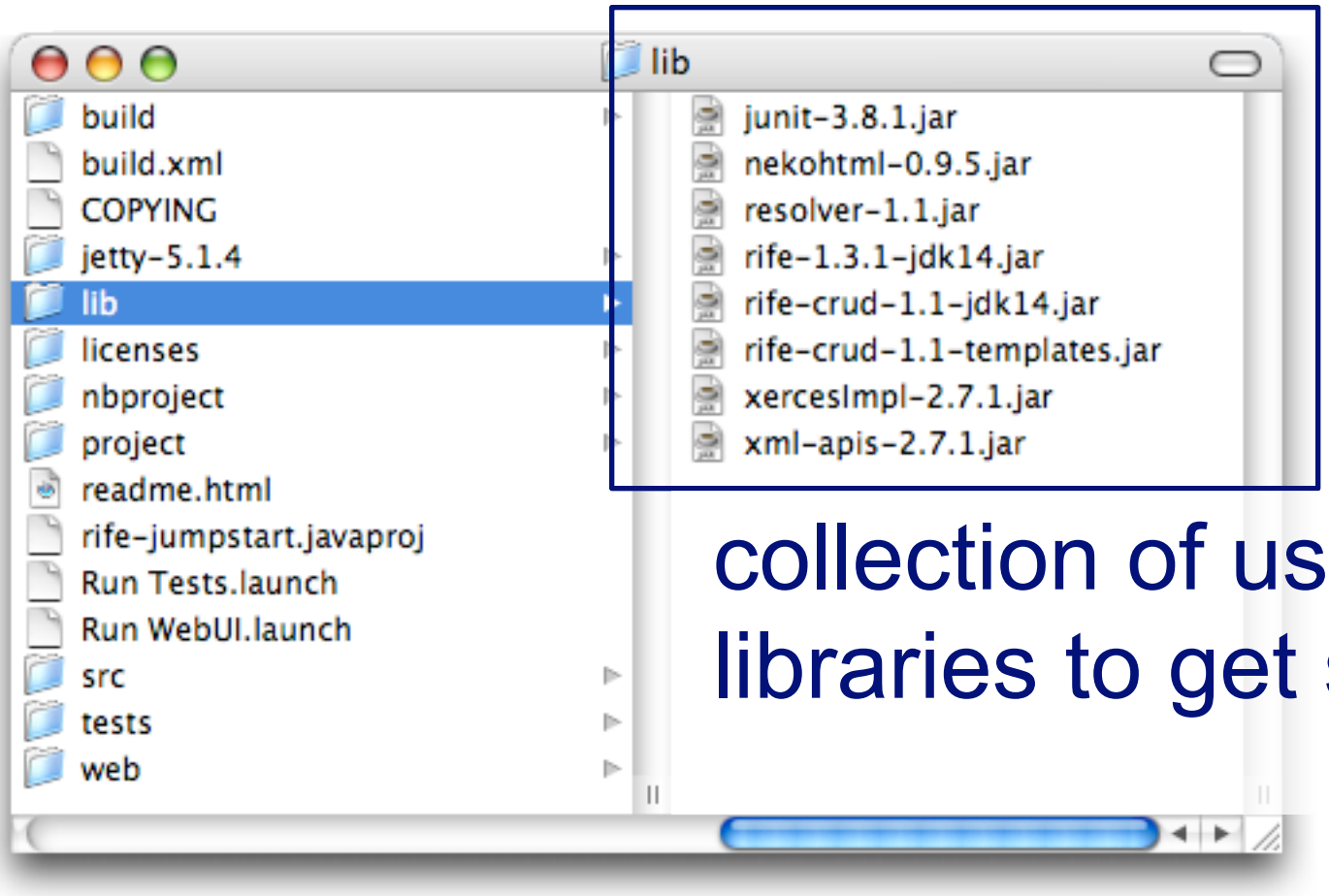
fully functional Ant build file with run, test and package targets

What's inside RIFE/Jumpstart?



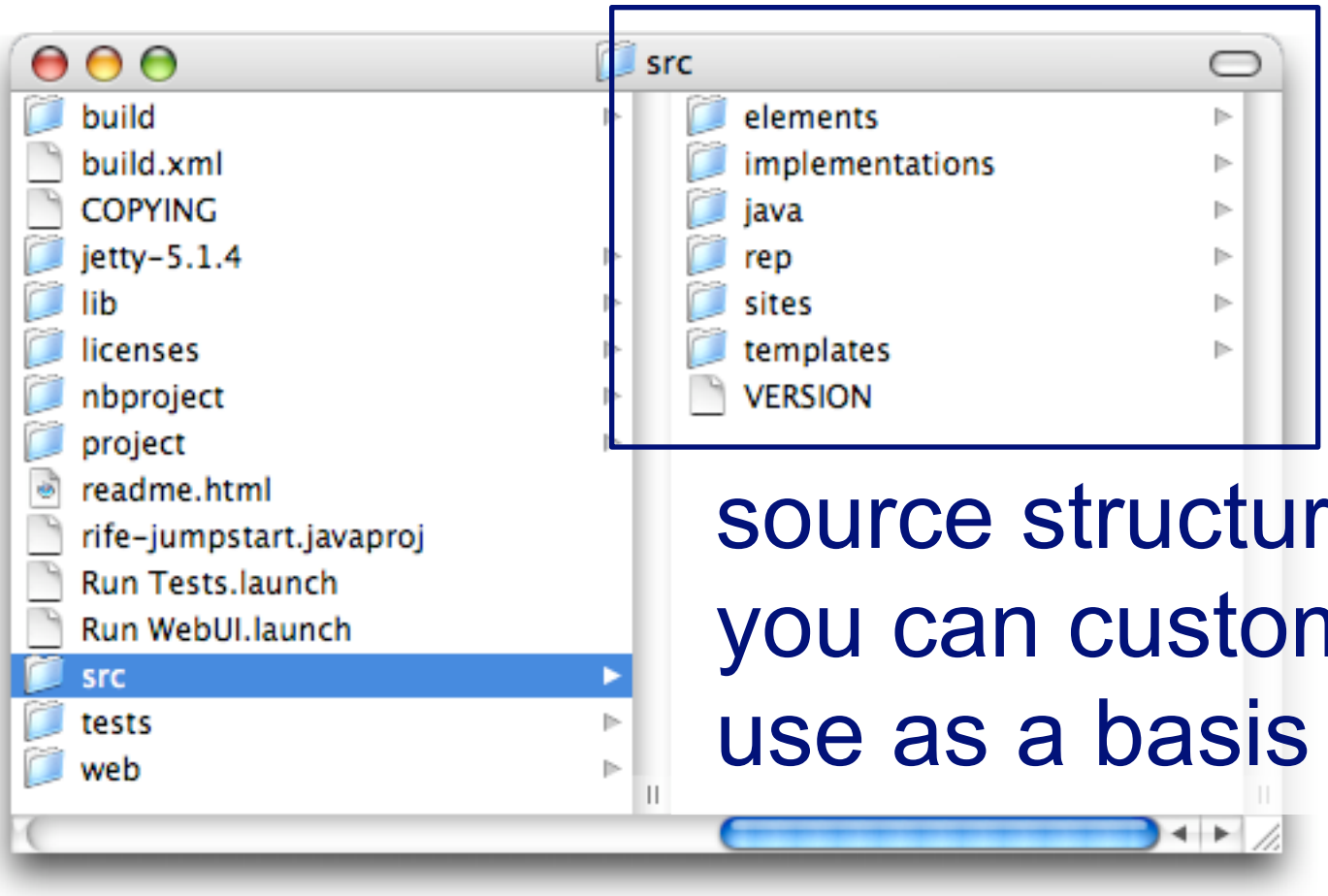
ready-to-run
Jetty servlet container

What's inside RIFE/Jumpstart?



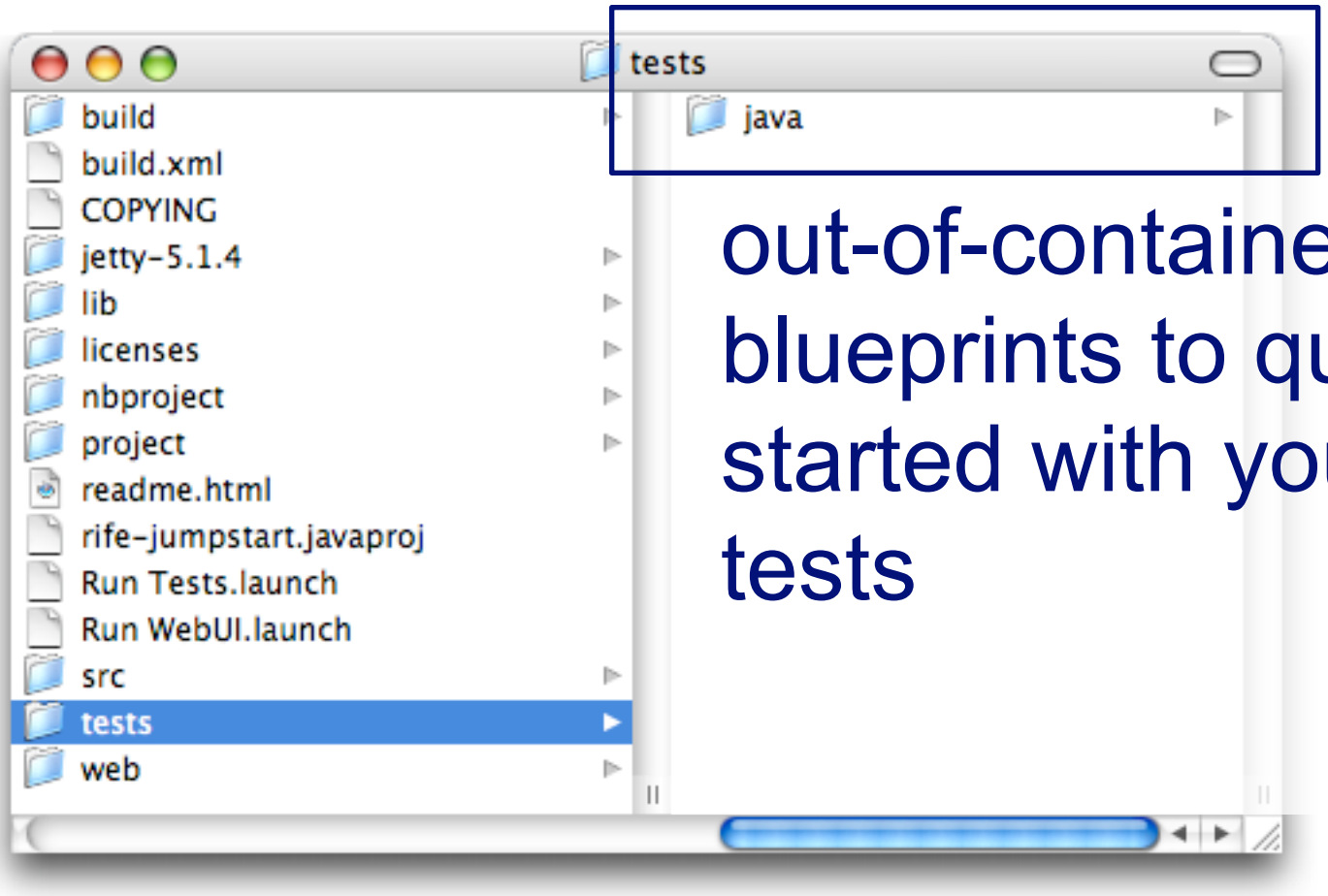
collection of useful libraries to get started

What's inside RIFE/Jumpstart?



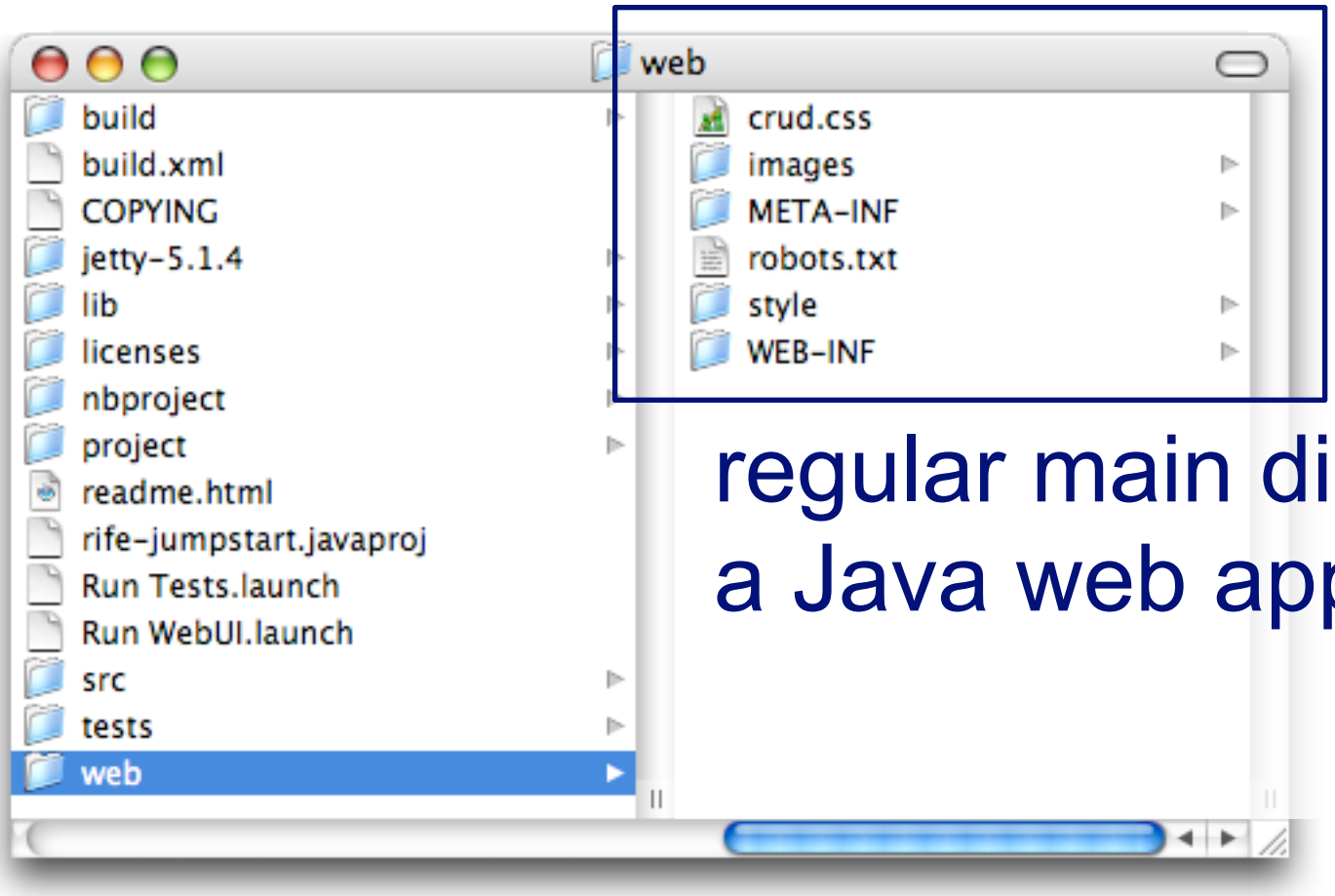
source structure that you can customize and use as a basis

What's inside RIFE/Jumpstart?



out-of-container test blueprints to quickly get started with your own tests

What's inside RIFE/Jumpstart?



regular main directory of a Java web application

What's inside RIFE/Jumpstart?

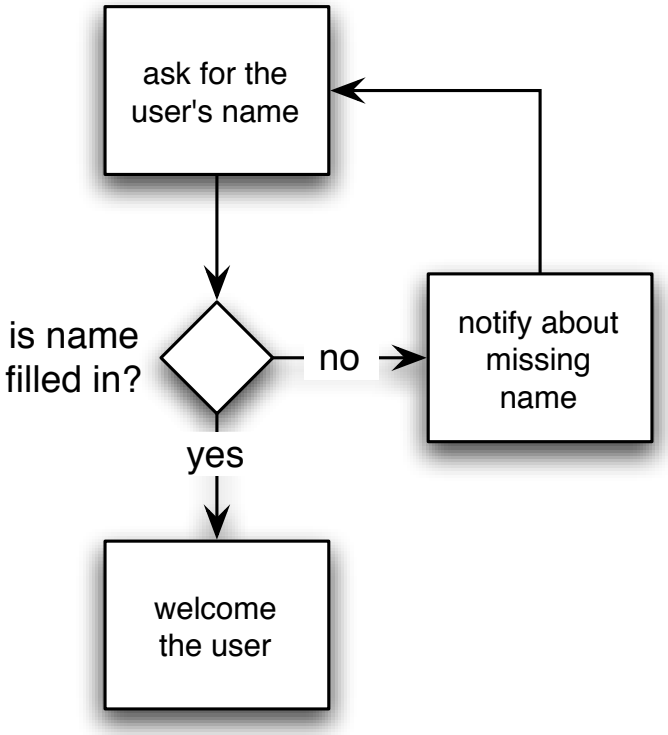
- The directory structure is just a suggestion, you can modify it
- Looks up everything through the classpath by default, your application can be packaged as a Jar
- Setup for development: automatic reloading without restarting
- The entire development of your application is being jump-started, even the packaging

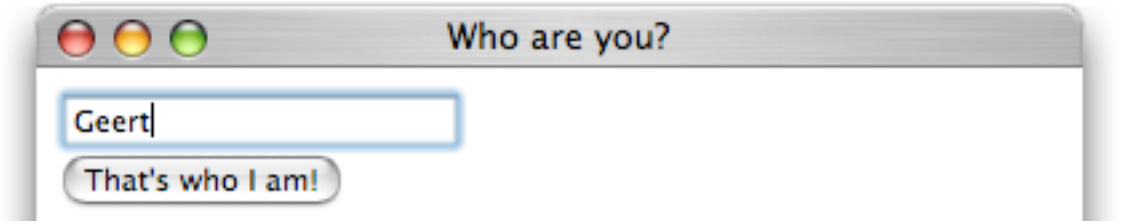
Overly simple example

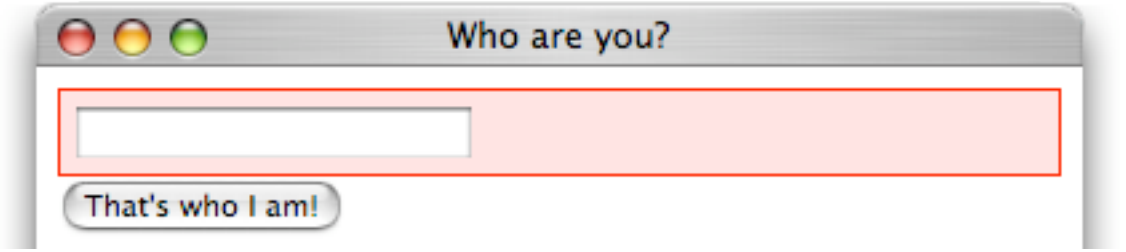
Overly simple example

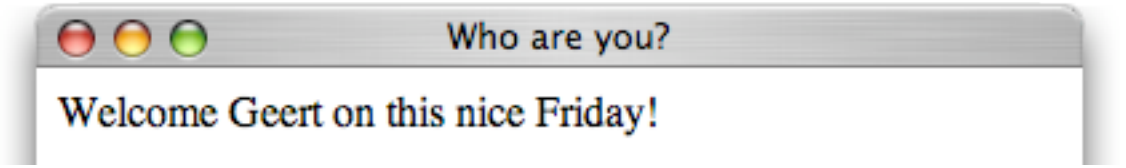
- It's time to look at some code!
- Let's start with the web engine and some templating
- Very few features have been used to keep things as basic as possible
- There are many ways to implement the example, this one is IoC-oriented and centralized in one component
- It might seem verbose, just as HelloWorld.java seems verbose: the platform wasn't built to make simple things simpler

Garrett IA diagram of the example









High-level structure and flow

High-level structure and flow

```
<site>
    <arrival destid="Home"/>

    <element id="Home" implementation="mypackage.Home" url="welcome">
        <property name="template"><template>pub.home</template></property>

        <submission name="myData">
            <param name="name"/>
        </submission>
    </element>
</site>
```

High-level structure and flow

```
<site>
    <arrival destid=

    <element id="Home
        <property nam

        <submission
            <param na
        </submission>
    </element>
</site>
```

```
welcome">
te></property>
```

Your web components (elements) are grouped together in a site.

A site is a component in itself and can be reused as such.

High-level structure and flow

```
<site>
    <arrival destid="Home"/>

    <element id="Home" implementation="mypackage.Home" url="welcome">
        <property name="template"><template>pub.home</template></property>

        <submission n
            <param na
        </submission>
    </element>
</site>
```

An element has a unique ID within its site, an implementation, and an optional URL.

High-level structure and flow

```
<site>
    <arrival destid="Home"/>

    <element id="Home" implementation="mypackage.Home" url="welcome">
        <property name="template"><template>pub.home</template></property>

        <submission name="myData">
            <param name="
        </submission>
    </element>
</site>
```

Properties can be injected into elements. In this case it's the template that will be used to construct the layout.

High-level structure and flow

```
<site>
    <arrival destid="Home"/>

    <element id="Home" implementation="mypackage.Home" url="welcome">
        <property name="template"><template>pub.home</template></property>

        <submission name="myData">
            <param name="name"/>
        </submission>
    </element>
</site>
```

Forms are sent to elements through named submissions with parameters.

High-level structure and flow

```
<site>
    <arrival destid="Home"/>

    <element id="Home" implementation="mypackage.Home" url="welcome">
        <property name="template"><template>pub.home</template></property>

        <submission name="myData">
            <param name="name"/>
        </submission>
    </element>
</site>
```

If the site is visited at the root URL, the Home element will be used.

High-level structure and flow

- **Declaration of all state transitions**
- **Clear overview of your application's building blocks**
- **RIFE's web engine will use this information to give you very advanced features** (eg. three-dimensional flow, portlet-like embedded elements, total relocatability, URL localization, complete reusability, ...)
- **XML, Java, Groovy and Janino builders**
- **Declaration can be automated: RIFE/Crud**

Component implementation (Element)

Component implementation (Element)

```
public class Home extends Element {

    private Template    template;
    private String      name;

    public void setTemplate(Template template) { this.template = template; }
    public void setName(String name) { this.name = name; }

    public void processElement() {
        print(template);
    }

    public void doMyData() {
        if (null == name || 0 == name.trim().length()) {
            template.setBlock("name class", "class error");
        } else {
            String day = new SimpleDateFormat("EEEE").format(new Date());
            template.setValue("day", encodeHtml(day));
            template.setBlock("content", "welcome");
        }
        print(template);
    }
}
```

Component implementation (Element)

```
public class Home extends Element {

    private Template    template;
    private String      name;

    public void setTemplate(Template template) { this.template = template; }
    public void setName(String name) { this.name = name; }

    public void pro
        print(templ
    }

    public void doM
        if (null ==
            templat
        } else {
            String day = new SimpleDateFormat("EEEE").format(new Date());
            template.setValue("day", encodeHtml(day));
            template.setBlock("content", "welcome");
        }
        print(template);
    }
}
```

The element implementations can either extend the Element class, or implement the ElementAware interface.

Component implementation (Element)

```
public class Home extends Element {

    private Template    template;
    private String      name;

    public void setTemplate(Template template) { this.template = template; }
    public void setName(String name) { this.name = name; }

    public void processElement() {
        print(template);
    }

    public void doMyD
        if (null == n
            template.
        } else {
            String da                                    new Date());
            template.
            template.
        }
        print(template);
    }
}
```

The template will be injected each time the element is instantiated.

The name will be injected when the submission data is sent.

Component implementation (Element)

```
public class Home extends Element {

    private Template    template;
    private String      name;

    public void setTemplate(Template template) { this.template = template; }
    public void setName(String name) { this.name = name; }

    public void processElement() {
        print(template);
    }

    public void doMyData() {
        if (null == name || 0 == name.trim().length()) {
            template.setBlock("name class", "class error");
        } else {
            String da                                          new Date());
            template.
            template.
        }
        print(template);
    }
}
```

The default entrance method here simply prints out the template.

Component implementation (Element)

```
public class Home extends Element {

    private Template    template;
    private String      name;

    public void setTemplate(Template template) { this.template = template; }
    public void setName(String name) { this.name = name; }

    public void processElement() {
        print(template);
    }

    public void doMyData() {
        if (null == name || 0 == name.trim().length()) {
            template.setBlock("name cla
        } else {
            String day = new SimpleDate
            template.setValue("day", en
            template.setBlock("content"
        }
        print(template);
    }
}
```

When a submission is sent, its name will be used to look for a matching method, which will be executed.

Component implementation (Element)

```
public class Home extends Element {

    private Template    template;
    private String      name;

    public void setTemplate(Template template) { this.template = template; }
    public void setName(String name) { this.name = name; }

    public void processElement() {
        print(template);
    }

    public void doMyData() {
        if (null == name || 0 == name.trim().length()) {
            template.setBlock("name class", "class error");
        } else {
            String day = new
            template.setValue(
            template.setBlock(
        }
        print(template);
    }
}
```

If the name hasn't been provided, set the appropriate template block to change the CSS class (I didn't use the validation API for demo purposes).

Component implementation (Element)

```
public class Home extends Element {

    private Template    template;
    private String      name;

    public void setTemplate(Template template) { this.template = template; }
    public void setName(String name) { this.name = name; }

    public void processElement() {
        print(template);
    }

    public void doMyData() {
        if (null == name || 0 == name
            template.setBlock("name class", "class error");
        } else {
            String day = new SimpleDateFormat("EEEE").format(new Date());
            template.setValue("day", encodeHtml(day));
            template.setBlock("content", "welcome");
        }
        print(template);
    }
}
```

Otherwise, get the current weekday, set it in the template and display the welcome message.

Component implementation (Element)

- Very intuitive
- Doesn't have to worry about state handling, form submission, application flow, ... since the engine takes care of it all
- Regular Java with few artifacts
- Can be written in scripting languages: Beanshell, Groovy, Janino, Jython, Pnuts, Rhino (javascript) and Tcl
- Each element is totally decoupled from any other element and the wiring happens in the site-structure

XHtml Template

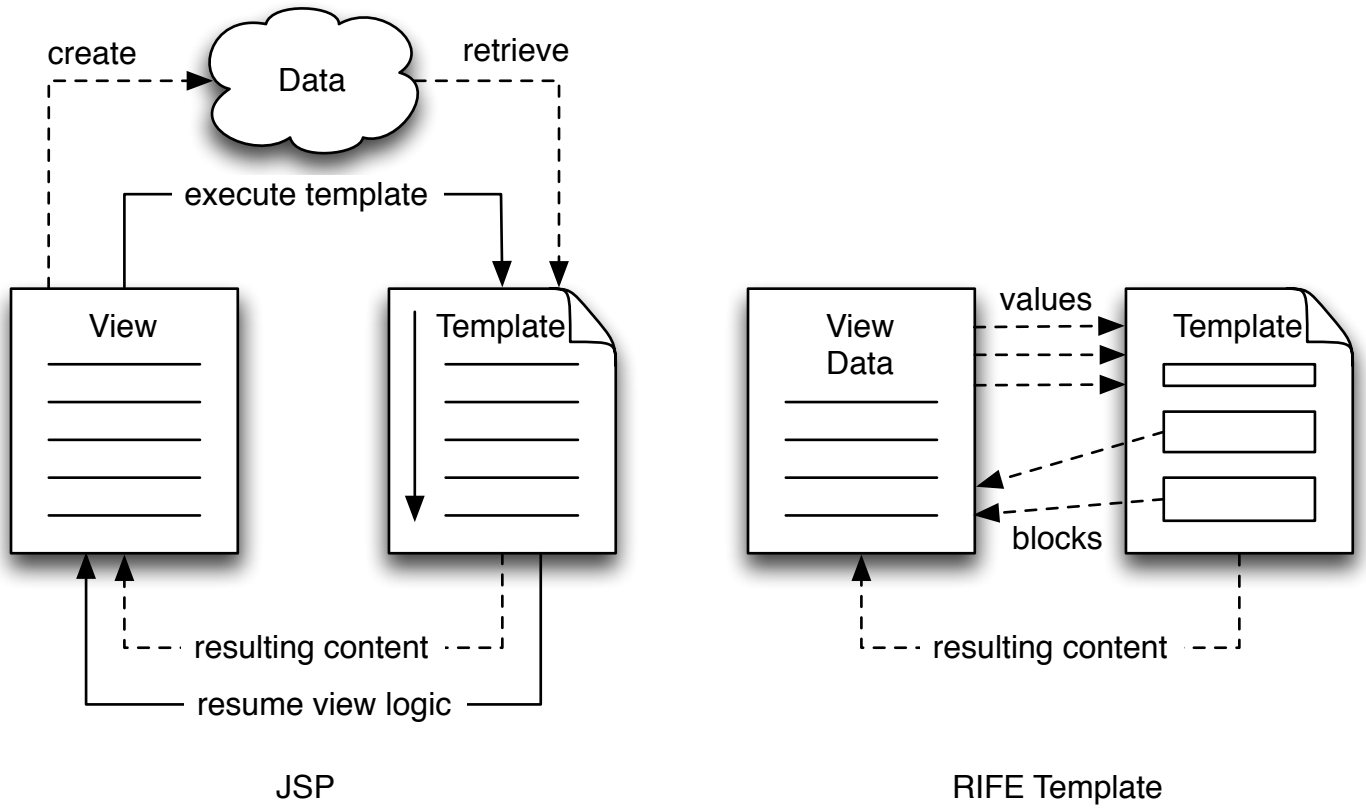
XHtml Template

```
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
<head>
    <base href="[!V 'WEBAPP:ROOTURL'/]" />
    <title>Who are you?</title>
    <style type="text/css">.error { background: #ffcccc; padding: 4px; }</style>
</head>
<body>
    <!--V 'content'/-->

    <!--BV 'content'-->
        <form action="[!V 'SUBMISSION:FORM:myData'/]" method="post">
            <!--V 'SUBMISSION:PARAMS:myData'/-->
            <div class="[!V 'name class'][!/V][!B 'class error']error[!/B]">
                <input name="name" value="[!V 'PARAM:name'][!/V]" />
            </div>
            <input type="submit" value="That's who I am!" />
        </form>
    <!--/BV-->

    <!--B 'welcome'-->
        <div>Welcome <!--V 'PARAM:name'/--> on this nice <!--V 'day'/-->!</div>
    <!--/B-->
</body>
</html>
```

XHtml Template

- **Don't worry, it's very intuitive once you get used to the concepts and the notation** (if you must, you can change it)
- **Logic-less, unintrusive template engine that's not bound to the format of the language** (eg. DOM)
- **You mark up existing layouts and indicate content placeholders and design blocks**
- **You assemble the templates in plain Java**
- **A blueprint that contains everything you'll need to build the final layout**

Comparison with JSP



Let's go over it step-by-step

XHtml Template

```
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
<head>
    <base href="[!V 'WEBAPP:ROOTURL'/]" />
    <title>Who are you?</title>
    <style type="text/css">.error { background: #ffcccc; padding: 4px; }</style>
</head>
<body>
    <!--V 'content'/-->

    <!--BV 'content'-->
        <form acti                                                    st">
            <!--V
            <div c                                                    error[!/B]">
                <i                                                    " />
            </div>
            <input type="submit" value="That's who I am!" />
        </form>
    <!--/BV-->

    <!--B 'welcome'-->
        <div>Welcome <!--V 'PARAM:name'/--> on this nice <!--V 'day'/-->!</div>
    <!--/B-->
</body>
</html>
```

This is a regular XHtml document.
(the document type is missing due to space constraints)

XHtml Template

```
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
<head>
    <base href="[!V 'WEBAPP:ROOTURL'/]" />
    <title>Who are you?</title>
    <style type="text/css">.error { background: #ffcccc; padding: 4px; }</style>
</head>
<body>
    <!--V 'content
    <!--BV 'conten
        <form acti
            <!--V
            <div c
                <i
            </div>
            <input
        </form>
    <!--/BV-->
    <!--B 'welcome
        <div>Welco
    <!--/B-->
</body>
</html>
```

This is a regular XHtml header apart from one value tag.

RIFE has a series of standard value tag names that will be automatically replaced.

In this case, it will contain the root URL of your web application and make it completely relocatable.

XHtml Template

```
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
<head>
    <base href="[!V 'WEBAPP:ROOTURL'/]" />
    <title>Who are you?</title>
    <style type="text/css">.error { background: #ffcccc; padding: 4px; }</style>
</head>
<body>
    <!--V 'content'/-->

    <!--BV 'content'-->
        <form action="[!V 'SUBMISSION:FORM:myData'/]" method="post">
            <!--V 'SUBMI
            <div class="
                <input n
            </div>
            <input type="submit" value="That's who I am!" />
        </form>
    <!--/BV-->

    <!--B 'welcome'-->
        <div>Welcome <!--V 'PARAM:name'/--> on this nice <!--V 'day'/-->!</div>
    <!--/B-->
</body>
</html>
```

A value tag (V) indicates where content can be inserted.

XHtml Template

```
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
<head>
    <base href="[!V 'WEBAPP:ROOTURL'/]" />
    <title>Who
    <style ty
</head>
<body>
    <!--V 'co
    <!--BV 'c
        <form
            <
            <
            </div>
            <input type="submit" value="That's who I am!" />
        </form>
    <!--/BV-->

    <!--B 'welcome'-->
        <div>Welcome <!--V 'PARAM:name'/--> on this nice <!--V 'day'/-->!</div>
    <!--/B-->
</body>
</html>
```

A block tag (B) marks up a piece of reusable content that can contain values of its own.

The block will be stripped from the template.

The standard PARAM value tag will contain the value of the submission parameter after it has been sent.

XHtml Template

```
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
<head>
    <base href="[!V
    <title>Who are yo
    <style type="tex                                    4px; }</style>
</head>
<body>
    <!--V 'content'/-->

    <!--BV 'content'-->
        <form action="[!V 'SUBMISSION:FORM:myData'/]" method="post">
            <!--V 'SUBMISSION:PARAMS:myData'/-->
            <div class="[!V 'name class'][!/V][!B 'class error']error[!/B]">
                <input name="name" value="[!V 'PARAM:name'][!/V]" />
            </div>
            <input type="submit" value="That's who I am!" />
        </form>
    <!--/BV-->

    <!--B 'welcome'-->
        <div>Welcome <!--V 'PARAM:name'/--> on this nice <!--V 'day'/-->!</div>
    <!--/B-->
</body>
</html>
```

A block-value tag (BV) is a regular block, but automatically sets its content to a value with the same name.

XHtml Template

Forms are regular XHtml.

There are two specific value tags that will contain the URL and the hidden parameters to handle the flow and state transitions.

```
<!--BV 'content'-->
    <form action="[!V 'SUBMISSION:FORM:myData'/]" method="post">
        <!--V 'SUBMISSION:PARAMS:myData'/-->
        <div class="[!V 'name class'][!/V][!B 'class error']error[!/B]">
            <input name="name" value="[!V 'PARAM:name'][!/V]" />
        </div>
        <input type="submit" value="That's who I am!" />
    </form>
<!--/BV-->

<!--B 'welcome'-->
    <div>Welcome <!--V 'PARAM:name'/--> on this nice <!--V 'day'/-->!</div>
<!--/B-->
```

XHtml Template

```
<html xmlns="http://www.w3.or
<head>
    <base href="[!V 'WEBAPP:R
    <title>Who are you?</title>
    <style type="text/css">.e                                    yle>
</head>
<body>
    <!--V 'content'/-->

    <!--BV 'content'-->
        <form action="[!V 'SUBMISSION:FORM:myData'/]" method="post">
            <!--V 'SUBMISSION:PARAMS:myData'/-->
            <div class="[!V 'name class'][!/V][!B 'class error']error[!/B]">
                <input name="name" value="[!V 'PARAM:name'][!/V]" />
            </div>
            <input type="submit" value="That's who I am!" />
        </form>
    <!--/BV-->

    <!--B 'welcome'-->
        <div>Welcome <!--V 'PARAM:name'/--> on this nice <!--V 'day'/-->!</div>
    <!--/B-->
</body>
</html>
```

Input fields are regular XHtml.

The value will be automatically filled after a submission and is empty by default.

XHtml Template

```
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
<head>
    <base href="[!V 'WEBAPP:ROOTURL'/]" />
    <title>Who are you?</title>
    <style type="text/css">.error { b
</head>
<body>
    <!--V 'content'/-->

    <!--BV 'content'-->
        <form action="[!V 'SUBMISSION:FORM:myData'/]" method="post">
            <!--V 'SUBMISSION:PARAMS:myData'/-->
            <div class="[!V 'name class'][!/V][!B 'class error']error[!/B]">
                <input name="name" value="[!V 'PARAM:name'][!/V]" />
            </div>
            <input type="submit" value="That's who I am!" />
        </form>
    <!--/BV-->

    <!--B 'welcome'-->
        <div>Welcome <!--V 'PARAM:name'/--> on this nice <!--V 'day'/-->!</div>
    <!--/B-->
</body>
</html>
```

This value and block allows the view logic to update the class of the field's div to indicate an error.

Component implementation (Element)

```
public class Home extends Element {

    private Template    template;
    private String      name;

    public void setTemplate(Template template) { this.template = template; }
    public void setName(String name) { this.name = name; }

    public void processElement() {
        print(template);
    }

    public void doMyData() {
        if (null == name || 0 == name.trim().length()) {
            template.setBlock("name class", "class error");
        } else {
            String day = new SimpleDateFormat("EEEE").format(new Date());
            template.setValue("day", encodeHtml(day));
            template.setBlock("content", "welcome");
        }
        print(template);
    }
}
```

Component implementation (Element)

```
public class Home extends Element {

    private Template    template;
    private String      name;

    public void setTemplate(Template template) { this.template = template; }
    public void setName(String name) { this.name = name; }

    public void processElement() {
        print(template);
    }

    public void doMyData() {
        if (null == name || 0 == name.trim().length()) {
            template.setBlock("name class", "class error");
        } else {
            String day = new SimpleDateFormat("EEEE").format(new Date());
            template.setValue("day", encodeHtml(day));
            template.setBlock("content", "welcome");
        }
        print(template);
    }
}
```

XHtml Template

- **Only four tags**: V, B, BV and I
- Template **language independent**, can be used for XHtml, Html, Xml, Text, SQL, Java, ... templating
- **Invisible** markup that keeps the language valid
- Alternative **compact** markup that's less convoluted
- **Hierarchical** dynamic templates, using includes
- **Bi-directional**: from code to template and vice-versa
- **No logic!**

Agenda

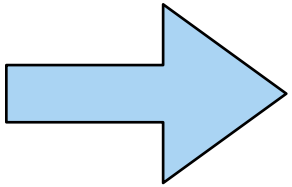
- Essentials
- A quick look
- **Metaprogramming**
- Q&A

What is metaprogramming?

Writing programs that write or manipulate other programs.

Benefits of metaprogramming

- You work with a domain specific API or language that allows you to build with larger blocks
- High-level approach to easily achieve otherwise complex or time-consuming tasks
- Still access to the underlying framework for customizations or finer-grained implementations



You can concentrate on your business logic and waste less time on trivialities

Introducing RIFE's Constraints

Rich dynamic meta-data API for Java bean instances and their properties

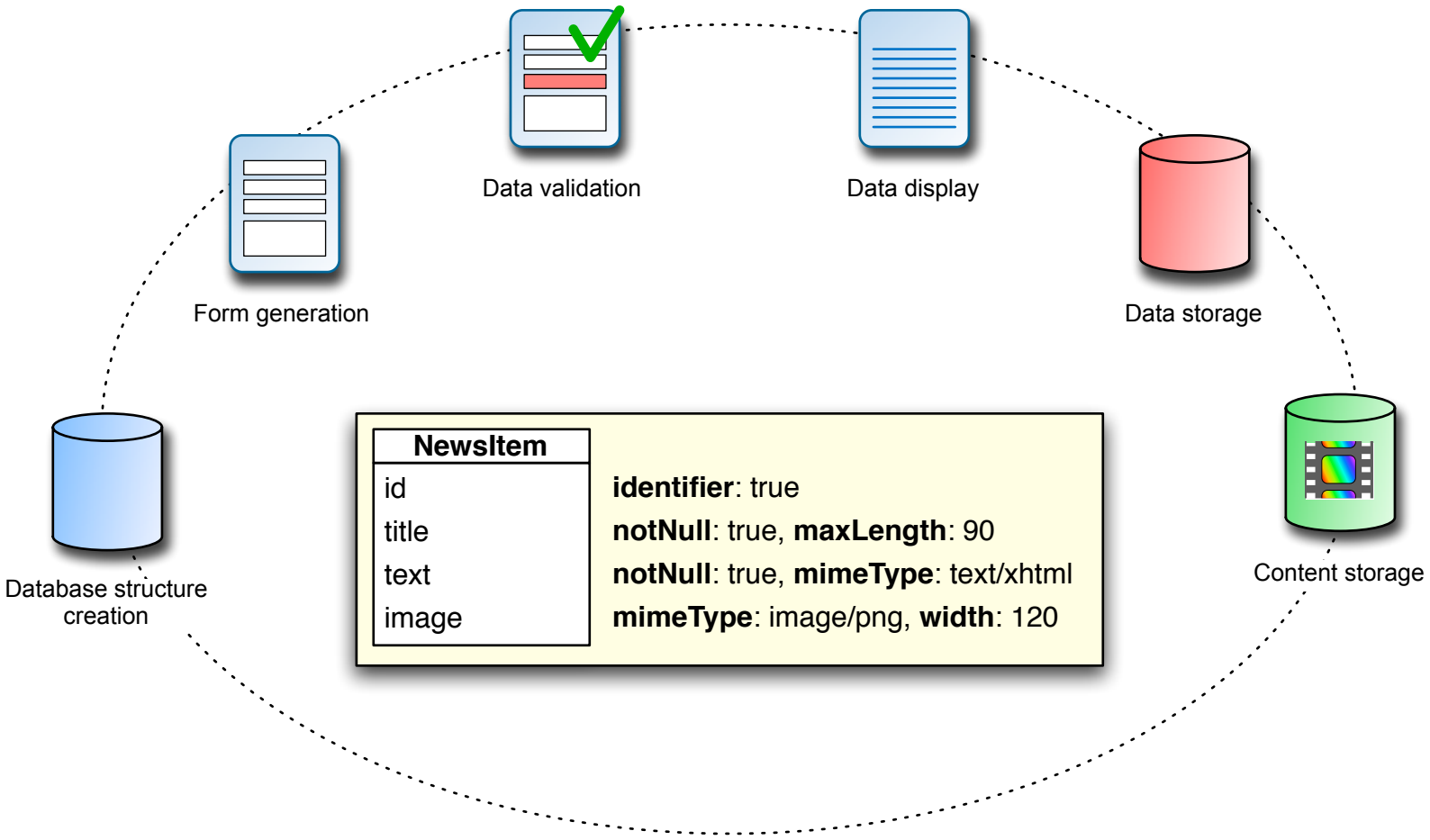
Introducing Constraints

- **Regular JavaBean instance**

Introducing Constraints

- **Regular JavaBean instance**
- **Additional Constraints that provide rich meta-data**

The entire framework adapts to the Constraints



Introducing Constraints

- **Unobtrusive: applies to any existing POJO**
- **When quickly writing a simple application, you can combine the bean and the constraints in one class**
- **Instance and runtime based: they can dynamically change at the appropriate moment** (ie. callbacks)

What do Constraints look like?

What do constraints look like?

- **Regular POJO JavaBean with a number of properties**

```
public class NewsItem {

    private Integer   id;
    private String    title;
    private String    text;
    private byte[]    image;

    public void    setId(Integer id)        { this.id = id; }
    public Integer getId()                  { return id; }
    public void    setTitle(String title)   { this.title = title; }
    public String  getTitle()               { return title; }
    public void    setText(String text)     { this.text = text; }
    public String  getText()                { return text; }
    public void    setImage(byte[] image)   { this.image = image; }
    public byte[]  getImage()               { return image; }
}
```

What do constraints look like?

- **Regular POJO JavaBean with a number of properties**

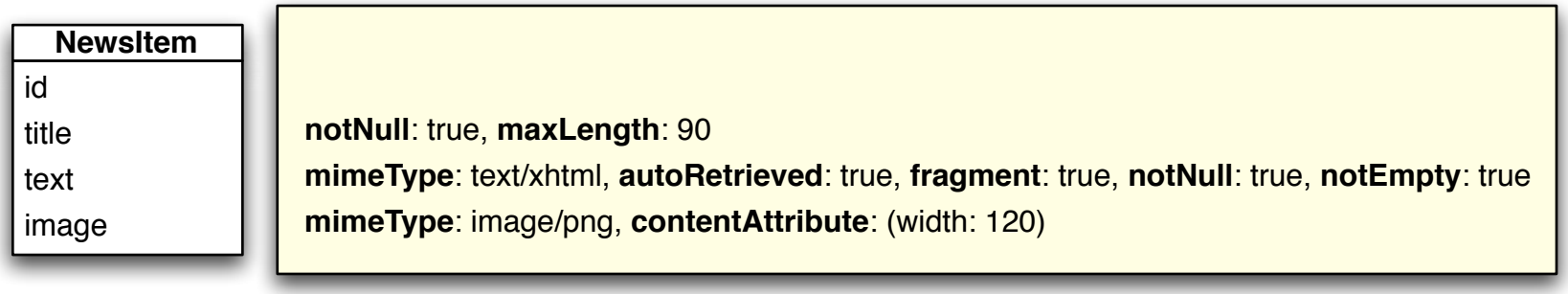
What do constraints look like?

- **Regular POJO JavaBean with a number of properties**
- **Constraints as additional rich meta-data**

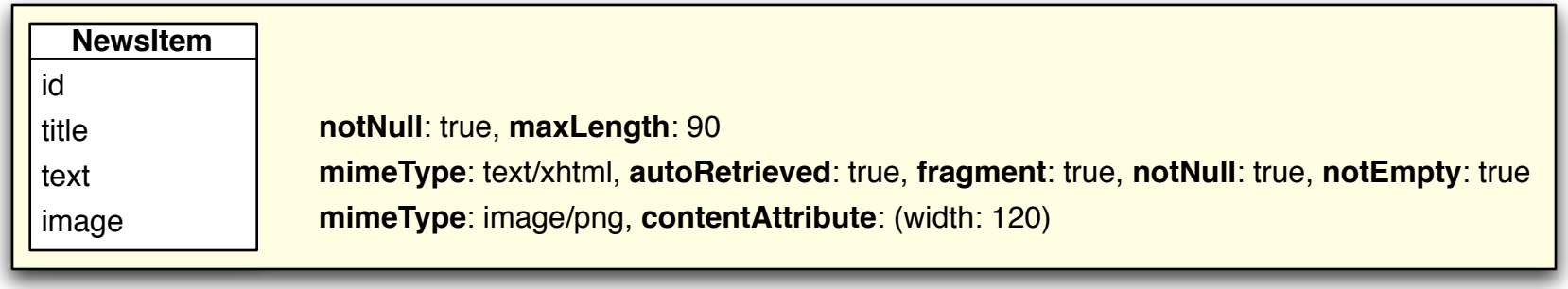

```
public class NewsItemMetaData extends MetaData {
    public void activateMetaData() {

        addConstraint(new ConstrainedProperty("title")
            .notNull(true)
            .maxLength(90));
        addConstraint(new ConstrainedProperty("text")
            .mimeType(MimeType.APPLICATION_XHTML)
            .autoRetrieved(true)
            .fragment(true)
            .notNull(true)
            .notEmpty(true));
        addConstraint(new ConstrainedProperty("image")
            .mimeType(MimeType.IMAGE_PNG)
            .contentAttribute("width", 120)
    }
}
```

What do constraints look like?

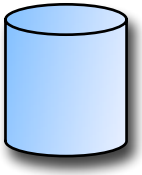
- **Regular POJO JavaBean with a number of properties**
- **Constraints as additional rich meta-data**



RIFE automatically combines both behind the scenes



... and this is how it's used



Database structure creation

```
new ContentQueryManager(datasource, NewsItem.class).install();
```

The database table (and if relevant, the sequence) will be created with the correct column names, types, precisions, constraints, primary keys, foreign keys, ...

... and this is how it's used

Form generation

```
<div class="form_field">
    <div><label for="title">Title</label></div>
    <div [!V 'MARK:title'][!/V]>
        <!--V 'ERRORS:title'--><!--/V-->
        <!--V 'FORM:INPUT:title'-->id="title"<!--/V-->
    </div>
</div>
<div class="form_field">
    <div><label for="text">Text</label></div>
    <div [!V 'MARK:text'][!/V]>
        <!--V 'ERRORS:text'--><!--/V-->
        <!--V 'FORM:TEXTAREA:text'-->id="text" cols="80"<!--/V-->
    </div>
</div>
<div class="form_field">
    <div><label for="body">Image</label></div>
    <div [!V
        <!--V
        <inpu
    </div>
</div>
```

These value tags will be replaced by actual XHtml tags with attributes that correspond to the constraints (eg. maxlength)

... and this is how it's used

Data validation

```
NewsItem newsitem = getSubmissionBean(NewsItem.class);

ContentQueryManager<NewsItem> manager =
    new ContentQueryManager(datasource,NewsItem.class);
if (newsitem.validate(manager)) {
    manager.save(newsitem);
} else {
    generateForm(template, newsitem);
}
```

The bean can be directly retrieved from the request and validated. Validating against its manager allows additional checks to be performed (like uniqueness, foreign key value existence, ...).

The form can be immediately regenerated with the known validation errors.

... and this is how it's used

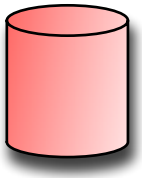
```
<div class="form_field">
    <div><label for="title">Title</label></div>
    <div [!V 'MARK:title'][!/V]>
        <!--V 'ERRORS:title'--><!--/V-->
        <!--V 'FORM:INPUT:title'-->id="title"<!--/V-->
    </div>
</div>
<div class="form_field">
    <div><label for="text">Text</label></div>
    <div [!V 'MARK:text'][!/V]>
        <!--V 'ERRORS:text'--><!--/V-->
        <!--V 'FORM:TEXTAREA:text'-->id="text" cols="80"<!--/V-->
    </div>
</div>
<div class="form_field">
    <div><label for="body">Image</label></div>
    <div [!V 'MARK:image'][!/V]>
        <!--V 'ERRORS:image'--><!--/V-->
        <input type="file" name="image"/>
</
```

Form generation

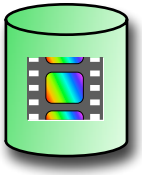
Data validation

Errors that were detected during validation will generate markings and error messages in the template. Their look can be totally customized with block tags.

... and this is how it's used



Data storage



Content storage

```
NewsItem newsitem = getSubmissionBean(NewsItem.class);

ContentQueryManager<NewsItem> manager =
    new ContentQueryManager(datasource,NewsItem.class);
if (newsitem.validate(manager)) {
    manager.save(newsitem);
} else {
    generateForm(template, newsitem);
}
```

The bean instance is saved with a simple method call and all rich content properties are automatically stored in the content management framework instead of in the table.

... we can do better though!

... we can do better though!

RIFE/Crud

Introducing RIFE/Crud

- **Automatically generates** administration functionalities for tedious, repetitive CRUD operations
- **No code generation**
- Completely **runtime-based**, reloads on-the-fly
- Driven by your **domain model beans**
- **Constraints** gives total control over the data and content formats
- Builds on RIFE's **component model**

Introducing RIFE/Crud

- **Can be integrated into any other RIFE site**
- **Fully customizable:**
  - built-in support for **localization**
  - layout through **well-structured CSS**
  - markup is driven by **templates**,
    they can be replaced in different levels of granularity
  - **CRUD API** for reusing most of the automated
    run-time features during customization
  - **replacement** of individual CRUD components
  - customizable **menu generation**
  - features like **element enhancements** and **callbacks** are still available

Quick Demo

Same NewsItem bean with some additions

```
public class NewsItem {

    private Integer  id;
    private Date     moment = Calendar.getInstance().getTime();
    private Integer  categoryId;
    private String   title;
    private String   text;
    private byte[]   image;
    // ... accessors
}
```

```
addConstraint(new ConstrainedBean()
    .defaultOrder("moment", ConstrainedBean.DESC)
    .defaultOrder("title"));

addConstraint(new ConstrainedProperty("id")
    .editable(false).identifier(true));

addConstraint(new ConstrainedProperty("moment")
    .editable(false).notNull(true)
    .listed(true)
    .format(DateFormat.getDateTimeInstance()));

addConstraint(new ConstrainedProperty("categoryId")
    .notNull(true).listed(true)
    .manyToOne(Category.class, "id"));
```

```
addConstraint(new ConstrainedProperty("title")
    .notNull(true).notEmpty(true)
    .maxLength(90).listed(true));

addConstraint(new ConstrainedProperty("text")
    .notNull(true).notEmpty(true)
    .mimeType(MimeType.APPLICATION_XHTML)
    .autoRetrieved(true).fragment(true));

addConstraint(new ConstrainedProperty("image")
    .mimeType(MimeType.IMAGE_PNG)
    .contentAttribute("width", 120)
    .listed(true)
    .file(true));
```

New Category bean

```
public class Category {

    private Integer id;
    private String  name;
    // ... accessors
}
```

```
addConstraint(new ConstrainedBean()
    .defaultOrder("name")
    .textualIdentifier(new AbstractTextualIdentifierGenerator<Category>() {
            public String generateIdentifier() {
                return getId() + " : " + getName();
            }
        }));
    }

addConstraint(new ConstrainedProperty("id")
    .editable(false).identifier(true));

addConstraint(new ConstrainedProperty("name")
    .notEmpty(true).notNull(true).maxLength(255)
    .listed(true));
```

We just need to make RIFE aware of them

- **Site structure declaration** (with the crud handler)

```
<site>
    <arrival destid="News"/>

    <subsite id="News" file="crud:Newsitem" urlprefix="news"/>
    <subsite id="Category" file="crud:Category" urlprefix="category"/>
</site>
```

- **Additional repository participants to create the database structure**

```
<rep>
    <!-- snip -->
    <participant param="Category">CreateCrudStructureParticipant</participant>
    <participant param="Newsitem">CreateCrudStructureParticipant</participant>
    <!-- snip -->
</rep>
```

Done!

Done!

This is what it looks like after adding some localization

Questions