JAVA Eğitimleri

Google Web Toolkit ile Vaadin Bileşenleri Geliştirme

harezmi bilişim çözümleri

Client Side Widget Oluşturma

- **GWT ile** yeni bir Vaadin **UI bileşeni** oluşturma iki ana adımda gerçekleşir
  - İlk adımda **client side bir widget** oluşturulur
  - İkinci adımda ise bu widget **server side bir bileşen ile eşleştirilir**
- **Sadece istemci tarafında** çalışan widget da geliştirmek mümkündür
- Client side widget'lara topluca **widgetset** adı verilir

Client Side Widget Oluşturma

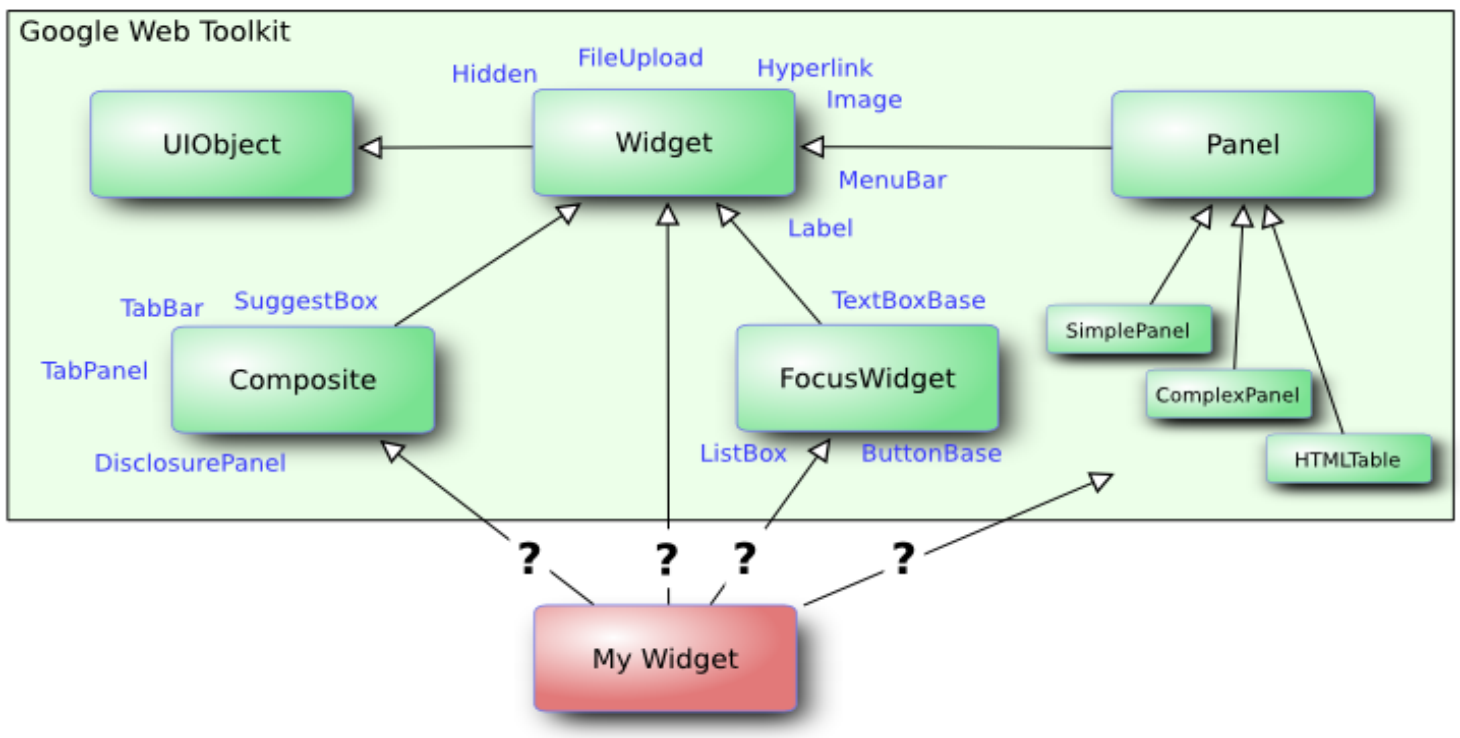
- Yeni widgetset basitçe **DefaultWidgetSet'i inherit ederek** oluşturulur
- Widgetset oluşturmak için öncelikle **module descriptor dosyası**na ihtiyaç vardır

Widgetset Module Descriptor

PetClinicWidgetSet.gwt.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE module PUBLIC
  "-//Google Inc.//DTD Google Web Toolkit 1.7.0//EN"
  "http://google-web-toolkit.googlecode.com/svn/tags/1.7.0/distro-
source/core/src/gwt-module.dtd">

<module>

    <inherits name="com.vaadin.DefaultWidgetSet" />

</module>
```

Client Side Widget Oluşturma



- Bütün widget'lar **Widget** sınıfından veya bunun alt sınıflarından türerler

Client Side Widget Oluşturma

- Client side widget sınıfları module descriptor ile **aynı paketin altında**, **client** isimli bir pakette yer almalıdırlar

```
package vaadin.client;

import com.google.gwt.user.client.ui.Label;

public class MyWidget extends Label {
    public static final String CLASSNAME = "mywidget";

    public MyWidget() {
        setStyleName(CLASSNAME);
        setText("Hello World!");
    }
}
```

Sadece İstemci Taraflı Uygulamalar

- Sunucu tarafında hiçbir UI bileşen karşılığı olmayan, tamamen client side widget'lardan oluşan uygulamalardır
- Bu tür uygulamalara **module** adı verilir

```
package vaadin.client;

import com.google.gwt.core.client.EntryPoint;
import com.google.gwt.user.client.ui.RootPanel;

public class HelloWorld implements EntryPoint {
    @Override
    public void onModuleLoad() {
        RootPanel.get().add(new MyWidget());
    }
}
```

Bu modüllerin bir **EntryPoint** sınıfı olur. Bu sınıfın **onModuleLoad()** metodu JS tarayıcı'da yüklenirken çalıştırılır

Bu kod Vaadin tarafında **UI** sınıfına ve **init()** metoduna benzetilebilir

Client Side Module Descriptor ve EntryPoint

PetClinicWidgetSet.gwt.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE module PUBLIC
  "-//Google Inc.//DTD Google Web Toolkit 1.7.0//EN"
  "http://google-web-toolkit.googlecode.com/svn/tags/1.7.0/distro-
source/core/src/gwt-module.dtd">

<module>

    <inherits name="com.vaadin.DefaultWidgetSet" />


    <entry-point class="vaadin.client.HelloWorld"/>

</module>
```

Widgetset Derleme

- Client widgetset'in kullanılabilmesi için **öncelikle derlenmesi** gerekir
- Derleme IDE içindeki bir **plugin**, **maven** veya **ant script**'i ile gerçekleştirilebilir
  - **mvn vaadin:compile** ile derleme yapılabilir

Client Side Uygulamanın Yüklenmesi ve Çalıştırılması

- Derleme işleminden sonra **widgetset dizini** altında **javascript dosyaları** oluşmuştur
- Bu dosyalardan **nocache.js** uzantılı javascript dosyası bir **HTML sayfası içerisinden yüklenerek** çalıştırılabilir

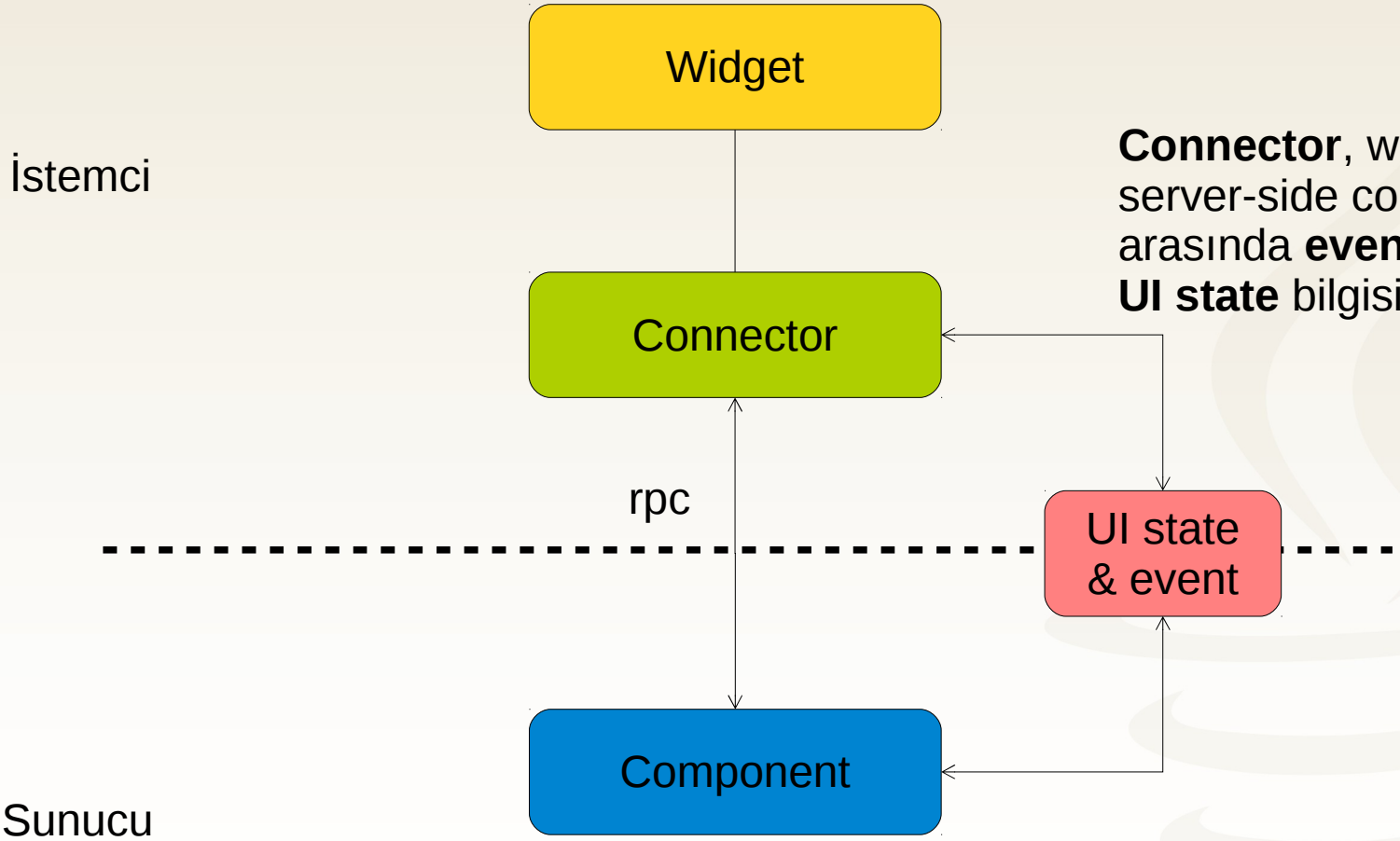
Client Side Uygulamanın Yüklenmesi ve Çalıştırılması

```
<html xmlns="http://www.w3.org/1999/xhtml">
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8"/>

        <style type="text/css">
            .mywidget {
                background-color: red;
            }
        </style>

        <link type="text/css" rel="stylesheet"
            href="VAADIN/themes/petclinic/styles.css"></link>
    </head>

    <body class="petclinic">

        <script type="text/javascript"
src="VAADIN/widgetsets/vaadin.PetClinicWidgetSet/vaadin.PetClinicWidget
Set.nocache.js">
        </script>
    </body>
</html>
```

Sunucu Tarafında Bir Bileşenle Entegrasyon

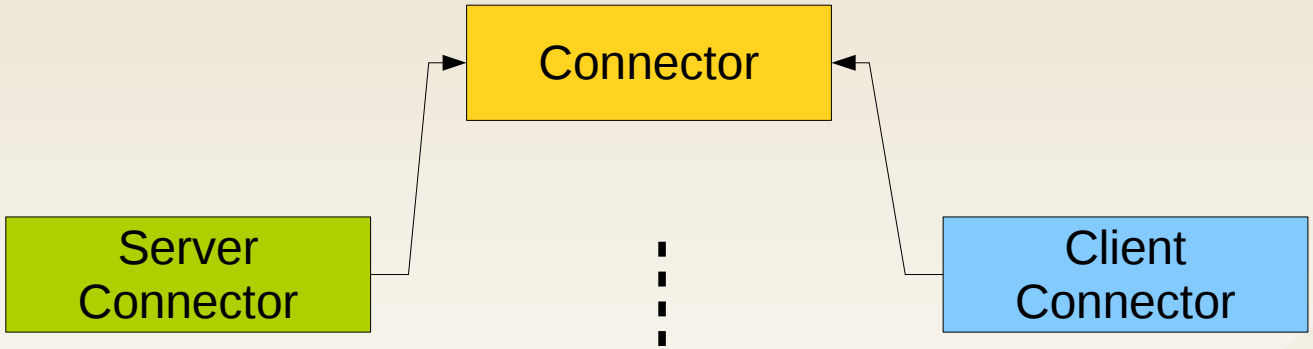
- Client side **widget** ile server side **UI component**'i arasındaki iletişim **connector** nesne üzerinden gerçekleşir
- Connector **istemci tarafında** bir sınıftır
- İki bileşen arasında **state paylaşımı**nı ve **event'lerin sunucu tarafına** gitmesini sağlar

Sunucu Tarafında Bir Bileşenle Entegrasyon



Connector, widget ile server-side component arasında **event** ve **UI state** bilgisini yönetir

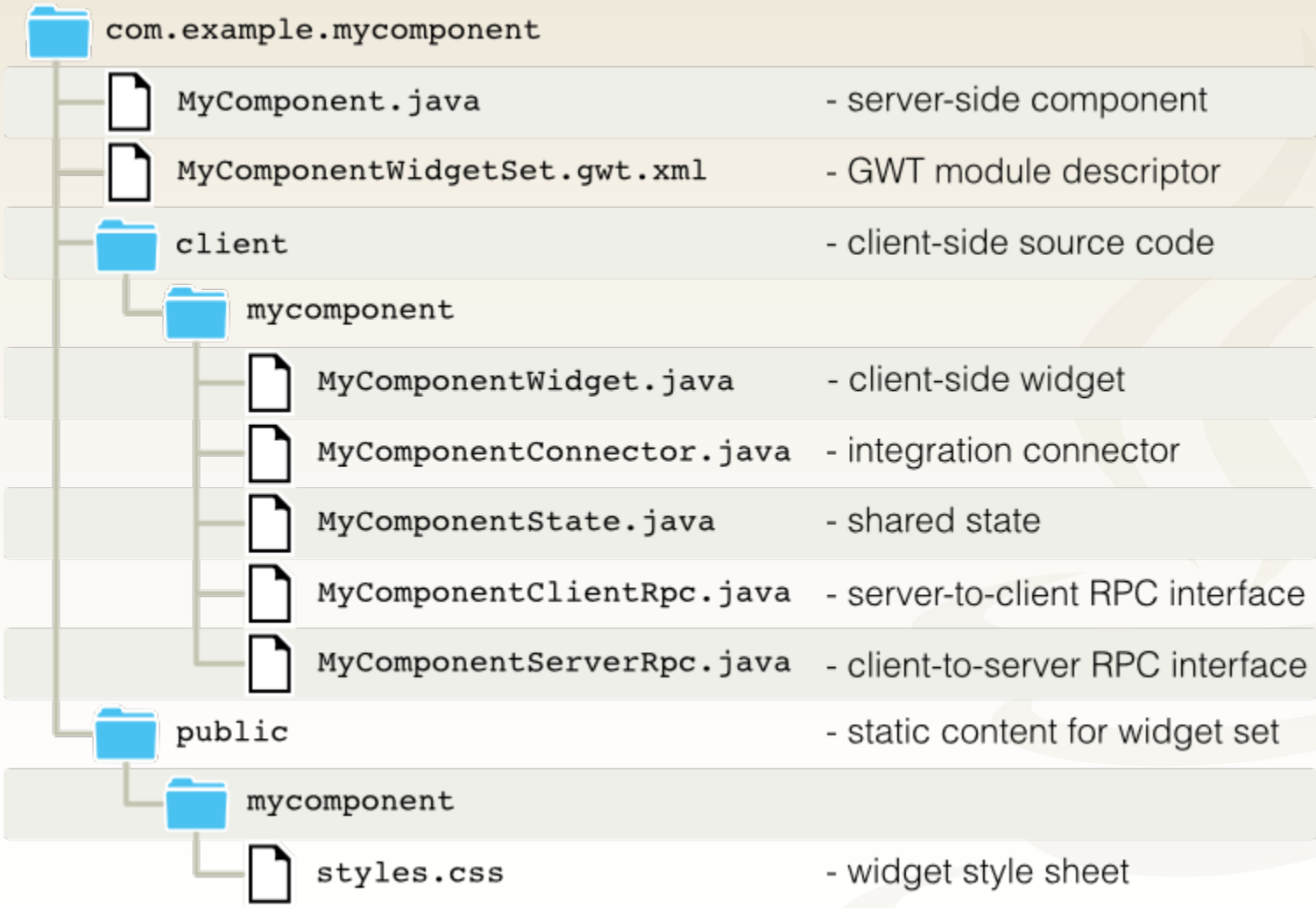
Connector Hiyerarşisi



- **İstemci** tarafında yazılır
- Sunucu tarafı ile iletişim sağlar
- Widget'ın dışında **ayrı bir sınıf** olarak implement edilir
- **@Connect** anotasyonu ile ilişki kuracağı server side bileşen belirtilir

- **Sunucu** tarafında yazılır
- İstemci tarafı ile iletişim sağlar
- **Bütün server side Vaadin UI bileşenleri** aynı zamanda ClientConnector'dür

Client ve Server Side Bileşen Dizin Yapısı



Shared State Sınıfının Oluşturulması

```
package vaadin.client;

import com.vaadin.shared.AbstractComponentState;

public class MyComponentState extends AbstractComponentState {

    private static final long serialVersionUID = 1L;

    private String text;

    public String getText() {
        return text;
    }

    public void setText(String text) {
        this.text = text;
    }
}
```

State sınıfı da client side widget ile aynı paket altında olmalıdır

İçerisinde sadece state tutulur, kesinlikle iş mantığı olmamalıdır

Field'lar public olabilir veya setter/getter'lar tanımlanabilir

Server Side Bileşenin Oluşturulması

Server side component herhangi bir paket altında olabilir

```
import vaadin.client.MyComponentState;

import com.vaadin.ui.AbstractComponent;

public class MyComponent extends AbstractComponent {

    public MyComponent() {
        getState().setText("Hello world from server side!");
    }

    @Override
    protected MyComponentState getState() {
        return (MyComponentState) super.getState();
    }
}
```

Server side component'in görevi herhangi bir iş mantığını çalıştırmanın yanı sıra, client side widget ile **state senkronizasyonu** da yapmaktır

Connector Sınıfının Oluşturulması

```
package vaadin.client;

@Connect(MyComponent.class)
public class MyComponentConnector extends AbstractComponentConnector {

    @Override
    public void onStateChanged(StateChangeEvent stateChangeEvent) {
        super.onStateChanged(stateChangeEvent);
        String text = getState().getText();
        getWidget().setText(text);
    }

    @Override
    public MyWidget getWidget() {
        return (MyWidget) super.getWidget();
    }

    @Override
    public MyComponentState getState() {
        return (MyComponentState) super.getState();
    }
}
```

@Connect anotasyonu sunucu tarafındaki bileşeni tanıtır

Connector sınıfının görevi UI interaction'larından doğan event'leri ele almak ve herhangi bir state değişikliğini widget'a yansıtmaktır

State değişikliği farklı yöntemlerle de ele alınabilir

State Değişikliklerinin Ele Alınması: @OnStateChange

```
@Connect(MyComponent.class)
public class MyComponentConnector extends AbstractComponentConnector {

    ...

    @OnStateChange("text")
    void updateText() {
        final String text = getState().getText();
        getWidget().setText(text);
    }
}
```

State değişikliğine uğrayan her bir property için ayrı ayrı handler metotlar tanımlanabilir

Sadece belirli birkaç property state'i değişiyorsa ve bu değişimler birbirlerinden bağımsız ise faydalıdır

onStateChanged() metoduna gerek kalmaz

State Değişikliklerinin Ele Alınması: @DelegateToWidget

```
public class MyComponentState extends AbstractComponentState {

    @DelegateToWidget
    private String text;

    ...
}
```

Text property'sindeki herhangi bir değişikliğin widget sınıfında aynı isimdeki property'ye otomatik olarak aktarılmasını sağlar

Eğer widget sınıfındaki property'nin ismi farklı ise @DelegateWidget anotasyonuna widget property'sinin ismi belirtilebilir

```
public class MyComponentState extends AbstractComponentState {

    @DelegateToWidget("description")
    private String text;

    ...
}
```

İstemci – Sunucu Arasında RPC Çağrıları

JAVA Eğitimleri

ServerRpc

- **İstemci** tarafından yapılacak RPC çağrıları içindir
- **ServerRpc** arayüzü istemci tarafında yazılır
- İstemci tarafında proxy instance'ı **AbstractConnector.getRpcProxy()** ile elde edilir
- Sunucu tarafındaki karşılığı **AbstractComponent.registerRpc()** ile tanıtılır

ClientRpc

- **Sunucu** tarafından yapılacak RPC çağrıları içindir
- **ClientRpc** arayüzü sunucu tarafında yazılır
- Sunucu tarafında proxy instance'ı **AbstractComponent.getRpcProxy()** ile elde edilir
- İstemci tarafındaki karşılığı **AbstractConnector.registerRpc()** ile tanıtılır

İstemci – Sunucu Arasında RPC Çağrıları

- RPC çağrıları **state içermeyen event**'lerin (buton click gibi) iletiminde kullanılır
- Öncelikle **ServerRpc** arayüzünü extend eden bir arayüz tanımlanır
- Ardından **RpcProxy.create()** ile bir proxy yaratılır
- Son adımda **proxy instance üzerinden** RPC çağrısı yapılır

ServerRpc Arayüzünün Geliştirilmesi

```
package vaadin.client;

import com.vaadin.shared.communication.ServerRpc;

public interface MyComponentServerRpc extends ServerRpc {

    public void clicked(String buttonName);

}
```

Metot return tipi void olmalıdır

Metot overload yapılamaz

Input parametrelerin tipi primitif java tipleri, bunların wrapper sınıfları, String, List, Set, Map gibi collection'lar, array'ler, Vaadin Connector ve bazı internal sınıflar olabilir

İstemci Tarafından RPC Çağrısı Yapılması

```
@Connect(MyComponent.class)
public class MyComponentConnector extends AbstractComponentConnector {

    public MyComponentConnector() {

        RpcProxy.create(MyComponentServerRpc.class, this);

        getWidget().addClickHandler(new ClickHandler() {
            public void onClick(ClickEvent event) {
                final MouseEventDetails mouseDetails =
                        MouseEventDetailsBuilder
                            .buildMouseEventDetails(
                                    event.getNativeEvent(),
                                    getWidget().getElement());

                MyComponentServerRpc rpc =
                        getRpcProxy(MyComponentServerRpc.class);


                rpc.clicked(mouseDetails.getButtonName());
            }
        });
    }
}
```

Sunucu Tarafında RPC Çağrısının Karşılanması

```
public class MyComponent extends AbstractComponent {

    private MyComponentServerRpc rpc = new MyComponentServerRpc() {
        private int clickCount = 0;

        public void clicked(String buttonName) {
            ++clickCount;
            Notification.show("Clicked " + buttonName + " " + clickCount
                    + " times!");
        }
    };

    public MyComponent() {
        registerRpc(rpc);
    }

    ...
}
```

Component Extension Kabiliyeti

- Inheritance kullanmadan mevcut UI bileşenlerine **yeni kabiliyetler ekle**meyi sağlar
- Böylece ihtiyaca göre bir UI bileşenine **birden fazla extension** eklenebilir
- UI bileşeninin nasıl extend edileceğine extension bileşeni karar verir
- Genellikle **AbstractExtension** sınıfından türerler ve **extend()** isimli bir metoda sahiptirler

Component Extension Geliştirme

- Extension'ların **istemci tarafında widget karşılığı yok**tur
- Sadece bir tane **extension connector** sınıfı vardır
- İhtiyaca göre extension connector ortak **state bilgisi**ni sunucu tarafındaki extension bileşen ile paylaşabilir
- Yada **RPC çağrısı** yapabilir

Component Extension Geliştirme

Sunucu tarafındaki extension sınıfı

```
public class CapsLockWarning extends AbstractExtension {

    public void extend(PasswordField field) {
        super.extend(field);
    }
}
```

Component Extension Geliştirme

Extension connector'ler widgetset içerisinde yer almalıdırlar. Bu yüzden client paketi altındadır.

Sunucu tarafındaki extension sınıfını tanıtır

```
package vaadin.client;

@Connect(CapsLockWarning.class)
public class CapsLockWarningConnector extends AbstractExtensionConnector {

    @Override
    protected void extend(ServerConnector target) {
        final Widget pw = ((ComponentConnector)target).getWidget();

        final VOverlay warning = new VOverlay();
        warning.setOwner(pw);
        warning.add(new HTML("Caps Lock is enabled!"));

        pw.addDomHandler(new KeyPressHandler() {
            public void onKeyPress(KeyPressEvent event) {
                if (isEnabled() && isCapsLockOn(event)) {
                    warning.showRelativeTo(pw);
                } else {
                    warning.hide();
                }
            }
        }, KeyPressEvent.getType());
    }
    private boolean isCapsLockOn(KeyPressEvent e) {
        return e.isShiftKeyDown() ^ Character.isUpperCase(e.getCharCode());
    }
}
```

extend(..) metodu override edilerek extension kabiliyeti uygulanır

Parametre olarak extend edilen UI bileşenin connector'ü (PasswordFieldConnector) verilir

Component Extension Kullanımı

```
PasswordField password = new PasswordField("Password");

new CapsLockWarning(password);

layout.addComponent(password);
```

Normal UI bileşenine extension kabiliyetinin eklenmesini sağlar

Ekleme işlemi farklı biçimlerde yapılabilir:

Layout'a bileşenin kendisi eklenir

```
new CapsLockWarning().extend(password);
```

veya

```
CapsLockWarning.addTo(password);
```

kullanılabilir

İletişim

- Harezmi Bilişim Çözümleri
- Kurumsal Java Eğitimleri
- http://www.java-egitimleri.com
- info@java-egitimleri.com