Object Partners Training

Building Web Applications with Project Avatar

Enterprise Java, written in JavaScript

Object Partners

Code:
https://github.com/objectpartners/techtalk-avatar

Slides:
https://github.com/objectpartners/techtalk-avatar/slides.pdf

AVATAR

About Me

@danveloper

/danveloper

#editor

daniel.woods@objectpartners.com

Project Avatar

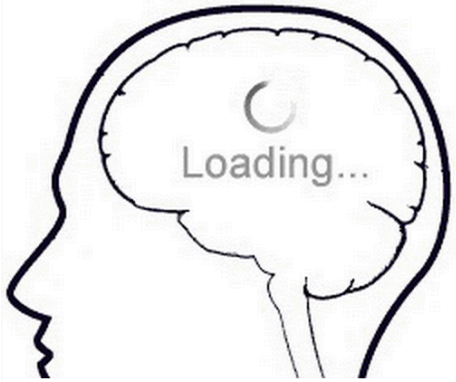
- Designed to take advantage of the "Future Tech" in Java EE 7
- Server-Side Events, WebSockets
- Polyglot Enterprise!
- Requires Java SE 8
- Node Services running inside of Glassfish
  - JavaScript Services!
- Avatar.js: No client-side JavaScript required
- … No *Java* Required.

… Wait …

- … No *Java* Required.

- … No *Java* Required.

- … No *Java* Required.

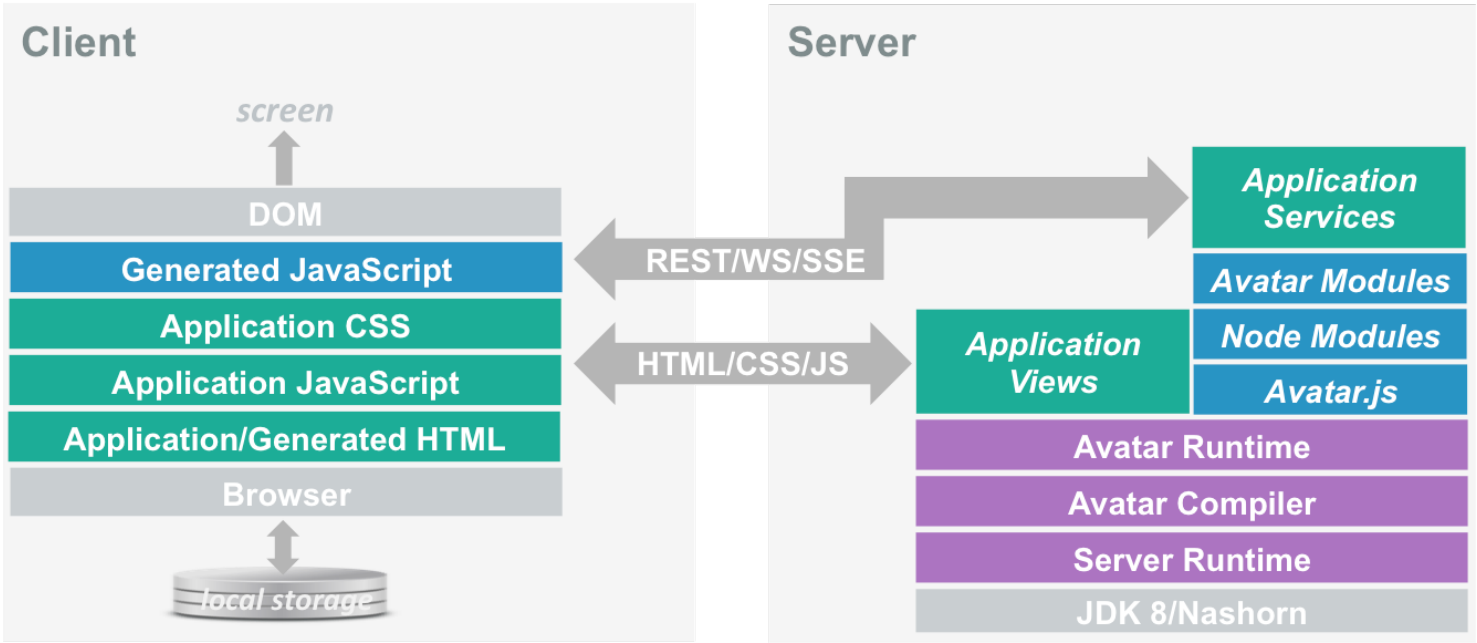


… What?

Project Avatar

- ***Requires Java SE 8***
  - For Nashorn ECMAScript Compiler
- Server-side components written in JavaScript, compiled to Java
- Client-side components written with Java EL, compiled to JavaScript
- Full-stack JavaScript with Java 8, running on Java EE 7
- Server-side & Client-side components can be used together or on their own

Project Avatar



```
login
|-- WEB-INF
|   `-- web.xml
|-- avatar.properties
`-- service
    `-- src
        `-- main.js
`-- view
    `-- com.objectpartners.avatar
        `-- src
            `-- app.html
```

The View Layer

- Organized Layout Components:
  - `tabContainer, stackContainer, contentPane`
- Theming:
  - JQuery UI (default) or Dijit
  - Bundled: Avatar (default), Redmond, Smoothness
  - Custom themes possible
    - Copy custom theme to:
      - `[app-dir]/view/bin/css/[theme]/jquery.ui.theme.css`
    - Update `avatar.properties`
      - `theme=[theme]`

The View Layer

- "Just Enough JavaScript"
  - Really only need to know how to define a View Model
- View Models
  - Apps are driven through model change events
  - local, rest, websocket, or push
- Data Interactions managed through Expression Language Expressions

View Models are just JavaScript Objects

```
<script data-model="local" data-instance="message">
    /* World's sickest View Model */
    var Message = function() {
        this.content = new Date();

        this.update = function() {
            this.content = new Date();
        };
    }
</script>
```

Java EL Expressions Drive View Model Interactions

```
<h1>#{message.content}</h1>
<button onclick="#{message.update()}">Update</button>
```

The View Layer

Remote View Models are backed by a remote endpoints

```
<script data-model="rest">
    var Person = function() {
        this.name = '';

        this.hello = function() {
            return this.name ? "Hello, " + this.name + "!" : "";
        };

        this.get = function() {
            this.$get();
        };
    }
</script>
<script data-type="Person"
        data-instance="person"
        data-url="rest/person"></script>
```

The URL `rest/person` is where we will get instances of the model.

The View Layer

Remote View Models can be used for seamless interface with their endpoint

```
<script data-model="socket">
    var HelloMessage = function() {
        this.name = '';

        this.send = function() {
            this.$send(this.name);
            this.name = '';
        };
    }
</script>
<script data-type="HelloMessage"
        data-instance="msg"
        data-url="websockets/hello"></script>
```

The `send` method on the `msg` instance will send the object's `name` property to the server's `websockets/hello` websocket endpoint, no other code needed.

The Service Layer

Services and endpoints are defined in server-side JavaScript

+ Snippet from `[app]/service/src/main.js`:

```
avatar.registerRestService({ url: 'rest/person'}, function () {
  this.$onGet = function (request, response) {
    return response.$send({name: LatestPerson.name});
  };
});

avatar.registerSocketService({ url: 'websockets/hello'}, function () {
  this.superOnMessage = this.$onMessage;

  this.$onMessage = function(ctx, msg) {
    LatestPerson.setName(msg);
    this.superOnMessage.call(this, ctx, msg);
  };
});
```

No other work is needed to build the URLs into the application

AVATAR

- View widgets define "pages"

```
<div data-widget="view"
     id="landingPage"
     data-title="Welcome to My Website!!"
     data-main="true"
     data-controller="vc">
    <h2>Who Are You?</h2>
    <form>
        <label for="name">Name: </label>
        <input id="name" data-value="#{user.name}"><br/>
        <button id="welcome"
                onclick="#{avatar.navigateTo('#accountPage')}">
            Go To Your Page!</button>
    </form>
</div>
```

Navigate between "pages" with the `avatar` module, using JQuery-like selector expression

AVATAR

Models, Views, and … ?

Single Page Apps

- Controllers.

```
<script data-controller="uiController" data-instance="vc">
    var ViewController = function() {
        this.onShow = function(node, instance) {
            if (['accountPage', 'manageProfile'].indexOf(node.id)>-1
                    && !instance.user.name) {
                avatar.navigateTo('#landingPage');
            }
        }
    };
</script>
```

Overload the `onShow` method to handle user experience events

Doing something useful…

Persistence

- JavaScript support for DataProviders
  - FileDataProvider
  - JPADataProvider

Persistence

- JPADataProvider uses Java EE, `persistence.xml`
  - Backed by EclipseLink
  - Specify “providers” that link to a PU/Entity
  - Async calls for getting entities and collections
  - Super easy to build endpoints that tie to the persistence model

- Building a REST API backed by Oracle

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.0" xmlns="http://java.sun.com/xml/ns/persistence" xmlns:xsi="http://www.w3.org/2001/
    <persistence-unit name="mem" transaction-type="RESOURCE_LOCAL">
        <provider>org.eclipse.persistence.jpa.PersistenceProvider</provider>
        <mapping-file>META-INF/orm.xml</mapping-file>
        <exclude-unlisted-classes>false</exclude-unlisted-classes>
        <properties>
            <property name="javax.persistence.jdbc.driver" value="oracle.jdbc.OracleDriver"/>
            <property name="javax.persistence.jdbc.url" value="jdbc:oracle:thin:@192.168.1.102:1521:xe"/>
            <property name="javax.persistence.jdbc.user" value="dan"/>
            <property name="javax.persistence.jdbc.password" value="test1234"/>
        </properties>
    </persistence-unit>
</persistence>
```

Must go in `WEB-INF/classes/META-INF/persistence.xml`

- Building a REST API backed by Oracle

```
<?xml version="1.0" encoding="UTF-8"?>
<entity-mappings version="2.4" xmlns="http://www.eclipse.org/eclipselink/xsds/persistence/orm"
                 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                 xsi:schemaLocation="http://www.eclipse.org/eclipselink/xsds/persistence/orm
                 http://www.eclipse.org/eclipselink/xsds/eclipselink_orm_2_4.xsd">
    <entity class="rest.Person" access="VIRTUAL">
        <table name="PERSON" />
        <attributes>
            <id name="id" attribute-type="Long">
                <column name="id"/>
            </id>
            <basic name="firstName" attribute-type="String">
                <column name="first_name" />
            </basic>
            <basic name="lastName" attribute-type="String">
                <column name="last_name" />
            </basic>
            <basic name="twitter" attribute-type="String">
                <column name="twitter" />
            </basic>
            <basic name="github" attribute-type="String">
                <column name="github" />
            </basic>
        </attributes>
    </entity>
</entity-mappings>
```

Define the entity mappings that your REST API will use.

- Building a REST API backed by Oracle

```
var personProvider = new avatar.JPADataProvider(
  { persistenceUnit: "mem", createTables: "true", entityType: "rest.Person" });

avatar.registerRestService({ url: 'api/person'}, function () {
  this.$onGet = function (request, response) {
    var promise = personProvider.$getCollection().then(function(results) {
      response.$send({ people: results.data });
    }, function(error) {
      avatar.log("error");
      avatar.log(error);
    });
  };
});
```

Define the Collection API!

- Building a REST API backed by Oracle

```
avatar.registerRestService({ url: 'api/person/{id}'}, function () {
  this.$onGet = function(request, response) {
    personProvider.$get(this.id, function(error, person) {
      if (!person) {
        person = {};
      }
      response.$send(person);
    });
  };

  this.$onPut = function(request, response) {
    personProvider.$put(this.id, request.data, function(result) {
      response.$send(result);
    });
  };

  this.$onDelete = function(request, response) {
    personProvider.$delete(this.id, function(result) {
      response.$send(result);
    });
  };
});
```

Define the Item API!

Enterprise Integration

- Other Java EE Integration:
  - JMS
  - Publish/Subscribe Message Bus
    - Event-driven/Reactive Programming

Testing?

Testing

- For the client…
  - No Test Fixtures :-(
  - Generated code obscures your implementation
  - Probably can only reliably handle Functional Tests

Testing

- For the server…
  - No Test Fixtures :-(
  - JavaScript code is a participant in the container, so there's some hope there…
  - Projects like Arquillian can probably help bootstrap the testing environment

Opinions…

Opinions

- A very opinionated framework needs very verbose documentation.
- There is no community.

- GREAT to see Oracle embracing polyglot server-side development
- Future is bright for the JVM as a “Platform”

Object Partners

The End.

contact@objectpartners.com

AVATAR