Request mapping

Wicket 1.4

To mount a page in Wicket 1.4 the developer had to use *org.apache.wicket.protocol.http.WebApplication*'s:

- `#mount(IRequestTargetUrlCodingStrategy)`
- `#mount(String, PackageName)`
- `#mountBookmarkablePage(String, Class<T>)`
- `#mountBookmarkablePage(String, String, Class<T>)`
  And to mount a resource:
- `#mountSharedResource(String, String)`
  For more information about these methods check URL Coding Strategies.

Wicket 1.5

`IRequestTargetUrlCodingStrategy` interface and all its implementations are replaced with the `IRequestMapper` and its respective implementations.

To add a mapper into the list of mappers which Wicket will use to process a request use `Application.getRootRequestMapperAsCompound().add(mapperInstance)`.
When a request comes Wicket will ask all registered mappers whether they are able to process the request. Mappers with bigger `IRequestMapper.getCompatibilityScore(Request)` are asked first. So Wicket calls `IRequestMapper.mapRequest(Request)` for each mapper and if it returns non-null `IRequestHandler` then this is the handler which will be used to process the current request. In `#mapRequest(Request)` the mapper have to check request's segments (this is similar to `httpServletRequest#getPath()`) and request's parameters (GET and POST) and decide whether they match to the mapper's functionality. For example, `HomePageMapper` is the mapper used to process all requests without any segments, i.e. requests to '/' with or without any query parameters.
The actual processing of the request is being done with `IRequestHandler.respond(IRequestCycle)`. During the processing Wicket asks the mappers to create a `Url` object for each callback handler (e.g. link, form, ...) via `IRequestMapper.mapHandler(IRequestHandler)`.

Sometimes you may want a specific IRequestMapper to be process all incoming requests. To do this you should use `Application.setRootRequestMapper(IRequestMapper)`. This mapper may manipulate the Request's URL and then pass it for further processing to the registered non-root mappers. For examples of this idea see the source code of `CryptoMapper` and `HttpsMapper`.

Default mapper implementations

HomePageMapper

This mapper is pre-configured by Wicket and there is no need to register it. It is used to create `IRequestHandler` for requests to the root ('/') of the application context.

BookmarkableMapper

This mapper decodes and encodes bookmarkable URLs like:

- /wicket/bookmarkable/com.example.pages.MyPage - using `BookmarkablePageRequestHandler` for stateless pages and using RenderPageRequestHandler for stateful/hybrid pages
- /wicket/bookmarkable/com.example.pages.MyPage?2-click-foo-bar-baz - using `BookmarkableListenerInterfaceRequestHandler` to process bookmarkable listeners (e.g. Behavior).

This mapper is also pre-configured and there is no need to register it.

To change 'wicket' and 'bookmarkable' segments in the URL to something else see `IMapperContext.getNamespace()` (the default implementation can be replaced with `Application.newMapperContext()`).

MountedMapper

This mapper is similar to `BookmarkableMapper` but the difference is that the user application defines the mount point where this mapper matches.
For example:

- */path/to/page1*
- */path/to/pageN*
- */path/to/page?2-5.click.1-foo-bar-baz* (2 is the page version, 5 is render count, 1 is behavior index)

Usage:

MyApp.java

```
public void init() {
        super.init();

        getRootRequestMapperAsCompound().add(new MountedMapper("/mount/point", MyPage.class));
        mountPage("/mount/point", MyPage.class); // convenient method doing the same as above
}
```

This mapper is a combination of all `IRequestTargetUrlCodingStrategy` implementations from Wicket 1.4. It supports:

Indexed parameters - /page/idx1/idx2

`mountPage("/page", MyPage.class);`
Now a request to "/page/a/b/c" will be handled by MyPage and the parameters can be get with `PageParameters.get(int)` (e.g. *parameters.get(2)* will return "c")

Named parameters - /page/${named1}/${named2}

`mountPage("/page/${named1}/${named2}", MyPage.class);`
Now a request to "/page/a/b" will be handled by MyPage and the parameters can be get with `PageParameters.get(String)` (e.g. *parameters.get ("named1")* will return "a")

Optional named parameters - /page/${named1}/#{named2\

`mountPage("/page/${named1}/#{named2}", MyPage.class);`
This means the second parameter is optional. Requests to "/page/a/b" , "/page/a/b/" and "/page/a/" will be handled by MyPage and the parameters can be get with `PageParameters.get(String)` (e.g. *parameters.get("named2")* will return "b" for the first case and ***null*** for the second).
The mapper is smart enough to handle optional named parameters in any segment, not just the last one.

Arbitrary named parameters - /page/param1Name/param1Value/param2Name/param2Value

`mount(new MountedMapper("/page", MyPage.class, new UrlPathPageParametersEncoder()));`
Now a request to "/page/a/1/b/2" will be handled by MyPage and the parameters can be get with `PageParameters.get(String)` (e.g. *parameters.get ("a")* will return "1")

Query parameters - /page?param1Name=param1Value&param2Name=param2Value

`mountPage("/page", MyPage.class);`
Now a request to "/page?a=a1&b=b1" will be handled by MyPage and the parameters can be get with `PageParameters.get(String)` (e.g. *parameters. get("a")* will return "a1")

The mapper can handle a mix of the supported parameters - indexed + named + query.

PackageMapper

This mapper can mount a whole package. That is you mount a single page with a mount path prefix and then the mapper knows how to map all `Page` implementations in that package.

Usage:

MyApp.java

```
public void init() {
        super.init();

        getRootRequestMapperAsCompound().add(
                        new MountMapper("/mount/point", new PackageMapper(
                                PackageName.forClass(Page3.class))));
        mountPackage("/mount/point", Page3.class);
}
```

Assuming that PageA package is "com.example.pages" a request to "/mount/point/PageB" will use com.example.pages.PageB if it exists and is an instance of Page.

ResourceMapper

A mapper which mounts `ResourceReference` implementations.

Usage:

MyApp.java

```
public void init() {
        super.init();

        getRootRequestMapperAsCompound().add(new ResourceMapper("/company/logo", new PackageResourceReference
(MyPage.class, "res/logo.gif")));
        mountResource("/company/logo", new PackageResourceReference(MyPage.class, "res/logo.gif"))); //
convenient method doing the same as above
}
```

CryptoMapper

A wrapper around another mapper which will encrypt/decrypt the URLs generated by the inner one.

Usage:

MyApp.java

```
public void init() {
        super.init();

        IRequestMapper cryptoMapper = new CryptoMapper(getRootRequestMapper(), this);
        setRootRequestMapper(cryptoMapper);
}
```

HttpsMapper

A mapper which makes a redirect to the same URL with HTTPS protocol if the requested page is annotated with @RequireHttps or to HTTP protocol if the last processed page had @RequireHttps and the one going to be processed has no such annotation.

Usage:

```
public class MyApplication extends WebApplication
{
         public void init()
         {
                 super.init();

                 getRootRequestMapperAsCompound().add(new MountedMapper("secured", HttpsPage.class));

                 setRootRequestMapper(new HttpsMapper(getRootRequestMapper(), new HttpsConfig(80, 443)));
         }
}
```

Since Wicket 6.x HttpsMapper can be easily extended

The `HttpsMapper` can now be subclassed. This means that by overriding the `HttpsMapper.getDesiredScheme` method you can programmatically determine what scheme to use.
Add this to your Application.java:

```
public void init() {
    super.init();
    setRootRequestMapper(new HttpsMapper(getRootRequestMapper(), new HttpsConfig(80, 443)) {
        @Override
        protected Scheme getDesiredSchemeFor(Class<? extends IRequestablePage> pageClass) {
            if (getConfigurationType()==RunTimeConfigurationType.DEVELOPMENT)
                return Scheme.HTTP;
            else
                return super.getDesiredSchemeFor(pageClass);
        }
    }
```

Making all URLs absolute

See page Making all URLs absolute