Departamento
Engenharia
Informática

Modelação
Engenharia de Software
Sistemas Distribuídos

Livro de Receitas

Framework de aplicações com Web Services

2009-04-17 Versão 1.1

Índice

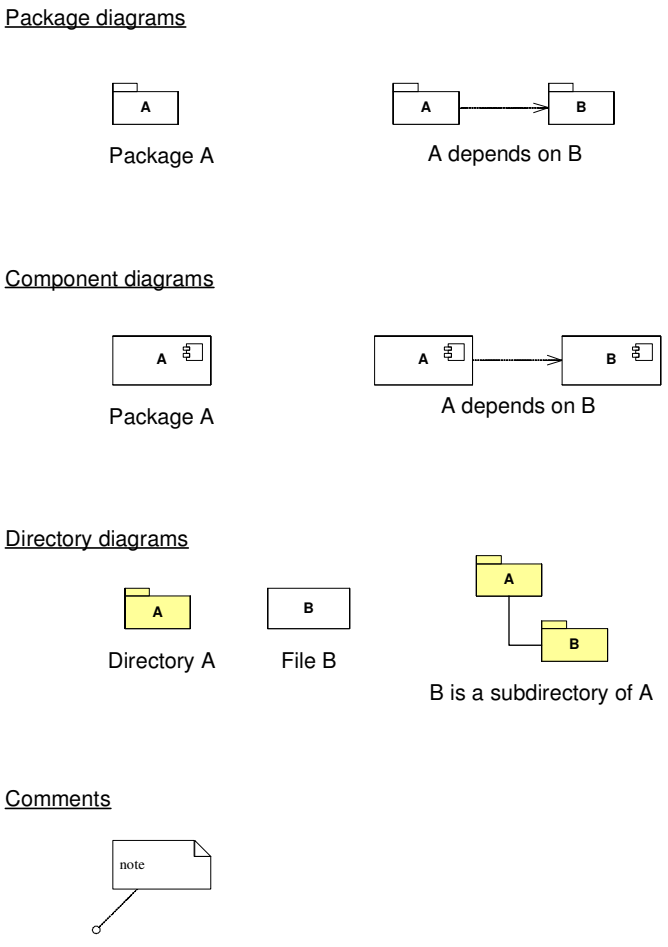
Nota prévia

Este documento visa guiar a equipa de desenvolvimento no uso da *framework* utilizada no projecto conjunto de Engenharia de Software e Sistemas Distribuídos. Embora a *framework* seja de uso genérico, o livro de receitas que se segue é direccionado tendo em conta as especificidades do projecto a desenvolver.
As *receitas* aqui descritas são um complemento das descritas no "Livro de Receitas *Framework* de aplicações web" e assumem conhecimento da tecnologia de Web Services.

Convenções

`$`	Indica a execução de um comando na consola.
`Enunciado.pdf`	Indica texto (por exemplo: nomes de classes, ficheiros, caminhos, comandos do sistema operativo) que deve ser introduzido exactamente como apresentado.
`filename``.txt`	Indica um elemento variável que é necessário substituir por um valor concreto.

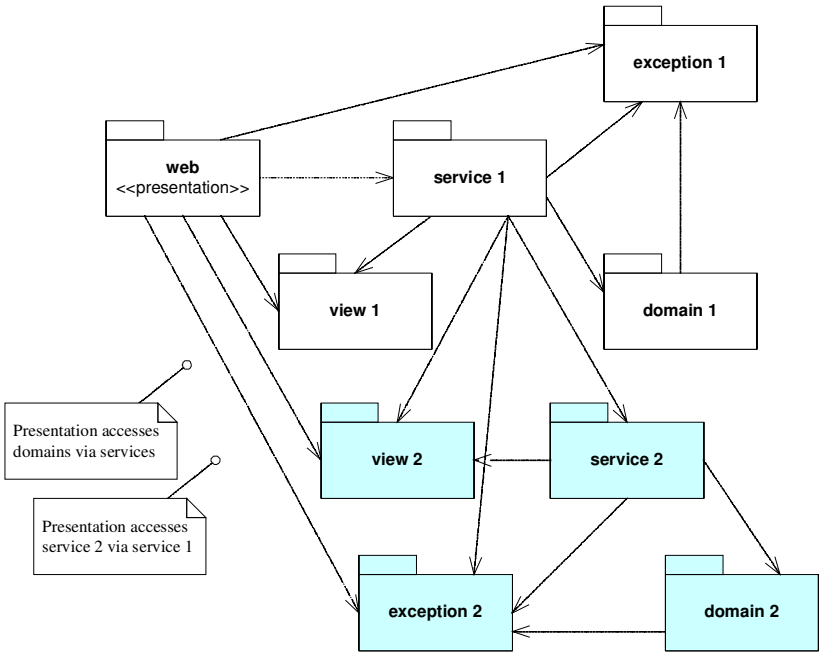
Legenda



No texto: A-->B significa que A depende de B

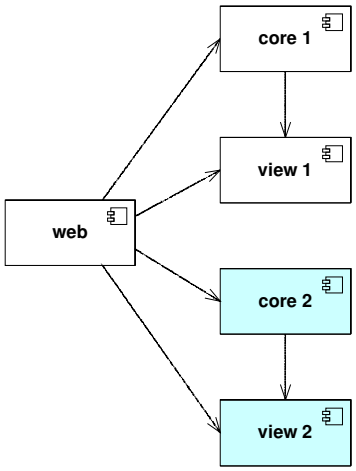
Breve introdução à distribuição de aplicações

As aplicações da framework podem estar estruturadas em 2 ou mais domínios **independentes**. Para uma aplicação Web com **2 domínios** temos os seguintes **pacotes**:

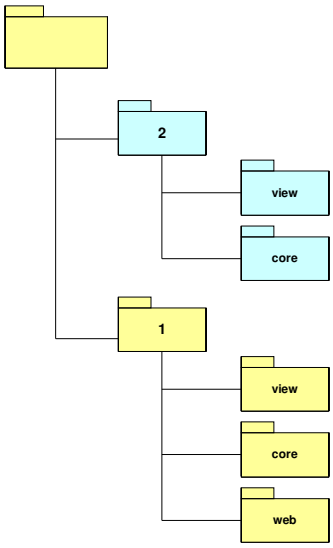


A camada de apresentação (web) não acede directamente aos domínios. Acede indirectamente através das vistas (view1 e view2) e dos serviços (service1 e service2).
Na camada de apresentação surgem novos serviços que usam serviços dos dois domínios de forma coordenada.

No que respeita a **componentes** a organização é a seguinte:



E a arrumação em **directórios** é feita da seguinte forma:

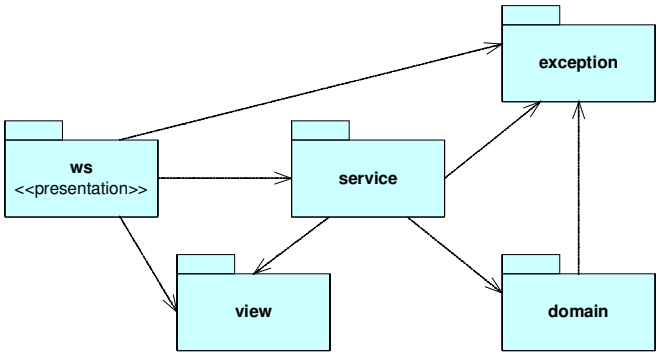


Até aqui temos uma **separação lógica** entre domínios, pois na realidade a aplicação executa-se no contexto da mesma máquina virtual Java e usa a mesma base de dados.

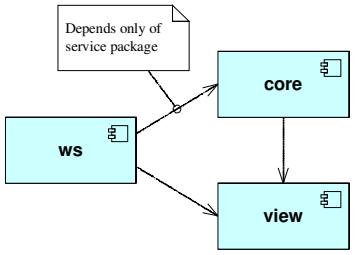
Pretende-se agora ter **separação física** entre os domínios, ou seja, cada domínio vai ser uma aplicação independente, a executar-se numa máquina virtual Java própria e a usar uma base de dados própria.

As aplicações vão comunicar entre si através de **Web Services**.

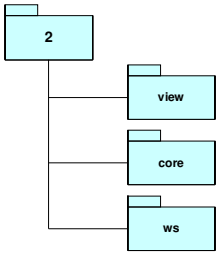
O domínio 2 vai tornar-se independente como um Web Service. Os **pacotes** da aplicação 2 serão os seguintes:



Os **componentes** da aplicação 2 serão:



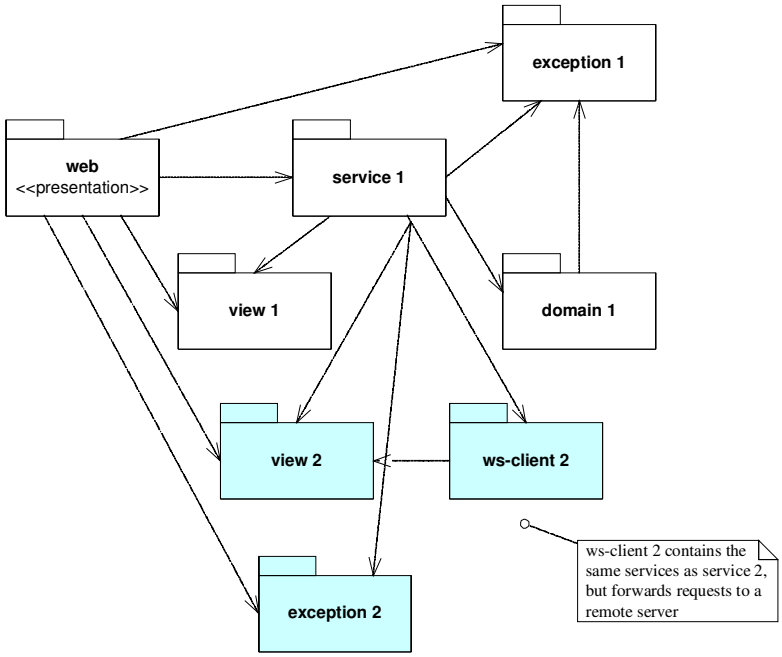
E os **directórios** serão:



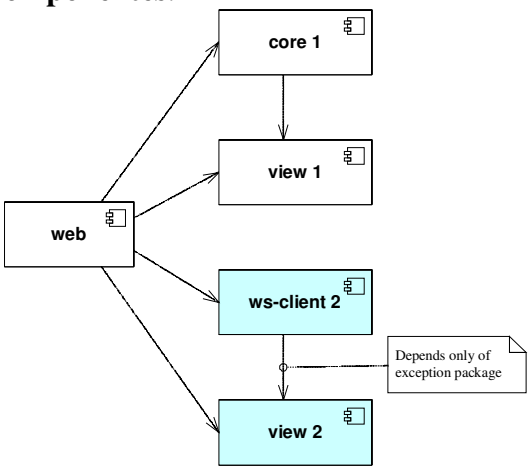
Depois de construída e instalada num servidor aplicacional (o Tomcat, por exemplo), a aplicação 2 está disponível para invocação e pode ser usada por qualquer cliente de Web Services.

Para facilitar a utilização da aplicação 2 por outras aplicações da framework (como a aplicação 1, por exemplo), é necessário acrescentar o pacote **ws-client**.

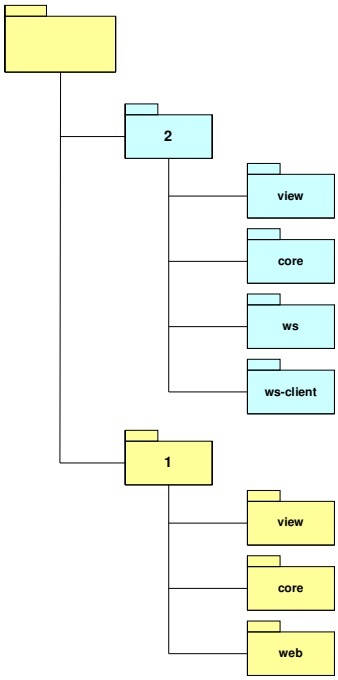
Após a distribuição física de domínios, os **pacotes** da aplicação 1 passam a ser:



O que corresponde aos **componentes**:



E aos **directórios**:



Distribuir uma aplicação web usando Web Services

Distribuir uma aplicação web usando Web Services

Estratégia

A separação de domínios de uma aplicação é muitas vezes motivada pela necessidade de uma separação de responsabilidade pela gestão de cada domínio e dos seus recursos (dados, servidores, etc.).
Pretende-se permitir que cada domínio seja uma aplicação independente, de modo a que possa ser gerido por uma diferente unidade organizacional.
Para tornar esta divisão de responsabilidades efectiva, é necessário estender a arquitectura das aplicações com Web Services - servidor e cliente - que vão permitir a invocação remota de serviços.

Passos básicos

1. Disponibilizar serviços como Web Service (componente ws).
2. Disponibilizar serviços remotos de invocação remota (componente ws-client).
3. Alterar processo de construção de aplicação para usar domínios remotos.
4. Alterar processo de teste de aplicação para usar domínios remotos.

Ver

Disponibilizar serviços como Web Service (componente ws)
Disponibilizar serviços remotos de invocação remota (componente ws client)
Alterar processo de construção de aplicação para usar domínios remotos
Alterar processo de teste de aplicação para usar domínios remotos

Disponibilizar serviços como Web Service (componente ws)

Estratégia

Para que a funcionalidade de uma aplicação possa ser invocada a partir de outra aplicação é necessário disponibilizar um conjunto de serviços através de um Web Service.

Os procedimentos de construção de um Web Service são aqui apresentados de forma resumida. Para uma descrição mais detalhada de cada passo, consultar o *cookbook* de Java Web Services.

Passos básicos

1. Criar o componente `ws`. Este componente terá código fonte XML (`src/xml`) para definir o Web Service e código fonte Java (`src/java`) para a implementação do Web Service.
2. Criar os ficheiros de configuração habituais nas seguintes pastas: `config/database`, `config/jax-ws-server`, `config/resources`.
3. Criar uma nova base de dados para o *`projecto`*. Editar o `build.properties` para mudar o nome da base de dados.
4. Criar o `build.xml` do componente `ws` (`ws/build.xml`) a partir de um exemplo, modificando o que for necessário.
5. Editar o `build.xml` para criar o *target* `-replace-hibernate-custom-tokens(dir)` para definir os *mappings* de todas as classes de domínio de forma idêntica ao componente `core`. Fazer `ant generate-db-schema` e restantes tarefas relacionadas com o preenchimento da base de dados.
6. Editar o `build.xml` para referenciar o XSD da `view` através da definição `jax.external`. Referenciar o JAR da `view` e o JAR do `core` através da definição `jar.external`.
7. Criar o ficheiro *`projecto.wsdl`* que estipula o contrato do Web Service. Este WSDL deverá definir uma operação por cada serviço a disponibilizar para invocação remota. Deverá definir também *faults* para as situações de erro: `ServiceError` e a *`ProjectoFault`* que mapeiam, respectivamente, erros de sistema e excepções de domínio. Os tipos deverão ser baseados no XSD da view e acrescentar um `complexType` e `element` para cada pedido, resposta e *fault*. O tipo correspondente a *`ProjectoFault`* deverá conter o campo de texto `faultType`, para levar o tipo de erro até aos clientes, além da mensagem de erro.

```
...
    <xsd:complexType name="ServiceError" />
    <xsd:element name="serviceError" type="tns:ServiceError" />

    <xsd:complexType name="ProjectoFault">
        <xsd:sequence>
            <xsd:element name="faultType" type="xsd:string"/>
        </xsd:sequence>
    </xsd:complexType>
    <xsd:element name="projectoFault" type="tns:ProjectoFault" />
...
    <message name="projectoFault">
        <part name="fault" element="tns:projectoFault" />
    </message>
    <message name="serviceError">
        <part name="fault" element="tns:serviceError" />
    </message>
...
    <portType ...
        <operation ...
            <input ...
            <output ...
            <fault message="tns:projectoFault" name="projectoFault"/>
            <fault message="tns:serviceError" name="serviceError"/>
        </operation>
        ...
...
    <binding ...
        ...
        <operation ...
            ...
            <fault name="projectoFault">
                <soap:fault name="projectoFault" use="literal" />
            </fault>
            <fault name="serviceError">
                <soap:fault name="serviceError" use="literal" />
            </fault>
        </operation>
        ...
```

Um WSDL é uma cadeia de definições que têm que ser consistentes: *Service –> Port –> Binding –> PortType –> Message –> Element –> ComplexType*

8. Fazer `ant build-jax-ws-server-ties` e consultar o código Java gerado em `build/jax-ws-server/wsimport`. Observar, em particular o ficheiro *`ProjectoPortType.java`*. É esta a interface Java que terá que ser implementada.
9. Implementar o Web Service criando a classe *`ProjectoWebServiceImpl.java`*. que implementa a interface *`ProjectoPortType`*. Esta classe irá ter um método Java para cada operação WSDL.

   Em cada método, deverá chamar o serviço local correspondente (por exemplo, a implementação de uma operação *`sayHello`* deverá chamar o serviço *`HelloService`* que concretiza a funcionalidade pretendida).

Todo o corpo do método que implementa a operação terá que ter um bloco *try-catch* que: apanha `ProjectoDomainException` e a lança dentro de uma `ProjectoFault_Exception`; qualquer outra `Exception` deve ser apanhada e lançada dentro de uma `ServiceError_Exception`.

10. O componente `ws` está concluído. Fazer `ant deploy` para construir tudo e instalar.

Variantes

Em caso de erro

Os erros propagam-se ao longo da cadeia de dependências. No caso de um Web Service, as dependências são:

- *ProjectoWebServiceImpl.java* --> *projecto.wsdl*
- ws-->view
- ws-->core.

Tipicamente, a melhor estratégia de identificação e correcção de erros é olhar para o **primeiro** erro. A sua **mensagem** é importante, por isso deve ser lida atentamente e interpretada na perspectiva do componente que está a falhar.

Disponibilizar serviços remotos de invocação remota (componente ws client)

Estratégia

De forma a permitir uma fácil integração com aplicações framework que são clientes de Web Services, é necessário construir o componente **ws-client**, que disponibiliza uma implementação alternativa dos serviços de `core` que não fazem mais do que invocar remotamente os serviços originais.
A ideia principal é que os *stubs* (que existem em qualquer cliente de Web Service) são estruturados e disponibilizados como serviços da framework.

Passos básicos

1. Criar o componente `ws-client`. Este componente terá código fonte Java (`src/java`) com serviços de invocação remota, que utilizam *stubs* gerados pela ferramenta `wsimport` do JAX-WS.
2. Criar os ficheiros de configuração habituais nas seguintes pastas: `config/jax-ws-client`, `config/resources`.
3. Criar o `ws-client/build.xml` a partir de um exemplo, modificando o que for necessário.
4. Editar `ws-client/build.xml` para referenciar o XSD da `view` e o WSDL do `ws` através da definição `jax.external`. Referenciar o JAR da `view` através da definição `jar.external`.
5. Fazer `ant build-jax-ws-client-stubs` e consultar o código Java gerado em `build/jax-ws-client/wsimport`. Observar, em particular os ficheiros *`ProjectoService.java`* e *`ProjectoPortType.java`*.
6. Implementar a fábrica de *stubs* cuja responsabilidade é centralizar a criação e configuração de *stubs* JAX-WS. Para este efeito, recomenda-se esta classe seja um Singleton. O nome da classe deverá ser *`projecto`*`.ws.client.`*`ProjectoStubFactory`*. A classe deverá herdar de `step.framework.ws.StubFactory` especificando os tipos genéricos `S` e `T` como *`ProjectoService`* e *`ProjectoPortType`*, respectivamente. Os métodos a implementar são, pelo menos, `getService()` e `getPort()`.
7. Implementar os serviços. Para cada classe de serviço em `core` no pacote `projecto.service`, criar uma classe de serviço com o mesmo nome em `ws-client` no pacote `projecto.ws.client.service`. Caso exista uma classe abstracta base dos serviços, criar uma classe com o mesmo nome e responsabilidade. O método `action()` de cada novo serviço deverá seguir o seguinte padrão: importar os tipos dos *stubs* e da fábrica; pedir um *stub* à fábrica; preencher os argumentos; invocar o método remoto correspondente ao serviço que está a ser substituído; receber resultados. As excepções devem ser tratadas da seguinte forma (mapeamento de excepções de domínio, erro do serviço remoto, erro na criação de *stub*, erro de comunicação; respectivamente):

```
        } catch (ProjectoFault_Exception e) {
            // remote domain exception
            log.error(e);
            ProjectoDomainException ex =
ExceptionParser.parse(e.getFaultInfo().getFaultType(), e.getMessage());
            throw ex;
        } catch (ServiceError_Exception e) {
            // remote service error
            log.error(e);
            throw new RemoteServiceException(e);
        } catch (StubFactoryException e) {
            // stub creation error
            log.error(e);
            throw new RemoteServiceException(e);
        } catch (WebServiceException e) {
            // communication error (wrong address, connection closed)
            log.error(e);
            throw new RemoteServiceException(e);
        }
```

8. O componente `ws-client` está concluído. Fazer `ant build` para construir o JAR respectivo.

Variantes

StubFactory com configuração de endereço do Web Service

A fábrica de stubs deverá preencher o endereço do Web Service a contactar. Este endereço é um URL como: http://*host*:*port*/*Projecto*/endpoint. Uma forma de o fazer é ler o endereço a partir de um ficheiro de configuração:

1. Criar o ficheiro de configuração `config/resources/config.properties`, com a propriedade *`projecto`*`.ws.EndpointAddress=http://`*`localhost`*`:8080/`*`projecto-ws`*`/endpoint`, por exemplo.
2. Modificar a fábrica de *stubs*, acrescentando o método `getPortUsingConfig()`, que vai ler o parâmetro de configuração.
3. Modificar os serviços para chamarem o método `getPortUsingConfig()` em vez do `getPort()`.

StubFactory com localização de Web Service a partir do UDDI

A abordagem é semelhante à anterior, ou seja, modificar a fábrica de *stubs*. Nesta variante pretende-se obter o endereço do Web Service a partir de um servidor UDDI.

Criar um teste de invocação do Web Service

Para verificar que o componente `ws-client` está correctamente construído poderá ser útil fazer uma verificação:

1. Criar a classe: *`projecto`*`.ws.client.TestConsole`. Criar um método `public static void main(String[] args)` que cria e invoca *stubs*, fazendo: importar os tipos dos *stubs* e da fábrica; pedir um *stub* à

fábrica; preencher os argumentos; invocar o método remoto correspondente ao serviço que está a ser substituído; receber resultados. Este teste vai permitir verificar se o componente ws está a funcionar correctamente.

2. Acrescentar a propriedade `run.main-class` ao `build.xml` para que a classe de verificação seja executada quando se faz `ant run`.
3. Fazer `ant run` para executar a verificação.

Alterar processo de construção de aplicação para usar domínios remotos

Estratégia

A divisão de uma aplicação em várias aplicações distribuídas que cooperam para fornecer a mesma funcionalidade implica obrigatoriamente alterações ao à importação de classes de serviços e ao processo de construção da aplicação original.

Passos básicos

1. Criar uma nova base de dados para a aplicação `web` original. Editar o `build.properties` para mudar o nome da base de dados.
2. No código, alterar a utilização dos serviços locais (pacote `service`) por serviços remotos (pacote `ws.client.service`).
3. Abrir o `build.xml` da aplicação `web` original.
4. Alterar as dependências de código em `jar.external`: substituir o JAR do `core` pelo JAR do `ws-client`.
5. Alterar o *target* `-replace-hibernate-custom-tokens(dir)` para que apenas sejam feitos os *mappings* para as classes do domínio da aplicação `web`, excluindo os *mappings* do domínio remoto.
6. Fazer `ant generate-db-schema` e restantes tarefas relacionadas com o preenchimento da base de dados.
7. Instalar e usar a nova versão da aplicação.

Alterar processo de teste de aplicação para usar domínios remotos

Estratégia

Qualquer aplicação necessita de ser testada para garantir que se comporta de acordo com o desejado. Uma bateria de testes que verifique o comportamento da aplicação e que possam ser corridos automaticamente facilita a detecção de erros. Ao distribuir a aplicação pretende-se manter o comportamento de teste, mas é necessário ter em conta que passam a haver múltiplas aplicações. Para que a bateria de testes se mantenha operacional é necessário garantir que se consegue definir o estado inicial de todas as aplicações, para cada caso de teste. Deste modo, é necessário usar uma única base de dados.

Passos básicos

1. Confirmar que todas as aplicações que compõem o sistema - `web`, `ws`, etc - estão operacionais e distribuídas. Nomeadamente, confirmar as dependências `jar.external` (`ws-cliente` e não `core`) e os *mappings* hibernate (apenas as classes do domínio local)..
2. Criar ou escolher uma base de dados nova, para testes. Esta base de dados de testes deverá ser diferente das habitualmente usadas por cada aplicação.
3. Editar os ficheiros `build.properties` de cada aplicação para que todas passem a usar a base de dados de testes.
4. Para correr os testes de apenas um domínio, fazer `ant run-tests` no respectivo `core`.
5. Para correr os testes de toda a aplicação, fazer `ant run-tests` no componente `web` da aplicação principal.
6. Após a realização dos testes, voltar à configuração anterior dos ficheiros `build.properties` de cada aplicação.