Step 1: Overview and Testing Strateg

lunedì 9 febbraio 2026 10:06

Lo Scenario dell'Applicazione

Il tutorial è incentrato su un prototipo di bacheca (bulletin board) che viene preparato per il rilascio come prodotto.

- **Funzionalità:** L'app consente agli utenti di sfogliare i post e aggiungere le proprie offerte alla bacheca.
- **Stato Attuale:** Il punto di partenza è un'app minimalista con un set di base di test e funzionalità.
- **Obiettivo:** Implementerai le funzionalità richieste ed estenderai la copertura dei test per trovare proattivamente i bug e garantire la stabilità del codice durante gli aggiornamenti o il refactoring.

Struttura del Progetto e Strumenti

Il prototipo segue le best practice di SAPUI5, separando il codice produttivo (situato nella cartella `webapp`) dal codice non produttivo (situato nella sottocartella `test`).

- **Pagina di Ingresso:** Il file `webapp/test.html` funge da punto di ingresso principale, consentendo di aprire l'app con dati fittizi (mock data) o di eseguire la suite di test.
- **Home Page:** Il file `webapp/test/mockServer.html` inizializza SAPUI5, avvia il mock server e istanzia il componente dell'applicazione.
- **Mock Server:** Configurato in `webapp/localService/mockserver.js`, questo strumento simula un servizio backend reale utilizzando metadati e dati JSON locali. Include un ritardo configurabile per imitare i tempi di risposta realistici del server.
- **Unit Test:** Situati in `webapp/test/unit`, questi test utilizzano QUnit per convalidare la logica dell'applicazione.
- **Test di Integrazione:** Situati in `webapp/test/integration`, questi test utilizzano OPA5 per verificare le interazioni dell'utente e gli elementi dell'interfaccia utente (UI).
- **Suite di Test:** Il file `webapp/test/testsuite.qunit.js` elenca tutti i test del progetto e gestisce la configurazione.

Modello Dati

L'applicazione utilizza le seguenti entità OData:

- **Post:** Identificato da `PostID`, questa entità include proprietà come `Title` (Titolo), `Description` (Descrizione), `Price` (Prezzo), `Category` (Categoria) e `Contact` (Contatto).
- **Category:** Questa entità viene utilizzata per ordinare i post e contiene una proprietà `Name` (Nome).
- **Comment:** Identificato da `CommentID`, questa entità è collegata a un post tramite `ParentID` e include `Author` (Autore), `Date` (Data) e `CommentText` (Testo del commento).

Strategia di Test

Il tutorial sostiene una solida strategia di test per ridurre gli sforzi di test manuali e mantenere la qualità in cicli di sviluppo brevi.

- **La Piramide dei Test:** La strategia si basa sulla piramide dei test agile, dove gli unit test formano la base, seguiti dai test di integrazione e infine dai test di sistema o manuali.
- **Automazione:** L'obiettivo è automatizzare il maggior numero possibile di passaggi.
- **Dati Mock:** I test di integrazione si basano su dati fittizi locali per garantire affidabilità e indipendenza dalla disponibilità del sistema backend.

Prerequisiti e Configurazione

- **Conoscenze:** Dovresti avere familiarità con le basi del unit testing in JavaScript utilizzando QUnit.
- **Installazione:** Per configurare il progetto, scarica i file sorgente, estraili, esegui `npm install` per installare le dipendenze e `npm start` per avviare il server web.

Step 2: A First Unit Test

lunedì 9 febbraio 2026 10:25

Specifiche della Funzionalità

Il team di prodotto ha definito la seguente mappatura tra prezzi e stati semantici:

- **Prezzo < 50**: Stato "Success" (Verde).
- **Prezzo tra 50 e 249**: Stato "None" (Normale).
- **Prezzo tra 250 e 1999**: Stato "Warning" (Arancione).
- **Prezzo >= 2000**: Stato "Error" (Rosso).

Il Concetto di TDD

Secondo la metodologia TDD, si scrive il test automatico *prima* di implementare la funzionalità. Il ciclo prevede:

1. Scrittura di un test che fallisce (poiché manca l'implementazione).
2. Implementazione del codice minimo necessario per far passare il test.
3. Refactoring del codice per renderlo più elegante, mantenendo i test verdi.

Infrastruttura dei Unit Test

I test unitari si trovano nella cartella `webapp/test/unit`. Possono essere avviati aprendo il file `webapp/test/testsuite.qunit.html` nel browser e selezionando `unit/unitTests`.

Codifica del Test

Per prima cosa, aggiungiamo una funzione vuota `priceState` nel file `webapp/model/formatter.js` per definire l'interfaccia, ma senza implementarne la logica.

Successivamente, implementiamo i test nel file `webapp/test/unit/model/formatter.js`:

- **Creazione di una funzione di riuso**: Per evitare ripetizioni (principio "DRY"), creiamo `priceStateTestCase`, che chiama il formattatore con un prezzo specifico e confronta il risultato con lo stato atteso tramite un'asserzione `strictEqual`.
- **Definizione dei casi di test**: Utilizziamo QUnit`.test` per definire cinque scenari che coprano tutte le fasce di prezzo e i casi limite (come il valore 50).

Al termine di questo passaggio, eseguendo i test nel browser, i nuovi test risulteranno **rossi (failing)**. Questo è il comportamento previsto nel TDD: il messaggio di errore confermerà che la funzione restituisce un valore non corretto, spingendoci a scrivere l'implementazione nel passaggio successivo.

Step 3: Adding the Price Formatter

lunedì 9 febbraio 2026 10:32

1. Implementazione della Logica (`formatter.js`)

Nel file `webapp/model/formatter.js`, sostituiamo la funzione vuota con la logica reale per soddisfare le specifiche definite nei test. La funzione `priceState` accetta il prezzo (`iPrice`) come input e restituisce una stringa che rappresenta lo stato semantico basato su quattro casi gestiti tramite istruzioni `if/else`:

- **Success**: Se il prezzo è inferiore a 50.
- **None**: Se il prezzo è maggiore o uguale a 50 e inferiore a 250.
- **Warning**: Se il prezzo è maggiore o uguale a 250 e inferiore a 2000.
- **Error**: Se il prezzo è maggiore o uguale a 2000 (ramo `else`).

2. Verifica tramite Unit Test

Una volta salvata l'implementazione, è possibile eseguire la suite di test (`webapp/test/testsuite.qunit.html`) e selezionare `unit/unitTests`. Se la logica è corretta, i casi di test implementati nel passaggio precedente risulteranno ora corretti (verdi), confermando che la funzionalità è formalmente valida.

3. Aggiornamento dell'Interfaccia Utente (`Worklist.view.xml`)

Anche se i test passano, la formattazione non è ancora visibile nell'app perché i test unitari verificano la logica indipendentemente dall'interfaccia. Per visualizzare i colori, modifichiamo il file `webapp/view/Worklist.view.xml`:

- Individuiamo il controllo `ObjectNumber` all'interno dell'aggregazione `columns`.
- Aggiungiamo l'attributo `state`.
- Colleghiamo l'attributo allo stesso percorso dati del numero (`Price`) ma utilizziamo il nuovo formatter `.formatter.priceState` per determinare lo stato.

Risultato Finale

Eseguendo l'app tramite `webapp/test/mockServer.html`, i prezzi dei prodotti saranno visualizzati in **verde, nero, arancione o rosso** a seconda del loro importo, confermando che la logica è stata applicata correttamente alla UI.

Step 4: Testing a New Module

lunedì 9 febbraio 2026 10:48

Obiettivo e Strategia

L'obiettivo è implementare un pulsante nell'interfaccia utente (UI) per contrassegnare i post interessanti.

- **La Sfida:** Il pulsante nella UI si aspetta un valore booleano (premuto/non premuto), ma il modello dati utilizza una rappresentazione intera binaria (1 o 0).
- **La Soluzione:** Poiché è necessaria una logica di conversione bidirezionale (dal modello alla UI e viceversa quando l'utente clicca), non possiamo usare un semplice formattatore; implementeremo invece un **Custom Data Type** (Tipo di Dati Personalizzato).

1. Creazione del Guscio del Tipo (`FlaggedType.js`)

Nel file `webapp/model/FlaggedType.js`, creiamo la struttura del nuovo tipo estendendo `sap.ui.model.SimpleType`. Seguendo il metodo TDD, lasciamo inizialmente vuote le funzioni per far fallire i test:

- **`formatValue`**: Per convertire il valore del modello in visualizzazione UI.
- **`parseValue`**: Per convertire il valore della UI in valore del modello.
- **`validateValue`**: Per controllare errori di validazione (anche se qui non ci sono errori attesi).

2. Scrittura degli Unit Test (`FlaggedType.js`)

Creiamo il file di test `webapp/test/unit/model/FlaggedType.js`, mantenendo la stessa struttura delle cartelle del codice produttivo per facilitare l'associazione tra test e implementazione. Definiamo due moduli QUnit per verificare entrambe le direzioni della conversione:

- **Test di Formattazione (Model → UI):**
- Verifichiamo che il valore intero `1` venga convertito in `true`.
- Verifichiamo che altri valori (come `0` o numeri negativi) vengano convertiti in `false`.
- **Test di Parsing (UI → Model):**
- Verifichiamo che il valore booleano `false` venga convertito in `0`.
- Verifichiamo che il valore booleano `true` venga convertito in `1`.

Nei test, istanziamo il tipo (`new FlaggedType()`) e chiamiamo manualmente le funzioni `formatValue` o `parseValue`, confrontando il risultato con l'asserzione `strictEqual`.

3. Esecuzione dei Test

Infine, registriamo il nuovo modulo di test nel file `webapp/test/unit/unitTests.qunit.js` affinché venga eseguito automaticamente. Eseguendo i test ora, ci aspettiamo che **falliscano**, confermando che la logica di conversione non è ancora stata implementata (fase "Red" del TDD).

Step 5: Adding a Flag Button

lunedì 9 febbraio 2026 12:07

1. Implementazione della Logica (`FlaggedType.js`)

Nel file `webapp/model/FlaggedType.js`, inseriamo il codice che soddisfa i test unitari scritti in precedenza.

- **`formatValue`**: Converte il valore intero `1` (dal modello) in booleano `true` (per la proprietà `pressed` del pulsante).
- **`parseValue`**: Effettua l'operazione inversa quando l'utente clicca il pulsante, convertendo `true` in `1` e `false` in `0`.
- **`validateValue`**: Restituisce sempre `true` poiché non ci sono vincoli di validazione client-side complessi per un booleano.

Verifica: Se esegui ora la suite di test (`webapp/test/testsuite.qunit.html`), i test creati nello Step 4 dovrebbero passare con successo (diventare verdi).

2. Aggiornamento dell'Interfaccia (`Worklist.view.xml`)

Aggiungiamo una colonna alla tabella per ospitare il pulsante.

- Inseriamo un controllo `sap.m.ToggleButton`.
- Impostiamo l'icona su `sap-icon://flag`.
- **Binding Fondamentale**: La proprietà `pressed` viene collegata al campo `Flagged` del modello, specificando il tipo personalizzato: `type: '.types.flagged'`.

3. Collegamento nel Controller (`Worklist.controller.js`)

Affinché la vista possa utilizzare il tipo personalizzato `.types.flagged`, dobbiamo istanziarlo nel controller.

- Importiamo `FlaggedType`.
- Creiamo un oggetto `types` che contiene una nuova istanza: `flagged: new FlaggedType()`.

4. Abilitazione del Two-Way Binding (`Component.js`)

Questa è una modifica critica. I modelli OData in SAPUI5 hanno per default un binding "OneWay" (dal modello alla vista). Affinché il clic sul pulsante aggiorni il modello (e quindi i dati), dobbiamo abilitare il "TwoWay" binding.

- Nel metodo `init` del `Component.js`, aggiungiamo: `this.getModel().setDefaultBindingMode("TwoWay");`.

5. Test Manuale

Dopo aver aggiunto la stringa per il tooltip in `i18n.properties`, puoi avviare l'app tramite `webapp/test/mockServer.html`. Vedrai la nuova colonna con le bandierine; cliccandole, lo stato cambierà visivamente e il dato verrà aggiornato nel modello locale grazie alla logica implementata.

Step 6: A First OPA Test

lunedì 9 febbraio 2026 12:21

Strumenti e Struttura

Utilizziamo **OPA5** (One-Page Acceptance tests), uno strumento incluso in SAPUI5 che esegue i test nella stessa finestra del browser dell'applicazione. I test sono strutturati in due elementi principali:

- **Journeys (Viaggi):** Descrivono il flusso di interazione dell'utente (es. `WorklistJourney.js`) utilizzando il pattern "Given-When-Then".
- **Page Objects (Oggetti Pagina):** Incapsulano le azioni e le asserzioni specifiche per una vista (es. `pages/Worklist.js`).

Implementazione del Test

1. **WorklistJourney.js:** Aggiungiamo un nuovo test `opaTest` chiamato "Should be able to load more items".
- **Azione:** Simuliamo la pressione del pulsante "altro" (`When.onTheWorklistPage.iPressOnMoreData()`).
- **Asserzione:** Verifichiamo che la tabella mostri tutte le 23 voci presenti nei dati mock (`Then.onTheWorklistPage.theTableShouldHaveAllEntries()`).
- **Cleanup:** Spostiamo la distruzione dell'app (`iTeardownMyApp`) alla fine di questo nuovo test.

2. **pages/Worklist.js:** Definiamo le funzioni chiamate nel journey.
- **`iPressOnMoreData`:** Utilizza `waitFor` per trovare la tabella e l'azione `Press` per simulare il click sul trigger di caricamento.
- **`theTableShouldHavePagination`**: Verifica che la tabella abbia inizialmente 20 elementi usando il matcher `AggregationLengthEquals`.
- **`theTableShouldHaveAllEntries`**: Verifica che la tabella contenga tutti i 23 elementi previsti.

Step 7: Changing the Table to a Growing Table

martedì 10 febbraio 2026 11:47

Modifica del Codice (`Worklist.view.xml`)

Per attivare il caricamento dinamico di ulteriori elementi, è sufficiente modificare la configurazione del controllo `Table` nella vista XML:

- Impostiamo la proprietà **`growing`** su **`true`**.

Risultato

Dopo questa semplice modifica:

1. **App**: Eseguendo l'applicazione, la tabella mostrerà inizialmente solo 20 elementi. Scorrendo verso il basso o cliccando sul pulsante "More" (Altro), verranno caricati i restanti 3 elementi.
2. **Test**: Eseguendo nuovamente il test di integrazione (OPA5) creato nello Step 6, questo ora avrà successo perché il trigger per caricare più dati è presente e funzionante.

Step 8: Testing Navigation

martedì 10 febbraio 2026 11:58

Nuovo Journey: `PostJourney.js`

Creiamo un nuovo file `webapp/test/integration/PostJourney.js` per descrivere il percorso dell'utente. Questo journey include tre test principali:

1. **Navigazione al Dettaglio**: L'utente clicca su un elemento della lista e dovrebbe vedere la pagina del post con il titolo corretto (es. "Jeans").
2. **Navigazione Indietro**: L'utente preme il pulsante "Indietro" nella pagina del post e dovrebbe tornare alla lista (TablePage).
3. **Navigazione Avanti (Browser)**: L'utente preme il pulsante "Avanti" del browser e dovrebbe ritrovarsi nuovamente sulla pagina del post.

Aggiornamento dei Page Objects

Per supportare queste azioni, aggiorniamo e creiamo nuovi oggetti pagina:

- **`pages/Worklist.js`**:
- **Action `iPressOnTheItemWithTheID`**: Utilizza il matcher `BindingPath` per trovare e cliccare (`Press`) un `ColumnListItem` specifico basato sul suo ID (es. `/Posts('PostID_15')`).
- **Assertion `iShouldSeeTheTable`**: Verifica semplicemente che la tabella sia visibile per confermare il ritorno alla lista.
- **`pages/Post.js` (Nuovo)**:
- **Action `iPressTheBackButton`**: Trova il pulsante di navigazione nella pagina tramite ID e simula il click.
- **Assertion `theTitleShouldDisplayTheName`**: Verifica che l'header della pagina mostri il titolo atteso, confermando l'avvenuta navigazione.
- **`pages/Browser.js` (Nuovo)**:
- **Action `iPressOnTheForwardButton`**: Simula il pulsante "Avanti" del browser utilizzando l'API `history.forward()`, avvolta in una `waitFor` per garantire la sincronizzazione.

Configurazione Finale

Infine, aggiungiamo il nuovo `PostJourney` al file di configurazione `webapp/test/integration/opaTests.qunit.js` affinché venga eseguito insieme agli altri test.

Eseguendo i test ora, OPA attenderà invano che la pagina del post appaia, fallendo (comportamento atteso dato che la pagina non esiste ancora), ma fornendo una specifica chiara per l'implementazione nel prossimo step.

Step 9: Adding the Post Page

martedì 10 febbraio 2026 12:22

1. Configurazione del Routing (`manifest.json`)

Aggiungiamo una nuova rotta e un target nel descrittore dell'applicazione:

- **Route**: Definiamo il pattern `Post/{postId}`, dove `{postId}` è un parametro obbligatorio che passeremo durante la navigazione.
- **Target**: Associamo la rotta alla vista `Post` e impostiamo il `level` a 2 per gestire correttamente le animazioni di transizione.

2. Abilitazione della Navigazione (`Worklist.view.xml` e `Worklist.controller.js`)

- **Vista**: Nella tabella della Worklist, impostiamo il tipo degli elementi (`ColumnListItem`) su `Navigation` e assegniamo l'evento `press` alla funzione `.onPress`.
- **Controller**: Implementiamo la funzione `onPress`. Questa recupera il contesto dell'elemento cliccato (il `PostID`) e istruisce il router a navigare verso la rotta "post", passando l'ID come parametro.

3. Creazione della Vista di Dettaglio (`Post.view.xml`)

Creiamo una nuova vista minimale che mostra i dettagli essenziali:

- Utilizziamo una `FullscreenPage` con un pulsante "Indietro" (`showNavButton="true"`).
- Inseriamo un `ObjectHeader` per visualizzare il titolo e il prezzo del post, soddisfacendo le aspettative dei test scritti in precedenza.

4. Logica del Controller di Dettaglio (`Post.controller.js`)

Il nuovo controller gestisce il binding dei dati e la navigazione a ritroso:

- **onInit**: Crea un modello JSON locale (`postView`) per gestire lo stato della vista (es. indicatore di occupato) e si aggancia all'evento di routing `attachPatternMatched`.
- **_onPostMatched**: Quando la rotta viene colpita, estrae il `postId` dagli argomenti e collega (`bindElement`) la vista al percorso specifico del post (es. `/Posts('...')`). Gestisce anche gli eventi `dataRequested` e `dataReceived` per mostrare/nascondere l'indicatore di caricamento.
- **onNavBack**: Utilizza la funzione helper `myNavBack` (ereditata dal `BaseController`) per tornare alla Worklist.

Con queste implementazioni, i test di integrazione sulla navigazione (OPA) dovrebbero ora passare con successo.

Step 11: Testing User Input

martedì 10 febbraio 2026 12:43

Modifica del Journey (`WorklistJourney.js`)

Aggiungiamo un nuovo test case chiamato `"Should be able to search for items"` al file `WorklistJourney.js`.

- **Azione:** Simuliamo la ricerca della stringa "Bear" chiamando `When.onTheWorklistPage.iSearchFor("Bear")`.
- **Asserzione:** Verifichiamo che la tabella mostri un solo risultato chiamando `Then.onTheWorklistPage.theTableHasOneItem()`.
- **Cleanup:** È fondamentale spostare il passaggio `iTeardownMyApp()` alla fine di questo nuovo test, rimuovendolo dal test precedente, per evitare che l'app venga distrutta prima dell'esecuzione dell'ultimo test.

Aggiornamento del Page Object (`pages/Worklist.js`)

Per supportare il nuovo test, implementiamo le funzioni necessarie nel Page Object, importando la nuova dipendenza `sap/ui/test/actions/EnterText`.

1. **Azione `iSearchFor`**:
- Utilizza l'azione `EnterText` per inserire il testo nel controllo con ID `searchField`.
- La funzione `waitFor` attende che il campo sia visibile e interagibile prima di digitare il testo; in caso contrario, il test fallisce con un messaggio di errore.
2. **Asserzione `theTableHasOneItem`**:
- Utilizza il matcher `AggregationLengthEquals` per verificare che l'aggregazione `items` della tabella contenga esattamente **1** elemento, come previsto dai dati mock per la ricerca "Bear".
- Se la condizione è soddisfatta, viene eseguita un'asserzione `Opa5.assert.ok` per confermare il successo.

Convenzioni

Una regola importante da seguire in OPA5 è che **le azioni non devono mai contenere asserzioni QUnit**; le verifiche devono essere sempre demandate alla fase di asserzione (blocchi `assertions`).

Step 12: Adding a Search

mercoledì 11 febbraio 2026 09:33

Modifiche alla Vista (`Worklist.view.xml`)

Per consentire l'interazione, aggiorniamo la barra degli strumenti (`headerToolbar`) della tabella:

- Inseriamo un `ToolbarSpacer` per posizionare il campo di ricerca a destra.
- Aggiungiamo il controllo `SearchField` con l'ID `searchField`.
- Colleghiamo l'evento `search` alla funzione del controller `.onFilterPosts`.

Logica del Controller (`Worklist.controller.js`)

Nel controller implementiamo la logica di filtraggio, simile a quella vista nel passaggio 24 del tutorial "Walkthrough":

1. **Dipendenze**: Importiamo i moduli `sap/ui/model/Filter` e `sap/ui/model/FilterOperator`.
2. **Funzione `onFilterPosts`**:

- Recupera la stringa di query (`sQuery`) dai parametri dell'evento.
- Se la query è presente, crea un oggetto `Filter` che verifica se la proprietà **"Title"** *contiene* (`FilterOperator.Contains`) la stringa cercata.
- Recupera il binding degli elementi della tabella (`items`) e
- applica l'array di filtri tramite o`Binding.filter(aFilter)`.

Aggiornamento del Contatore (`onUpdateFinished`)

Il codice fornito include anche la logica per aggiornare dinamicamente il titolo della tabella (es. mostrando il numero di risultati trovati):

- Utilizza o`Event.getParameter("total")` per ottenere il numero totale di elementi dopo il filtro.
- Aggiorna la proprietà `/worklistTableTitle` nel modello `worklistView` solo se il caricamento della lista è definitivo.

Step 13: Testing User Interaction

mercoledì 11 febbraio 2026 09:57

Aggiornamento del Journey (`PostJourney.js`)

Estendiamo il file `PostJourney.js` aggiungendo un nuovo test case chiamato `"Should select the statistics tab"`.

- **Azione:** Simuliamo il click sulla scheda delle statistiche chiamando `When.onThePostPage.iPressOnTheTabWithTheKey("statistics")`.
- **Asserzione:** Verifichiamo che il contatore delle visualizzazioni sia visibile chiamando `Then.onThePostPage.iShouldSeeTheViewCounter()`.
- **Cleanup:** È fondamentale spostare la chiamata `.and.iTeardownMyApp()` dall'ultimo test precedente alla fine di questo nuovo test, per evitare che l'app venga chiusa prima di aver verificato il tab delle statistiche.

Aggiornamento del Page Object (`pages/Post.js`)

Implementiamo le funzioni necessarie nel Page Object per supportare il nuovo test.

1. **Azione `iPressOnTheTabWithTheKey`**:
- Utilizza `waitFor` per cercare un controllo di tipo `sap.m.IconTabFilter`.
- Usa il matcher `Properties` per trovare l'elemento specifico che ha la proprietà `key` uguale a quella passata (es. "statistics").
- Esegue l'azione `Press` per simulare il click dell'utente sulla scheda.
2. **Asserzione `iShouldSeeTheViewCounter`**:
- Utilizza `waitFor` per cercare il controllo con ID `viewCounter`.
- **Nota sulla Visibilità:** In OPA5, `waitFor` controlla implicitamente che il controllo sia renderizzato e visibile. Pertanto, nella funzione di `success`, è sufficiente inserire un'asserzione generica come `Opa5.assert.ok(true, ...)` per soddisfare i requisiti di QUnit.

Convenzioni

Le azioni in OPA non devono mai contenere asserzioni QUnit; queste devono rimanere separate nella fase di "Assertions".

Step 14: Adding Tabs

mercoledì 11 febbraio 2026 10:55

Modifiche alla Vista (`Post.view.xml`)

Modifichiamo la vista di dettaglio aggiungendo un controllo `sap.m.IconTabBar` che suddivide il contenuto in due sezioni principali:

1. **Scheda "Info" (`key="info"`)**:
- All'interno di questa scheda inseriamo un `sap.ui.layout.form.SimpleForm`.
- Il form visualizza i campi per la data (`Timestamp`) e la descrizione (`Description`), mappati ai dati del modello.
2. **Scheda "Statistics" (`key="statistics"`)**:
- Questa è la scheda specifica che abbiamo testato nel journey OPA precedente.
- Al suo interno posizioniamo un controllo `Text` con l'ID **`viewCounter`**.
- In questo esempio semplice, il testo è statico ("Viewed 55555 times"), ma in un'applicazione reale sarebbe collegato al modello dati.

Aggiornamento delle Risorse (`i18n.properties`)

Infine, aggiungiamo le stringhe di testo mancanti nel file `i18n.properties` per fornire le etichette corrette per il titolo dell'oggetto, la data e la descrizione.

Step 15: Writing a Short Date Formatter Using TDD

mercoledì 11 febbraio 2026 10:57

1. Preparazione e Scheletro

Prima di scrivere la logica, prepariamo i file necessari.

Registrazione del Test: Aggiorniamo `webapp/test/unit/unitTests.qunit.js` per includere il nuovo modulo di test.

Scheletro dell'Implementazione (`webapp/model/DateFormatter.js`): Creiamo un oggetto vuoto che estende `sap.ui.base.Object`.

Primo Test "Esistenza" (`webapp/test/unit/model/DateFormatter.js`): Verifichiamo semplicemente che la classe sia istanziabile. Inizialmente fallirà perché il file non è ancora caricato/definito correttamente, poi passerà.

2. Iterazione 1: Gestione Input Vuoto

Test (Rosso): Scriviamo un test che verifica che, passando `null` o nessuna data, il formattatore restituisca una stringa vuota.

Implementazione (Verde): Implementiamo il metodo `format` nel `DateFormatter.js` affinché restituisca `""` di default.

3. Iterazione 2: "Oggi" e Dependency Injection (Locale)

Vogliamo che se la data è oggi, venga mostrata l'ora (es. "12:05 PM").

Test (Rosso): Scriviamo un test che passa una data e si aspetta l'ora formattata.

- **Problema:** Se il test gira su un browser italiano, si aspetterebbe "12:05". Se gira in USA, "12:05 PM".
- **Soluzione (Dependency Injection):** Passiamo la `Locale` ("en-US") al costruttore del formattatore nel test per forzare il formato americano.

  Implementazione (Verde): Nel costruttore di `DateFormatter.js`, istanziamo `DateFormat.getTimeInstance({style: "short"}, oProperties.locale)`. Il metodo `format` ora usa questa istanza.

4. Refactoring: Setup Comune

Poiché ogni test avrà bisogno di un'istanza di `DateFormatter` configurata con la locale "en-US", spostiamo questa inizializzazione nella funzione `beforeEach` del modulo QUnit. Questo rende il codice dei test molto più pulito.

5. Iterazione 3: "Ieri" e Dependency Injection (Tempo)

Vogliamo che se la data è ieri, venga restituito "Yesterday".

Test (Rosso):

- **Problema:** "Ieri" è relativo. Se esegui il test oggi, "ieri" è una data diversa da quella che sarà domani.
- **Soluzione (Dependency Injection):** Iniettiamo una funzione `now` nel costruttore che restituisce un timestamp fisso (es. `2015/02/14 14:00`). Così, per il test, "Oggi" è sempre il 14 Febbraio 2015.
- Il test verifica che passando il 13 Febbraio 2015, il risultato sia "Yesterday".

  Implementazione (Verde):

1. Salviamo `oProperties.now` nel costruttore.
2. Implementiamo un metodo helper `_getElapsedDays` che calcola la differenza in giorni tra

`this.now()` e la data passata.
3. Se la differenza è 1, restituiamo "Yesterday".

6. Iterazione 4: Giorni della Settimana (< 7 giorni)

Se sono passati meno di 7 giorni, vogliamo il nome del giorno (es. "Sunday").

Test (Rosso): Passiamo una data che è 6 giorni prima della nostra data "fissa" (14 Feb 2015 -> 8 Feb 2015 = Domenica). Ci aspettiamo "Sunday".

Implementazione (Verde):

4. Nel costruttore, aggiungiamo `this.weekdayFormat` usando `DateFormat.getDateInstance({pattern: "EEEE"})`.
5. Nel metodo `format`, aggiungiamo la condizione: `else if (iElapsedDays < 7) { return this.weekdayFormat.format(oDate); }`.

7. Iterazione 5: Date Vecchie (> 7 giorni)

Per tutte le altre date, vogliamo la data completa (es. "Dec 5, 2011").

Test (Rosso): Passiamo una data vecchia di 7 o più giorni (es. 7 Feb 2015) e ci aspettiamo il formato medio (es. "Mar 7, 2015" in en-US).

Implementazione (Verde):

6. Nel costruttore, aggiungiamo `this.dateFormat` usando `DateFormat.getDateInstance({style: "medium"})`.
7. Nel metodo `format`, aggiungiamo l' `else` finale che usa questo formattatore.

Risultato Finale

Abbiamo creato un formattatore robusto e testato al 100%. Grazie alla **Dependency Injection**, i test sono "Black Box": non sanno come funziona il formattatore, ma controllano che dato un input (Data + Locale + Ora Corrente Fittizia), l'output sia esattamente quello atteso.

Step 16: Adding the Date Formatter

mercoledì 11 febbraio 2026 11:12

1. Aggiornamento della Vista (`Post.view.xml`)

Modifichiamo la scheda "Info" all'interno della `IconTabBar`. Invece di mostrare il timestamp grezzo, applichiamo il formattatore:

- **Binding**: Modifichiamo il controllo `Text` per utilizzare una sintassi di binding complessa:
  XML
  ```
  <Text text="{
      path: 'Timestamp',
      formatter: '.formatter.date'
  }"/>
  ```
- **Nota sulla Sintassi**: Il punto iniziale in `.formatter.date` indica a SAPUI5 di cercare la funzione `formatter` all'interno del controller della vista, e successivamente la funzione `date`.

2. Aggiornamento del Formattatore (`model/formatter.js`)

Il file `formatter.js` funge da ponte tra la vista e la logica complessa del `DateFormatter`.

- **Importazione**: Aggiungiamo `sap/ui/demo/bulletinboard/model/DateFormatter` alle dipendenze.
- **Implementazione**: Aggiungiamo la funzione `date`. Qui avviene la magia della **Dependency Injection** in produzione:
  JavaScript
  ```
  date: function(date) {
      return new DateFormatter({now: Date.now}).format(date);
  }
  ```